AN INVESTIGATION OF MOBILE AD-HOC NETWORK PERFORMANCE WITH COGNITIVE ATTRIBUTES APPLIED

STEWART JOHN BLAKEWAY

A thesis submitted in partial fulfilment of the requirements of Liverpool John Moores University for the degree of Doctor of Philosophy

June 2015

Table of Contents

Abstract

Mobile Ad-Hoc Networks (MANETs) are known for their versatility, which is they are capable of supporting many applications. In addition to this versatility MANETs are quick to deploy without need for an existing predefined communications infrastructure. However, although the lack of infrastructure allows for the quick deployment of the data communications network, it adds many factors that hinder packet delivery. Such hindrances occur because of the dynamic topology caused by the mobility of the nodes which results in link breakages. Routing protocols exist that attempt to refresh available routes; however, this is after link breakages have occurred. The nodes also usually have constrained resources (i.e. energy source and limited bandwidth).

This thesis presents a novel approach of network behaviour and management by implementing cognitive attributes into a MANET environment. This allows an application to better meet its mission objectives, decreases the end-to-end delay, and increases packet delivery ratio. The network is able to make observations, consider previous actions and consequences of the actions, and make changes based on the prior knowledge and experience. This work also shows how the network can better utilise limited resources such as bandwidth allocation by applying cognitive attributes.

Simulations conducted show promising results and prove that an increase in network performance is possible if adopting a cross-layered approach and allow the network to manage and to ‘think’ for itself. Various simulations were run with various scenarios and results are presented without cognition applied, with partial cognition applied and with full cognition applied. A total of 52 simulations were run and from this the results were

compared and contrasted. The analysis shows that cognitive attributes does increase network performance in the majority of applications.

Acknowledgements

I would like to acknowledge my supervision team for their guidance, support and patience, Dr. Askwith and Professor Merabti. I extend this acknowledgement to Dr. Kirpichnikova who has acted as my internal supervisor during the last twelve months. Dr. Kirpichnikova has shown me the direction, kindness and has given me the occasional kick during this difficult year.

Liverpool Hope University who funded this degree have been instrumental in the success of this work and I would like to personally thank Professor Atulya Nagar, Dean of Sciences at Liverpool Hope University for his support during the process including through some difficult transitional years.

Also, a huge thank you to my family for all their support and kindness, in particular during the last year of this thesis. It was difficult not spending as much time with my Children as I once did but their understanding, love and proudness kept me motivated.

I would finally like to thank God for keeping me strong and healthy.

List of Figures

List of Tables

Abbreviations

AM	Amplitude Modulation
AODV	Ad-Hoc On Demand Distance Vector
ASK	Amplitude Shift Keying
BFSK	Binary Frequency Shift Keying
Bps	Bits per second
BPSK	Binary Phase Shift Keying
CCK	Complementary Code Keying
CN	Cognitive Network
CSMA/CA	Carrier Sense Multiple Access / Collision Avoidance
DBPSK	Differential Binary Phase Shift Keying
DQPSK	Differential Quadrature Phase Shift Keying
DSDV	Destination-Sequenced Distance Vector
DSSS	Direct-sequence spread spectrum
FAT	File Allocation Table
FHSS	Frequency Hopping Spread Spectrum
FM	Frequency Modulation
FSK	Frequency Shift Keying
FTP	File Transfer Protocol
GHz	Gigahertz
HTTP	Hyper Text Transfer Protocol
Hz	Hertz
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
IR	Infrared
ISP	Internet Service Provider
MANET	Mobile Ad-Hoc Network
Mbps	Megabits per second
MHz	Megahertz per second
MIMO	Multiple Input Multiple Output
NCC	Network Command Centre
NODE	Communication Point
NS2	Network Simulator 2
NS3	Network Simulator 3
NTFS	New Technology File System
OFDM	Orthogonal frequency-division multiplexing
PSK	Phase Shift Keying
QAM	Quadrature amplitude modulation
QoS	Quality of Service
QPSK	Quadrature Phase-Shift Keying
RF	Radio Frequency
SMTP	Simple Mail Transfer Protocol
TELNET	Terminal over Network
VANET	Vehicular Ad Hoc Network
VoIP	Voice over IP
Wi-Fi	Wireless Fidelity

Chapter 1 Introduction

In recent years networking technologies have become an integrated part of life and over the last decade networking communication technology has made its way into mainstream society, long gone are the days where networks were only found in corporate companies consisting of mainframes and terminals connected via a guided medium. Today networking communication technologies consist of an infrastructure supporting hundreds or thousands of clients, or in case of the Internet 1,668,870,408 clients, which is a growth of 362% since the year 2000, Miniwatts (2009). With the huge growth of users on the Internet and the new type of media available online; for example streaming video, more and more data is traversing the Internet backbones and being passing through the routers that connect the high speed trunks.

It can be said that traditional data communication networks are reactive, that is they react to tackle problems in the network after the problem has occurred. (Santivanez, 2007) explains further and claims that networks are still not aware of their state or needs and do not have knowledge of their goals and ways to achieve them, nor are they able to reason for their actions.

However, the evolution of hardware and the use of techniques to manipulate data have allowed the network infrastructure to continue to operate and to meet the demands of its many applications.

The new type of applications and new type of services are driving the research of new and innovative communication network designs, one of which is coined 'Cognitive Networks' (Mahmoud, 2007). Networks must be able to adapt to the traffic patterns created by the applications, in particular when delay cannot be tolerated.

In addition to this, wireless technologies have increased dramatically over the last decade and many mobile devices support multiple forms of wireless connectivity.

The popularity in wireless technology and this applied to wireless devices has spurred the development of Mobile Ad-Hoc Networks (MANETs). MANETs face all the challenges associated with that of data communications plus further challenges of wireless data communications. This research focuses on MANETs and answers the two following overarching questions:

1. Can methodologies typically found in cognitive networks be applied to a MANET?
2. If the above is true, will applying cognitive methodologies to a MANET aid the overall performance of the network and help meet mission objectives?

The above questions are quite vague. In order to answer these questions the identification of the issues that are likely to affect the performance of a data communications network are discussed. In section 1.5 the formulation of the aims and objectives and present these in much more detail.

The research focuses mostly on congestion, routing and link breakage. These three main areas have been identified because each area may have an adverse effect on the overall performance of a data communications network.

This thesis presents a solution to help alleviate against these issues by applying the concept of 'thought' and cognition to the network.

This research shows that the implementation of a cognition cycle will allow the data communication network to be aware of the requirements of the network, the current state of the network and the predicted state of the network.

When considering the state of the network consideration is given to: local density, mobility patterns, energy consumption and congestion.

In addition this research has taken the above points further by allowing the network to predict future network state and mobility patterns of the nodes of interest. In this case primarily, additional support is offered to the source nodes. In this thesis it is shown that this prediction scheme mitigates against network failures and increases network performance.

1.1 Mobile Ad-Hoc Networks

This section justifies why there is a need for a MANET that is able to adapt to current network conditions and predict future network state, thus, it begins with a very brief discussion that details the history of networking technology, the trends of network traffic, network infrastructure and the typical application of the network before moving to a discussion detailing the need for a MANET and then concluding with the justification for the need for a cognitive MANET.

1.1.1 Traditional Application of Data Communication Networks

Originally, data that was delayed on the network did not affect the services required by the applications because these services were not real-time, for example email was unaffected, as opposed to applications that required a certain quality of service such as Voice over IP. As data communication networks grew in popularity so did the applications and services offered. These applications and services generated more data, which in turn meant that the communications infrastructure saw a traffic growth that was of a high and steady rate (Ash and Ferguson, 2001).

The traditional data communication networks continued to provide for these services due to the relatively small image sizes and compression techniques.

Increasingly, network applications saw the introduction and the use of multimedia applications, which placed massive amounts of data on to the network infrastructure in relation to what it had seen previously. Although congestion and delay were a concern, the performance did not affect the operation of the application – albeit there was some delay.

New services and applications were developed that changed how network traffic was used - no longer could these applications patiently wait for data that was delayed on the network. This new generation of applications not only required a certain level of service from the network in order to function correctly but also placed a higher amount of traffic onto the data communications network that had not been seen before. This further stretched the resources of the network infrastructure. Applications and/or services include but are not limited to: Voice over IP (VoIP), streaming audio and streaming video which require a certain Quality of Service (QoS) to function correctly.

Demand for and popularity of the new networking applications among everyday consumers was growing, which in turn increased this new type of network traffic.

A case study of two major European operators shows that the majority of Internet traffic is now streaming video (Feknous et al. 2014).

In order for the network to facilitate the communication of data to service these applications advances were made to the overall networking infrastructure. For example, hubs began to be replaced with intelligent hub or switches, which not only offered greater link speeds but allowed for medium access to be managed better. Another example is the multiplexing techniques used in order to allow the combination of several data packets to facilitate the sharing of a communications link.

1.1.2 Traditional Mobile Networks

The use of mobile devices is another example of advancement in the field of data communication networks. Mobile devices initially were connected to the wired infrastructure by means of a communications port and cable (early laptops). The flexibility that those early mobile devices offered spurred investments into this technology and as a result a rapid advancement in the form of wireless communication technologies was witnessed.

Most mobile devices today offer some form of wireless connectivity and are able to connect to a wireless access point, which is in turn connected to the wired infrastructure. It is estimated that by 2017 there will be approximately 7 trillion wireless devices serving 7 billion people or 1000 wireless devices per individual (Sørensen and Scouby, 2009). This does not mean each individual will carry 1000 wireless devices but that wireless devices are becoming commonplace in everyday life and interaction with wireless devices is continuing to increase. An example of this might be the relatively new wireless contact payment services provided by most debit and credit cards for low cost purchases, which allows payment from a debit or credit card by using close proximity wireless sensors as opposed to using the chip and pin technology.

Wireless data communication is typically last hop, meaning that the wireless communication is established between the wireless device and the access point. Most wireless communication is in the form of radio waves (other wireless data communication is achieved by direct beams of light or laser).

1.1.3 Tethering

In some cases tethering is used to facilitate wireless communications, for example, if a laptop is not in range of a wireless access point but can connect to a mobile cellular device and take advantage of a distributed network using cellular technology. This is also typically last hop (to the mobile cellular device) but the cellular device acts as an intermediate forwarding node and facilities bi-direction communication between the wireless device and the cellular network. Figure 1 - Tethering shows a wired connection between the laptop and the mobile device that uses cellular communication technology. The connection between these devices could also be wireless using Wi-Fi (Wireless-Fidelity) standards or via Bluetooth technology. Thus mobile devices today have multiple forms of wired and wireless connectivity that interoperate seamlessly.

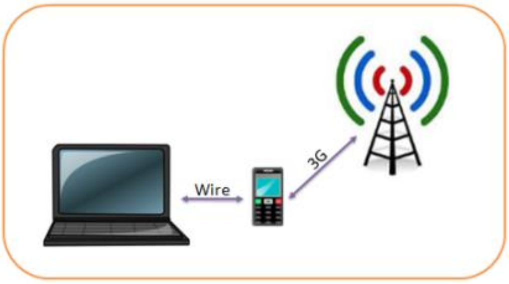


Figure 1 - Tethering

There are cases where a data communications infrastructure as described above is not available; for instance, in rural areas such as Rhiwhiriaeth (a beautiful rural setting in Wales) where there is no predefined infrastructure.

There are also situations where the current infrastructure has been damaged and would either be too costly (cost could be attributed to time as well as financial) or infeasible to deploy because of restrictions that could be attributed to landscape or other environmental issues.

1.1.4 MANETs

MANETs are much quicker to deploy than that of deploying a data communications network with an infrastructure. This led to the research and development of Mobile Ad-Hoc Wireless Networks. The primary aim of a MANET is the ability to quickly deploy a data communications network where there is no predefined infrastructure, a non-functional infrastructure, a restricted infrastructure or where there may be concerns of security and it may not be desirable to use the existing infrastructure.

MANET research began in the early 1970's and was sponsored by the Defence Advanced Research Agency (DARPA). This led to the development of larger scale networks that were proposed by DARPA in 1983 and were termed as a Survivable Adaptive Network (SURAN).

Ad-hoc networking technology was adopted by the IEEE committee that assigned the 802.11 standard in 1999 (Zhai and Fang, 2003). The latter standard allowed wireless devices the capability to transmit on a channel that is part of the unlicensed radio spectrum. The electromagnetic spectrum is discussed in more detail in the section 2.2.1.

1.1.5 MANET Application

A data communications network is infeasible unless there are applications that make use of the data traversing the network. A MANET has many applications including Military, Emergency Rescue, Local and Personal Area Network (PAN).

Military application allows for communication between soldiers, vehicles and command centres. A MANET would allow for the deployment of a communications network where there is no existing predefined infrastructure or if the infrastructure has been destroyed either by natural phenomenon (such as earthquake or flood) or by malicious intent (such as an attack, deliberate sabotage or explosion).

It can be also used in the cases when security of the existing infrastructure is of concern such as on a battlefield. MANETs can be used by the Emergency services (rescue or enforcement), as there might not be any predefined infrastructure. The emergency services would benefit from this communications network because the network is quick to deploy and the time saved in establishing communication could potentially save lives.

On a local level MANET can autonomously link to facilitate the sharing of information; for example, conference or classroom type scenarios. This can be extended to local civilian environments such as bus routes, taxi cab, sports stadia and others. A MANET can also be used as a PAN to facilitate intercommunication between various devices.

Similarly to the traditional network discussed earlier MANETs now support a various number of applications and services as demonstrated above. However, unlike the traditional wired infrastructure which has adapted to the changes by updating or enhancing the hardware on the infrastructure, applications support is more difficult in the case of a MANET because each device may be mobile which causes unpredicted changes in topology, each device normally has a self-contained energy source which could deplete, each device normally serves a primary application (for example cellular communication) and each device requires undertaking the role of a router to forward packets to the intended destination or towards the intended destination.

1.5 Research Aims and Objectives

It is the aim of this research to present a framework that will incorporate some of the Cognitive Network (CN) approaches in to a MANET environment. The overarching research questions are stated earlier, however, in order to better understand the aim that this thesis addresses the issues just discussed are addressed by the following;

Aim 1. Allowing the network to be conscious of its own state and environment

Aim 2. Allow for autonomous decision making behaviour to better utilise available resources in order to make the network more resilient to changes in topology and congestion.

a. routing protocols in use

b. the unlicensed electromagnetic spectrum

c. node mobility

Aim 3. Allow for the network to evaluate and reflect on the decisions made in order to make better informed decisions

Each aim mentioned above in this research has been addressed in this thesis, Aim 1 is achieved by the design and the implementation of the design of a network with cognitive attributes that proves the network can be conscious of its own state and its own environment. This is discussed in chapter 6.

Aim 2 is addressed by allowing the network to make decisions based on information gathering in Aim 1. Chapter 6 describes how this aim was met and the results presented in Chapter 7 prove that autonomous decision making does make better use of available resources.

The design presented in Chapter 6 discusses how the network reflects on previous decisions. In addition to this, detail is given that shows how the network can make better informed decisions based on the actions taken in Aim 2. In Chapter 7 the results show that the reflection and evaluation of the decisions made aids the network in making better informed future decisions.

The above aims are achieved by meeting the following objectives of this thesis:

Objective 1. To discuss the additional elements required for cognition to be applied to a Mobile Ad-Hoc Network.

Objective 2. To design a suitable testbed in order to test the implemented elements described in Objective 1, whilst ensuring that the solution is rigorous, transparent, and replicable for the testing of the scientific theories. It will be assumed that this objective is achieved when the MANET testbed has been configured using current wireless communication standards found in MANET technology.

Objective 3. To enhance the design of the MANET to incorporate cognitive attributes to the designed MANET in Objective 2. Discussion of the essential necessary features that allow one to call a network cognitive is presented. Once these features are listed with brief explanations, each attribute is discussed separately. It is assumed that this objective is achieved when all the cognitive attributes are ready to be implemented into the testbed.

Objective 4. To apply cognitive attributes to a MANET as discussed in Objective 3 into the testbed implemented in Objective 3 by enhancing the testbed and by making the MANET cognitive. It is assumed that this objective is achieved when simulations of a MANET with cognitive attributes applied are ready to be run and when the configuration settings used in the code run correctly.

Objective 5. To design and to run the admissible simulation test models which are based on a realistic scenario.

Objective 6. To design and to run simulation models that use well-known mobility models, such as Random Waypoint Model.

Objective 7. To analyse the results obtained in Objective 5 and Objective 6; to compare, contrast and discuss performance of each model. In particular, discussion is given that shows the differences between the simulation runs without cognition applied against the same model with partial cognition and full cognition applied. The latter will demonstrate whether the solution was or was not successful in regards to increasing network performance with cognition applied.

Objective 8. A reflection will be presented on how the performance of the MANET was affected with partial and full cognitive attributes. The identification of when performance increased or decreased is shown with discussion of the reasons for the change in the performance across each of the models.

Objective 9. In relation to objective 8, a formulation of a list of future objectives on cognitive MANET improvement is presented, in particular identifying any improvements that could further enhance MANET performance.

By correctly utilising the resources the network is able to reduce congestion, strengthen links (in terms of predicting link breakages and taking appropriate action to reintroduce a link or reinforce that link) and better meet mission objectives. Each of these aims is discussed in more detail in the feasible solution and conceptual design section. The next sections discuss traditional communication networks (wired and wireless) before defining the characteristics of a MANET and a Cognitive Network and how they differ to the traditional networks.

1.6 Overview of the Application of Cognition

This section presents an overview of how cognition was applied to the MANET infrastructure. In particular, discussion of an election process, the learning stage of a newly elected manager, the experience manager node, the Network Command Centre (NCC), link breakages, protocol performance, congestion, the concept of experience and actions that can be requested by the manager node(s), is detailed.

1.6.1 Election of a Management Node

An election process is performed in order to choose a suitable management node for any source node(s) that are transmitting. The election process considers various factors when determining suitable candidates. Any node that is in range of the source node becomes a candidate for the election process. The election process evaluates the characteristics of each candidate in order to select the most suitable management node.

One consideration taken into account is the distance between the candidate and the source node or more importantly the predicted time that the source node will be in range of the candidate. This consideration is important because it would be unwise to choose a candidate that is only momentarily in range of the source node.

The next consideration is the current energy level of the energy source for the candidate node because it is desirable to choose a management node with a higher life expectancy over that of another candidate node.

Next the election process considers if the candidate is currently contributing in the forwarding of packets for the source node. If the candidate node is part of the route that the packets take it would be desirable to choose a candidate node that is not part of the route, this is because the elected management node will be consuming resources when evaluating network performance, predicting performance and making plans.

Each of the considerations carries a weight and a score is calculated for each candidate. The candidate with the highest score becomes a management node for that source. There could be a situation where a management node manages multiple sources. This is allowed if a more suitable candidate is not available.

1.6.2 Infancy period of a Management Node

When the network is newly constructed there will be no prior knowledge of past events. Therefore the management node is allowed to make recommendations based on network state alone. At this stage; the management node is allowed to make a recommendation for change based on the current network state. This is what the management node can currently observe during the initial stage of management. The recommended decision is based on performance metrics and density; the most severe that no packets are being received at the destination, if this is the case then the management node will assume that a link breakage has occurred.

Should the topology be changing rapidly the management node will assume that the current protocol in use should be changed. Otherwise the management node will recommend switching to another operating channel. The management node during the initial management period may make the wrong decision, for example a change in topology that increases density could be attributed to congestion. It is intended that the management node will learn from its mistakes and becomes more experienced with the more recommendations it makes.

1.6.3 Experienced Management Nodes

Although the management node will make an initial decision based on current network state, as memories accumulate the management node may change this decision based on the passed memories and experience learned.

Therefore, the management node does not make a recommendation on network state if there are similar network states in memory. The consequence of each previous decision is evaluated for those network states that are similar to the current network state.

A good decision made in one situation may not necessarily be a good decision in another situation. The same is true for bad decisions. Therefore as part of the evaluation of past events the management node will also take into consideration the state of the network. Using the current state of the network the management node will look for decisions and the consequences of those decisions for situations that are similar to the current situation. For example the management node will take into account the current density of the network when choosing the most appropriate course of action.

1.6.4 Network Command Centre

Conflicting recommendations from managers wanted to be avoided (i.e. some managers wish to change frequency whilst others might wish to change routing protocol) so a Network Command Centre (NCC) is implemented. The NCC will acknowledge request that are received by the manager nodes but may not act on them immediately. At an appropriate time (normally determined by a prior action) the NCC will consider all the requests made from the management nodes. There is no priority in the Network (i.e. a manager does not have higher priority than another manager) and all Management nodes are treated equally.

The first decision that the NCC will make is which action to invoke from three available options. This is relatively straight forward; the NCC tallies the number of requests for each option. Next the NCC will determine if the action is currently available based on resource utilisation or allocation. If the action is currently not available the NCC will choose the next most popular action request based on the earlier tally.

1.6.5 Link Breakages

When a link breakage occurs the receive rate at the destination node will fluctuate until an alternative route is found. This may result in lost packets that have to be retransmitted. In the worst case scenario there may not be a viable route from the source node to the destination node via intermediate links. When this occurs the receive rate at the destination maybe significantly reduced, with a rate of zero received packet for this source node. In this case a request to move a controllable node might be initiated.

1.6.6 Protocol Performance

Different routing protocols perform differently in different situations, for example, in a relatively less dense environment the DSDV routing protocol might perform better than the AODV routing protocol. When receive rates decrease but do not drop completely it might be applicable to switch to another routing protocol, one that is more suited to this current topology. This is evident in research that compared AODV, DSDV and DSR which showed DSDV performed better in high density networks or with strict requirement on time, whereas DSR performs well in smaller networks and AODV is more adaptable in networks with high throughputs (Rahman et al, 2012).

1.6.7 Congestion

When more nodes enter the same transmission space, contention for that space increases. More nodes might want to transmit which would begin to cause congestion. To alleviate congestion a group of nodes serving an application could switch to another transmission frequency. This would separate them from the other nodes that may have been causing congestion.

1.6.8 Concept of Experience

The reader of this thesis has been introduced to possible causes that may decrease the performance of the network. In addition to this possible solutions are presented to mitigate against these causes. However, as in life, there may be many causes that produce the same effect. For example; Stress, Anger, Poor Posture, Perfume, Bad Weather, Grinding Teeth, Bright Lights, Chemicals, Sexual Intercourse and Ice Cream are ten causes of headaches (NHS Choices, 2013). Choosing the correct course of action is not always straight forward. As experienced is gained the manager node will remember the actions and the consequence of those actions, in this relatively simple example one might say "every time I eat Ice Cream I get a headache", course of action do not eat Ice Cream. This is known because of the memory of past events and the consequences. This information helps to evaluate and consider causes of certain effects and in some cases (if the effect was not desirable) how to prevent similar situations occurring again. In this case, network performance may degrade for multiple reasons but give the same effect. Therefore a concept of experience is used, the concept of memory and the concept of reflection (evaluation) in order to choose a more appropriate cause of action rather than the ones identified previously.

1.6.9 Actions

Change Channel

The NCC can instruct a group of nodes to change the current transmission channel. All manager nodes are instructed to back off from making further requests until a settle down period has elapsed.

Switch Routing Protocol

The NCC can instruct a group of nodes to switch from one routing protocol to another. As with change channel the manager nodes are instructed to back off from making further requests until the network has stabilised.

Move Controllable Node

In this research a controllable node is a node that can be requested to move to a new location by a management node. In terms of application a non-controllable node could be soldiers on the ground who have mission objectives, and a controllable node could be thought of as a robot or drone used to facilitate the soldiers mission objectives.

This action is relatively more complex to achieve than the previous two actions. When it has been determined that a controllable node is required, various actions will have to be performed in order to determine the likelihood that the controlled node will be of any use. For example: Firstly, there may not be any controllable nodes (which is referred to as partial cognition) in the network. Secondly, the controllable nodes might be busy performing another action. Thirdly, it could be that the source node and destination node will be in range before the controllable node gets to a position of where it would be of any help. Fourthly, the controllable node may never get into a suitable position (i.e. it cannot catch the destination or source node up).

In order to perform the above checks the mobility prediction scheme is used. It is very probable that the source node, the management node and the controllable node are in motion to new destinations. Therefore, the positions of all three nodes need to be predicted. This is further complicated because multiple predictions for different points in time for all three nodes will need to be calculated in order to perform the above checks.

If the NCC is satisfied that the controllable node will be of use, the NCC will determine where the predicted midpoint of the source node and destination node is. The midpoint is based on a time in the future. The time in the future is based on when the controllable node can get into a position to be of use.

When a controllable node is instructed to move, the NCC will instruct the manager not to make the same request again until the controllable node has completed its journey.

It is envisaged that controllable nodes as a resource would be quite limited; therefore, there could be many requests from manager nodes for the use of this resource at a given time. When this is the case the NCC will allocate controllable nodes to managers in order. The order is determined by the source node that is performing worst.

1.7 Implementation

The above was implemented using C++ and the discrete network event simulator “Network Simulator 3 (NS3)”. The novel implementation consists of over 200 methods (see appendix 2), each created entirely by us. That is, they are not part of the NS3 libraries of classes and did not exist prior to them being creating for this research. In addition to this the default configuration did not allow the latest developments in the C++ programming environment to be used, thus, NS3 had to be reconfigured in order to use the latest compilers that were available. This enabled rich features such as dynamic vector data structures which were used for recording various information about network state, memory, prediction and for the cognition process.

A simple ad-hoc wireless network that is to be modelled in NS3 can be coded is approximately 200 lines of code, with the use of very few methods. However, the complexity introduced with the implementation of the cognitive attributes resulted in approximately 10,000 lines of code.

1.8 Results

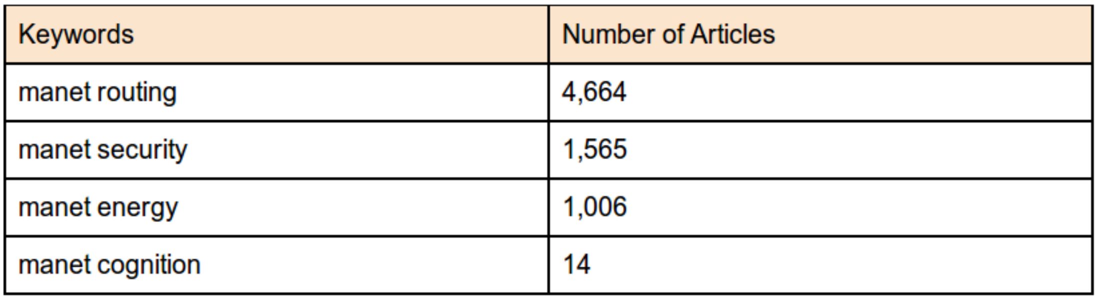
The results derived from the simulation runs show that the implementation of cognitive attributes using an OSI cross-layered approach does increase network performance. In some models the increase in performance was not as great as in other models. This could be attributed to less traffic flows (i.e. some models had a single traffic flow whilst others had four traffic flows), less participating nodes (some models ran with 10 nodes and others with 80 nodes) or the mobility model being used or other parameters of the network model design (density, speed, transmission range and so forth). However, all models performed better than that of the models that used a given established routing protocol with no cognitive features applied.

The use of a resource such as controllable node should increase network performance due to the fact that more nodes participating create more paths within the network. Therefore simulations were conducted with cognitive features that did not make use of the controllable nodes (partial cognition), in order to compare against the results of those with full cognition (with controllable nodes) applied. The results were surprisingly similar between partial cognition and full cognition in some cases.

The analysis of the data produced by the simulation runs shows an increase in network performance when cognitive attributes are applied.

1.9 Novelty

As far as aware this work is unique and novel. In the papers that the author is aware of, the research in MANETs focuses on routing protocol enhancement, security or energy. To illustrate this point see Table 1- Number of Articles related to Keywords, which was created based on performing a search of the IEEE Xplore Digital Library[1] using specific keywords and then recording the number of relevant articles. The date range used is from the first available article until December 2014, for example the table shows that the keywords “manet” and “routing" returned 4,664 articles, the first publication in 1998 (Lusheng & Corson, 1998).



Keywords	Number of Articles
manet routing	4,664
manet security	1,565
manet energy	1,006
manet cognition	14

Table 1- Number of Articles related to Keywords

The table does not show that of the 14 articles returned by the keywords “manet” and “cognition”, 50% are less than 5 years old, with the earliest dating from 2006. This shows that the research is novel and half of this research is less than a decade old. In addition to this and previously mentioned, most of the required methods for this project simply did not exist and were coded entirely for this thesis (see Appendix 1 and 2).

[1] The IEEE Xplore Digital Library offers instant online access to more than 3,928,685 articles which consist of peer reviewed conference publications, journal articles and e-books.

1.10 Thesis Summary

The initial work deliberately does not focus on Mobile Ad-Hoc Networks (MANETs) or Cognitive Networks (CNs) but discusses the issues that are associated with wireless and wired communication networks in general. This allows the differentiation between conventional network design and ad-hoc network design.

This thesis is structured in the following way:

Chapter 1. This chapter is introductory. Discussion and formulation of the main aims and objectives of this thesis is given; the research goals are set and discussion of the plans how to achieve them. Basic cognitive attributes required for the cognitive network are also discussed. This chapter also includes major results obtained in this thesis together with discussion of the novelty of these results. It concludes with the structure of this thesis.

Chapter 2. This chapter begins with a discussion on Wired and Wireless Networks' background which is essential for understanding before looking at MANETs. Then the discussion moves onto MANETs appearing to be a special case of a Wireless data communication Network. Then discussion of a MANET's major applications and structural elements are presented. Following this is a discussion of a Cognitive Network giving the objectives, background and the applications for a cognitive network.

Chapter 3. This chapter looks at the major and most recent research in the area of MANET's implementation and evaluation. Some of the other research papers on the related areas, necessary for the research in this thesis, are analysed. Together with chapter 2, chapter 3 completes Objectives 1 and 2 set above.

Chapter 4. This chapter discusses the problems associated with MANETs. Issues that disrupt the performance of the network such as: congestion, spectrum allocation and routing are focused upon. Detail is given on how to address the problems identified by using simulation software. Further detail of how the simulation software was used to address the issues identified is presented.

Chapter 5. This chapter explains a conceptual design and the discussion is intended to present the reader a design that shows how the problems identified in chapter 4 could be addressed. Particular attention is paid to the topology, cognition, cognitive attributes, memory, information capture, information processing and how it is intended to deal with multiple contending managers. This chapter concludes by detailing how this work is novel.

Chapter 6. This chapter discusses the implementation of the non-transmission side of a MANET, namely the nodes, the initial placement of the nodes and the mobility models. Towards the end of the chapter discussion of an implemented prediction scheme that allowed the prediction of the position of nodes at some time in the future is detailed.

Chapter 7. When considering the nodes particular focus was paid to the hardware associated with the node and discussion of the configuration at the physical layer, the media access control, the transmission radius, the protocol stack, addressing and interfacing, routing protocols and channels. Detail is also given on the configuration of traffic flows and how some nodes are configured to cause congestion. How performance of the network is measured by using packets is discussed. In this chapter energy models that are used and the logging components that have been implemented is discussed.

Chapter 8. This chapter details how cognition was applied to the MANET. The reader is reminded of the admissible cognitive actions, explanation of the implemented node election method, how the network state is obtained and used, the role of the management node and how this was implemented. Furthermore memory and attentiveness is discussed.

Chapter 9. This chapter contains representation, evaluation and analysis of the results of simulations. It meets the rest of objectives set above.

Chapter 10. This chapter contains the future work and plans that this research has led to.

Thesis concludes with References, Bibliography and two Appendices.

Appendix 1. The full C++/NS3 testbed code for the cognitive MANET is presented in Appendix 1.

Appendix 2. The full list of methods and structures used in the code is presented in Appendix 2. The methods are listed there in the same flow order they appear in the code.

Chapter 2 Background

This chapter discusses the background and some history of data communication networks. The discussion begins by looking at Traditional Wired Networks (Section 2.1) from their first application and the type of traffic that this network infrastructure supported and how this traffic has evolved as a result of new networking applications. Wireless data communication networks are discussed next (Section 2.2), in this section the characteristics of a Wireless Network such as: bandwidth, signal generation, modulation and multiplexing is discussed. The IEEE standards for wireless communication are also discussed. Section 2.3 details mobile ad-hoc networks and the objectives, background and applications for this network are discussed. The next section discusses Cognitive Network and is structured similar to Section 2.3, therefore, the objectives of a cognitive network, the background and the application of this network are discussed. The chapter concludes with a chapter summary before moving to Chapter 3 which reviews current literature.

2.1 Traditional Wired Networks

Traditionally computer networks served few applications, examples include: file transfer (FTP), terminal over network (TELNET) and electronic mail (SMTP). The traffic these applications placed on the network was relatively small, for example, TELNET is a text based interface and transferred short text commands. Email traditionally was text based and produced small text files to be transmitted over the network. When considering FTP, files where relatively small (the initial file allocation table (FAT) system supported a maximum of 16MB volume sizes). Computer networks grew in popularity and began to be adopted for educational purposes with many Universities being the first non-military application of a communications network. In 1989 the development of the World Wide Web further accelerated the pace, this introduced new applications, typically Web Browsers that used the hyper-text transfer protocol (HTTP), which transferred text documents that made use of mark-up tags such as <b> to be interpreted by the browser, the mark-up tag allowed the browser to display the text with formatting applied for example the <b> tag indicated the following text should be embolden, this still constituted as relatively small text based files but can be said to have been the catalyst for today's network traffic. Traditional text based traffic generated by traditional applications began to change: the world-wide-web saw the introduction of images, new file systems (FAT32) that supported bigger files and volume size (followed by NTFS or ext2 and ext3 if using Linux or equivalent for other operating systems) were implemented, faster more powerful end systems with greater clock speeds and increased memory all played a part in the increase in traffic, however, the new and innovative uses of the world-wide-web such as the introduction of multimedia content played a key part. Network infrastructure that traditionally supported the transfer of text based files and relatively small file sizes began to transfer larger and larger files as multimedia content became more accessible, in particular sound and video files which may be hundreds or thousands of megabytes in size, in comparison to a 3kb text file. Nevertheless, the traditional communications networks facilitated new applications and new volumes of traffic, albeit with some delay, the service or operation of the application was unaffected.

Essentially, a communications network primarily facilitates the propagation of electromagnetic signals which are the representation of data, data can be either analogue or digital and the type of data be it voice, text, image or video are treated the same.

This has now changed, research has shown that the explosion of multimedia applications have altered the trends in network traffic, (Stallings, 2009). New innovative multimedia applications and services are pushing the communication networks further with the introduction of applications that require a certain quality of service (QoS), examples include streaming audio, streaming video and VoIP (voice over IP), all of which cannot tolerate delay. Stallings (2009) has broadly classified two types of network traffic: 'Elastic' and 'Inelastic'. Elastic traffic is that which can adapt to changes in the network and is generated by applications that do not require a QoS. Inelastic traffic is that which cannot adapt to changes in the network and must maintain a minimum throughput in order to provide the QoS required by the application, streaming audio, streaming video and VoIP are typical examples of applications that generate inelastic traffic.

Networking technologies today are an integrated part of life and over the last decade networking communications technology has made its way into mainstream society, long gone are the days where networks were only found in corporate companies consisting of mainframes and terminals connected via a guided medium. Today networking communications technologies consists of an infrastructure supporting hundreds or thousands of clients, or in case of the internet 1,668,870,408 clients, which is a growth of 362% since the year 2000 (Miniwatts, 2009). With the huge growth of users on the Internet and the new type of media available online for example streaming video, more and more data is traversing the Internet backbones and being passing though the routers that connect the high speed trunks.

2.1.1 Evolution of the traditional wired Network

Today, the infrastructure that supports the communication of data continues to meet the demands of the applications, but this is only true because several advances have been made in which way the infrastructure is used. Many limitations of this infrastructure have been overcome by applying novel techniques in how data is transmitted, consider that the characteristics of a communications medium are that electromagnetic signals which represent data that propagate through the medium, or in the case of optical fibre light omitted by a laser. Only one electromagnetic signal or light may traverse the medium at any given instance, this would result in long delays and a huge cost in regards to an under-utilised resource, to circumvent this problem, data compression and a form of multiplexing (for example time division or frequency division) is used to combine several signals into a single signal for transmission (Stallings, 2009).

Another evolution of a traditional wired infrastructure is the ability to upgrade key devices in strategic locations, for example: there are many classes or segments of networking technologies and most of these technologies are connected to high speed communications trunks that are often comprised of optical fibre or co-axial cable, because the infrastructure is in place and is fixed, old slower switching or routing equipment can be replaced with newer and quicker equipment at strategic locations within the network.

In regards to QoS it is inherently difficult to guarantee a QoS because of the nature of packet switching networks[2]; however, with the use of virtual circuits and priority levels applied to packet headers QoS using a best effort algorithm can be achieved (Al-Soufy and Abbas, 2010).

The unprecedented growth that shows no sign of slowing have given concern to how long the Internet and the communications infrastructure will operate as it does today. The Cabinet Office's official advice, Preparing your Business for the Games, says that the country's telecoms system may be unable to cope with demand to access the Internet in certain areas (London Organising Committee, 2012).

[2] A communications network that groups all transmitted data, irrespective of content, type, or structure into suitably sized blocks, called packets

Some believe that the Internet will not stop working, but there would be many applications that would not work online (Johnson, 2010). If a predicted growth of at least 100% occurred the technology and the way in which communication occurs would need to be changed to keep pace with this growth (Linkins, 2008).

To summarise this section it can be said that traditional data communication networks are reactive, that is they react to tackle problems in the network after the problem has occurred. However, the evolution of hardware and the use of techniques to manipulate data have allowed the network infrastructure to continue to operate and to meet the demands of its many applications... but how long for?

The new type of applications and new type of services are driving the research of new and innovative communication network designs, one of which is coined 'Cognitive Networks'. Networks must be able to adapt to the traffic patterns created by the applications, in particular when delay cannot be tolerated.

2.2 Wireless Networks

One of the most popular and simplest networks to implement is the wireless network that may be found at home to connect a laptop wirelessly to a wireless router for Internet access provided by an ISP (Internet Service Provider), or the Wi-Fi hotspots increasingly found in airports and other locations where the public may congress such as: coffee shops, bars, libraries, schools, colleges or universities but to name a few of hundreds of examples. This technology is simple to implement because the wireless technology is connected to the wider area network (traditional wired network described earlier) using one hop. That is the wireless router or access point is connected to a wired infrastructure which then uses the conventional means of communication technologies as described in the last section.

Wireless antennas are used to transmit radio waves that are the representation of data, the antennas can be categorised as being omni-directional or directional. Wireless antennas that are omni-directional propagate radio waves in all directions equally[3] directional antennas are essentially a focused omni-directional antenna that has gain applied to produce greater signal strength in a particular direction at the expense of other directions. FM (Frequency Modulation) or AM (Amplitude Modulation) radio is an example of a technology that uses omni-directional antennas (as is a Wi-Fi network). Satellite is an example of a directional antenna and consists of a point-to-point directionally focused link using a parabolic dish for the transmitter and/or receiver.

Infrared is another example of wireless technology with the exception that a beam of light is used to represent the data and sensors are used rather than antennas to receive the light. This research focuses on wireless technology that uses Omni-directional antennas without antenna gain applied; this is typical of most wireless equipment that may participate in a MANET.

The very nature of omni-directional wireless networks is that they are a broadcast system; this in itself offers problems with how the network operates and how the available bandwidth of the spectrum is utilised. Wireless networks operate on a very restrictive bandwidth allocation scheme and use a restrictive number of channels, many of which overlap. The protocol in use determines the frequency of operation and how this frequency range is allocated into channels. This research discusses some of the most popular protocol implementations that form part of IEEE (Institute of Electrical and Electronics Engineers)

[3] An actual physical antenna will radiate more energy in some directions than in others because it does not create energy but radiates power. In some directions more energy is radiated at the expense of other directions.

802.11 group, prior to this short explanations of bandwidth, signal generation and modulation techniques are given which will give a greater understanding of the technologies used by each of the IEEE standards discussed.

2.2.1 Bandwidth, Signal Generation, Modulation and Multiplexing

Bandwidth is an ambiguous term because it has several meanings across similar disciplines for example; bandwidth in relation to digital computing is the rate at which data can be transferred, for example along a BUS, this is measured as the throughput or bit rate and is measured in (bps) bits per second (Dictionary.com, 2015). This is not the same as bandwidth associated with signal processing which is measured in (Hz) hertz. Bandwidth can be defined as the number of Hz between the lowest possible frequency and the highest possible frequency, and although there is relationship between data rate and bandwidth, it should not be confused with the digital computing term that refers to throughput. Often in radio communications, the higher the bandwidth the higher the data rate, however, there are many factors that decrease the data rate, for example, interference, thus reducing the throughput. To avoid the ambiguity for the remainder of this thesis each time the term bandwidth is used it is in relation to signal processing and will have a metric of Hz.

The nature of wireless communication is the encoding of data, either analogue or digital data can be encoded into electromagnetic signals for the transmission onto a medium, either guided or unguided, in wireless communications the medium is normally unguided and transmitted in the form of radio waves. Figure 2 - Wireless Transmission depicts the process of digital data being encoded to produce an analogue signal; this is a common technique used in wireless communications and is applicable encoding for a MANET.

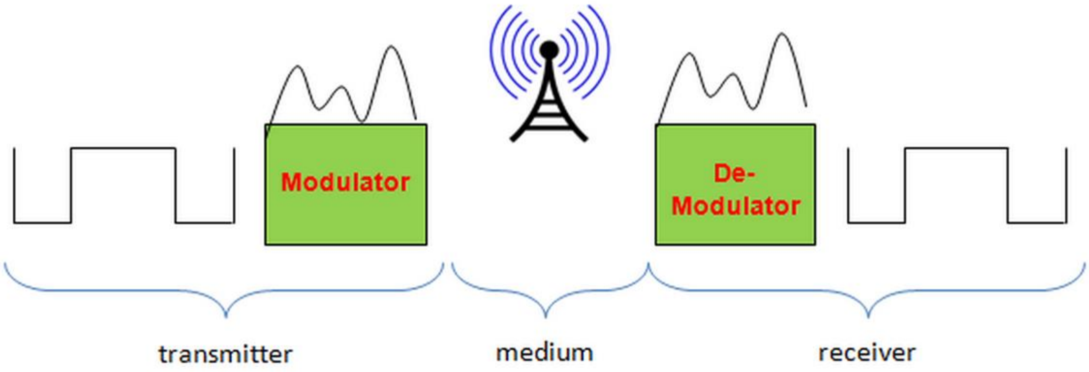


Figure 2 - Wireless Transmission

What the figure does not depict are the modulation techniques that can be used for the conversion of digital data to an analogue electromagnetic signal for transmission across an unguided medium, traditional wireless modulation techniques consist of FSK (frequency shift keying), ASK (amplitude shift keying) and PSK (phase shift keying). In order to understand how PSK, ASK or FSK modulation techniques work it is necessary to briefly discuss the components that constitutes a signal. Figure 3 - Sine Wave shows a sine wave, this depicts the three basic components of any signal. A sine wave is a waveform that oscillates periodically, and is defined by the function y=sin x, the diagram below shows the first cycle of the sine wave and would be repeated for the duration of the radio wave to be transmitted. It is the manipulation of this sine wave that allow PSK, ASK and FSK modulation techniques.

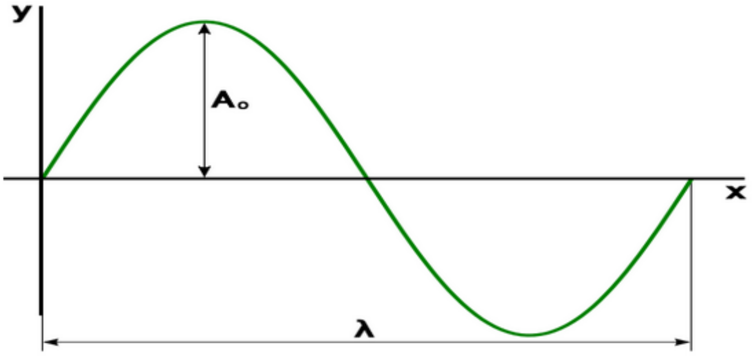


Figure 3 - Sine Wave

Axis y in Figure 3 - Sine Wave measures the strength of the signal and is normally a level of voltage, the voltage is normally measured in amplitude, and the highest voltage level is known as the peak amplitude and is represented in the diagram by the top of the arrow associated with A_0. Axis x can be thought of as time axis, each signal is a function of time. The number of times that the signal repeats within a given time period is the frequency of the signal and is a measurement of Hz. The frequency of a signal has a direct relationship with the ‘wavelength’ of a radio signal, and in this case the wavelength is represented by λ. Essentially, the greater the frequency or wavelength is, the greater amount of data that can be transmitted.

The last component that constitutes a signal is phase, phase can be described as the relative point at a given time interval.

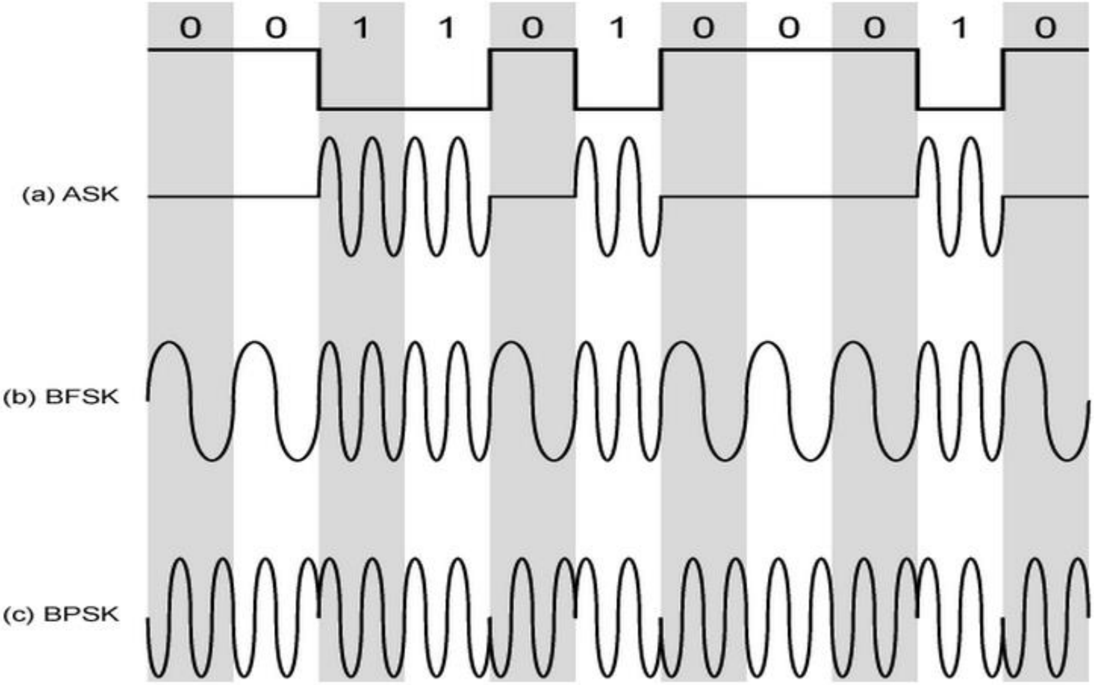


Figure 4 - Modulation Techniques (Stallings, 2009)

The modulation of data is the manipulation of a sine wave in order to 'overlay' the data onto the sine wave. Figure 4 - Modulation Techniques (Stallings, 2009) shows each of the three modulation techniques, the top of the image depicts the binary bits of digital data that is to be encoded. The diagram shows ASK, FSK and PSK modulation techniques and are labelled: (a) ASK, (b) BFSK and (c) BPSK, the B before FSK and PSK is binary. In (a) ASK each binary 1 is represented by a higher amplitude than that of a binary 0, in this case binary 0 is represented by no current. In (b) BFSK, a binary 1 is represented by a higher frequency than a binary 0. In (c) BPSK, a binary bits are represented by a shift in phase, for example when the first binary bit 1 is encountered there is a phase shift which inverts the signal, the shift does not happen for each binary bit 1, but for each alternating bit encountered, thus, because the phase was shifted at binary bit 1 the next phase shift will not occur until a binary bit 0 is encountered. There are techniques whereby several frequencies or phases can be used to represent more than one bit. An example of such a technique is Multiple FSK whereby two frequencies are used, in this case each signal element represents more than one bit. When considering PSK techniques different phases can represent more than one bit, for example, quadrature phase shift keying (QPSK) represents two bits at shifts of 90^0. PSK and QPSK are depicted in Figure 5 – PSK and Figure 6 – QPSK. Where PSK represents a binary 1 or binary 0 using shifts of 0^0 and 180^0, QPSK uses an additional two phases and thus can represent two binary bits per shift. Therefore QPSK is able to modulate double the amount of data per single signal element.

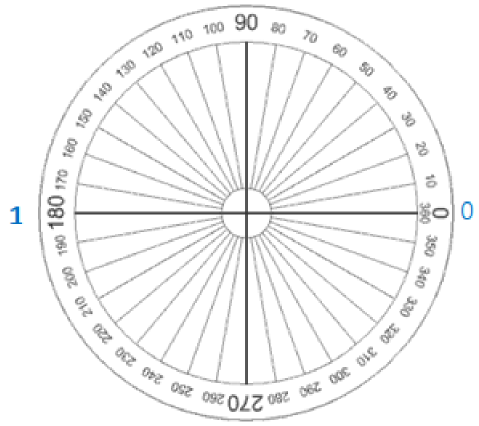


Figure 5 – PSK

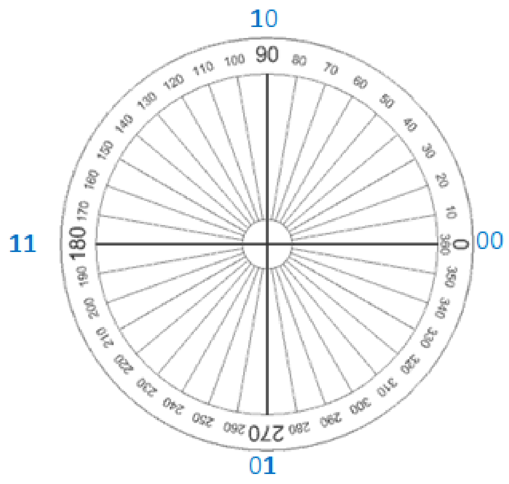


Figure 6 – QPSK

If the quality of the link is good then 16-QAM (Quadrature amplitude modulation) or 64-QAM phase shifts could be used, as the quality of the link degrades, so does the modulation scheme being used. 16-QAM represents 16 different phase shift that enable four bits per signal element. When considering 64-QAM, 64 different phase shifts are used which allows for the transmission of 6 binary bits per signal element.

This section has given definitions of ASK, PSK, FSK modulation techniques, this then expanded to detail how each of the techniques could modulate multiple bits per signal element for example using QAM and QPSK. It then detailed further how after using techniques to increase the number of bits per signal element how this could then be multiplexed to make better use of the limited radio spectrum, two media access techniques were discussed Direct Sequence Spread Spectrum (DSSS) and Frequency Hopping Spread Spectrum (FHSS), this section was necessary as they are used in the various wireless protocol implementations which have become IEEE standards, this thesis will focus discussions on the IEEE 802.11 which is the standard for wireless communications.

2.2.2 IEEE 802.11

The organisation for setting the standards in wireless communications is the IEEE association, more accurately the organisation serves standards for many research areas including: electrical, electronic, computing fields and other areas that relate to science and technology. The IEEE organisation is very well established in many of the research areas associated with the above fields.

One such area is the IEEE 802 area, which primarily dealt with Local Area Networks and Metropolitan Area Networks, such as the well-established Ethernet Local Area Network (802.3). As wireless network were introduced standards were adopted and assigned the standard 802.11 (11 being the next available number).

The 802.11 family of standards defines wireless communication in the 2.4GHz and 5GHz range and is primarily concerned with standards for the data link layer for media access and the physical layer for issues such as encoding. Although 802.11 defines all wireless communication including IR (Infrared) and RF (Radio Frequency) this thesis focuses on the RF wireless standard and details the advances and defines the characteristics of these standards, in particular on recent adaptations that are pertinent to the research area of MANET communication using the Wi-Fi (Wireless-Fidelity) standards.

The first standard adopted in 1997 initially offered a theoretical bandwidth of 2Mbps and used CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance). The modulation scheme is based on DSSS and FHSS. This adoption was not widely used and although much research was undertaken did not result in popular application.

2.2.3 802.11a-1999

In 1999 the IEEE 802.11a standard was ratified and was based on the original implementation, this was later reaffirmed in 2003. Multiplexing techniques are used in 802.11a in order to combine many data bits into a single symbol; the technique used is OFDM (orthogonal frequency-division multiplexing). The symbol is then modulated using 64-QAM, 16-QAM, QPSK or BPSK depending on the data-rate in Mbps.

The combined use of OFDM and 64-QAM offers data rates up to 54Mbps in ideal conditions. This implementation operates in the 5GHz frequency range, this range is subdivided into 52 subcarriers each occupying 20Mhz, 48 carriers are used for user data and 4 for control data (Patidar and Vijaykumar, 2007).

2.2.4 802.11b-1999

Also in 1999 was the ratification of the IEEE 802.11b standard. The multiplexing techniques used in this standard are the DSSS (direct-sequence spread spectrum scheme). The symbol is then modulated onto the carrier using the CCK (Complementary Code Keying) modulation scheme, which is a QPSK pair when using a chip rate of 11Mchip/s. The data-rates offered by the 802.11b protocol consisted of 5.5Mbps or 11Mbps depending on the number of bits (4 or 8 respectively) modulated onto the eight chips that form the single signal element. This implementation operates in the 2.4GHz section of the spectrum, the range is subdivided into 14 channels[4], and each of the channels are 5MHz apart with the exception of channel 14 which is 12 MHz, each channel is 22Mhz including channel 14. Control data is passed with user data on the same channel.

The allocation of channels is better illustrated in Figure 7 - Channels assignment , where it can be seen that channel 6 for example, is operating at a frequency between 2426MHz and 2440MHz. It can also be seen that in order for channels not to overlap channels 1, 6 and either 11, 12 or 13 must be used, giving only 3 non-overlapping channels.

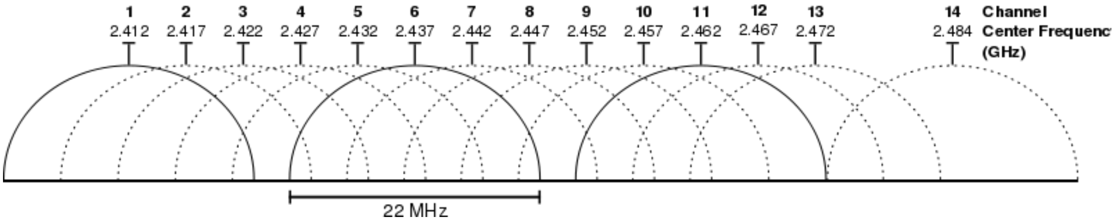


Figure 7 - Channels assignment (Grigorik, 2013)

2.2.5 802.11g-2003

IEEE 802.11g uses techniques from the implementation of the 802.11a protocol and the 802.11b protocol. The bits are multiplexed using the techniques found in 802.11a, which uses OFDM and then modulated onto the carrier using DBPSK (Differential Binary Phase Shift Keying), DQPSK (Differential Phase Shift Keying) and DSSS, depending on the link quality, the data rates that can be achieved are: 6, 9, 12, 18, 24, 36, 48, and 54 Mbit/s.

The protocols 802.11a and 802.11b cannot work together in the same network because of the different frequencies that each protocol operate on, however, the 802.11g protocol uses the same 2.4GHz band with the same channel allocation as 802.11b making the implementation of the 802.11g protocol backward compatible with legacy 802.11b devices, however, the symbol is modulated onto the carrier using the CCK modulation scheme,

[4] Channel 14 is only available for use in Japan. Channels 1-11 are available in the USA. Channels 1-13 are available in Israel but channels 1-4 are only allowed to be used for indoor application. Channels 1-13 are used in Europe and most other parts of the world. (IEEE, 2007) and (Autorité de régulation, 2003).

which is a QPSK pair when using a chip rate of 11Mchip/s as in the 802.11b implementation thus reducing the speed to 5.5 and 11Mbit/s.

Even though 802.11g operates in the same frequency band as 802.11b, it can achieve higher data rates because of its heritage to 802.11a. The application of 802.11g was quickly adopted partly due to the higher speeds offered over 802.11b but also because 802.11g is able to support legacy 802.11b devices, the adoption was before ratification of the IEEE standard in 2003.

2.2.6 802.11n-2009

One of the latest advancements of IEEE 802.11 implementation is IEEE 802.11n; this standard has a hugely increased data-rate in comparison to the previous protocols discussed at a maximum rate of 600Mbit/s. There are additional features and improvements over the other implementations such as security, error-correction and frame aggregation. The key feature of IEEE 802.11n is the support for MIMO (multiple-input multiple-output) by increasing the number of antennas being used; this is possible by using spatial multiplexing. Also the ability to increase the size of the channel operating on, the IEEE 802.11b and IEEE 802.11g operate using 22Mhz channels which are 5Mhz apart, the IEEE 802.11n protocol will allow channels of 40Mhz essentially doubling the available bandwidth. In addition to this the 802.11n protocol will operate in either the 5GHz band or the 2.4GHz band which allows for compatibility with both the IEEE 802.11a and IEEE 802.11b protocols.

Depending on link quality various modulation types can be used, however, nothing that has not already been discussed (BPSK, QPSK, 16-QAM, BPSK, QPSK and 64-QAM). In order to achieve the data-rate of 600Mbit/s there are 4 antennas used with 64-QAM’s modulation operating at a coding rate of 5/6, the antennas are operating using 40 MHz channels with a guard interval of 400 nanoseconds. Although this is theoretically possible, it is not usual because the 2.4GHz band is congested and other competing radio devices will probably interfere thus reducing the data rate, the 5GHz band could be used but most legacy devices operate in the 2.4GHz band.

The next implementation may be the IEEE 802.11ac and claims to be able to offer speeds of up to 138672Mbit/s using 8 antennas and 160MHz channels in the 5GHz range; this is detailed further in the 802.11ac Technology Introduction (Rohde and Schwarz 2012).

This illustrates that wireless communication is operating using a very restrictive frequency bandwidth and that the channel number that is assigned to the wireless transceiver determines the actual frequency range within the 2.4GHz range that the wireless device is using. Current restrictions of bandwidth and channel assignment mean that the maximum data rate that can be achieved is up to 54Mbps when using IEEE 802.11g.

Other examples operating in the 2.4GHz range consists of Satellite, Bluetooth and Infrared, which can make the 2.4GHz range congested, cause interference and thus reduce data-rate. There are moves to better utilise the 5GHz band, however, the range of frequencies that can currently be communicated on is a limited finite resource and as there is more application in wireless technology the 5GHz band will likely become as congested as the 2.4GHz band. In addition to this research has been conducted into using other bands that are restricted and licensed as secondary users, for example using the range that is licensed to the emergency services, this research is discussed further in the literature review chapter 3.

The next section will introduce a MANET, which is essentially a wireless network without infrastructure, currently the discussion has given an overview of how wireless networks operate and the techniques used, these techniques are also used in MANETs, however because of the lack of infrastructure and because the devices are not always last hop there are many additional factors that could affect the performance of the network.

Wireless networks can use other techniques and be adopted to work with a MANET; these are discussed in more detail in Media Access Control Techniques for Wireless Networks section later in this thesis.

2.3 Mobile Ad-Hoc Networks

The chapter will also discuss the overall objective of a MANET and continue by detailing and discussing the stages of research that has been conducted in this research field starting from the early 1970's when the concept of a MANET was first conceived to the recent advances. The chapter concludes with a study of the innovative applications that this type of communications network could be applied to.

In the earlier section wireless networks were introduced and the operation detailed. Those types of wireless networks are typically final hop, for example the wireless router/access point is connected to an infrastructure and the wireless communication is between the router and the end wireless device. Multiple hop routing is performed using dedicated routers that make up the infrastructure and are typically not wireless. Another important wireless network is a network that does not connect to a conventional wired infrastructure, or does so after using collaborating nodes within the network to forward data to a node that is connected to the wired infrastructure. This type of network may be a mobile network and is abbreviated to MANET (mobile ad-hoc wireless network). This network can be considered a peer-to-peer network and no centralised server is normally found in this configuration of network. The network has a topology of a mesh type network whereby each connecting device on the network automatically powers up and connects to other devices within the wireless range of that connecting device, however, it should be noted that the topology of the network is very difficult to predict because the nodes are mobile and may move into or out of wireless range.

A MANET can be defined as a network with no predefined infrastructure that supports end-to-end communications by all nodes within the network participating in the forwarding of packets; the nodes are usually mobile and are therefore often powered by battery.

Myoupo et al (2009) and Ousmane et al (2007) give their own definitions of a Mobile Ad-Hoc Network (MANET) and describe it as a network of self-configuring nodes. The nodes are free to join the network at any given moment with no prior warning. The nodes can be highly mobile which makes this type of network very dynamic in nature. There is no defined topology for this type of network as there is no fixed infrastructure. A node participating in this network will connect with all other nodes within its transmission range, this gives the visual representation of a mesh type network, however, unlike mesh wired networks the nodes are less likely to maintain their connection with the network. The volatility of connections is caused by elements associated with wireless communications that are not normally associated with networks of a fixed infrastructure. Some of these limitations are attributed to mobility, power and transmission (such as range, strength) and are discussed in more detail in chapter 4.

2.3.1 Objectives of a Mobile Ad-Hoc Network

The objective of a MANET is to establish a self-configuring network of mobile nodes and associated hosts that are connected by a series of wireless links. The nodes should also act as routers for the forwarding of packets intended for another destination. The nodes should be free to move from one location to another and reinitialise themselves based on the individual programming of that node; for example Basu et al (1999) writes about power-up-n-play, whereby devices are installed and, when powered up, the devices automatically configure themselves and connect to other devices in an environment where there is no predefined infrastructure.

When a source node wishes to establish a route to a destination node it initiates a route request packet that propagates and traverses the network. When the route request packet arrives at the destination or finds a route to its destination via other nodes a route reply packet is sent to the source node. Unlike conventional wired networks each node in a MANET effectively becomes a router. The simplest mobile network would consist of two nodes that are able to communicate directly with each other as in Figure 8 - Point-to-Point connection between two nodes; however, MANETs have to support an indirect connection.

Figure 8 - Point-to-Point connection between two nodes

The indirect connection is achieved by intermediate nodes participating in the routing of the packet by forwarding the packet to the destination node or to a node that is next in a series of intermediate nodes to the destination node as illustrated in Figure 9 - In-Direct Connection between two nodes.

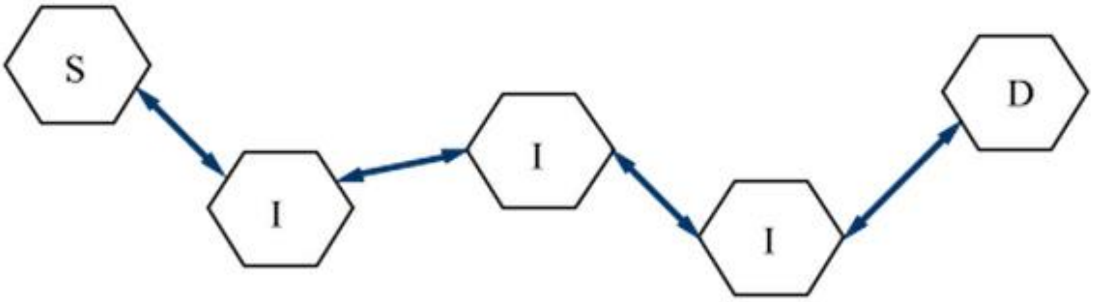


Figure 9 - In-Direct Connection between two nodes

Transferring a packet from one node to the next is called hopping and the number of nodes a packet has passed through is how many hops it has taken for the packet to arrive at its destination. The hop count of a packet is the number of nodes that have forwarded the packet plus one, for example the hop count in Figure 9 - In-Direct Connection between two nodes would be four. The hop count is usually one of the important factors when determining a route from a source node to a destination node via intermediate nodes. Another consideration when determining the route the packet should take is determined by the cost of that route. Several other factors are taken into consideration when determining the cost of a route such as upstream and downstream metrics that are used with the TORA protocol. The cost of establishing a route will be discussed in the chapter 4.

There are two general methods of route discovery for the mechanism of transferring a data packet from source node to destination node via intermediate node(s) and this generalisation can be classed as either reactive or proactive protocols.

Many of the proposed protocols for Mobile Ad-hoc Networks perform a cascade-based route discovery, whereby a route request packet is cascaded across the network. The search is known as a broadcast and is normally omni-directional. The packets' number increases since each node (other than the initial activated node), that does not know the location of the destination node, will cascade the route request packet to all other nodes within its transmitting range. This method of route discovery causes an abundant amount of network traffic which is very costly in terms of power, bandwidth, latency, packet delivery ratio and jitter (small rapid variations in a waveform resulting from fluctuations) of data packets. Jian and Mohapatra (2003, pp. 692- 696) explain furthermore:

"When a node S has some data to send to node D but has no existing route to the destination, it will initiate a route discovery process by broadcasting a route-request packet. An intermediate node I, upon receiving the route request packet for the first time, will rebroadcast the route request again if it does not know a route to the destination node D. Finally, when the route-request packet reaches a node (which may be the destination node D itself) that has 2 a route to node D, a route-reply packet is sent back to the sender node S."

Techniques of route discovery will be examined in greater detail in chapter 4. This will discuss specific algorithms used by specific protocols that have been designed or adapted to function in a MANET.

Many of the nodes in a mobile network are portable; such a device might be a mobile phone or a laptop. Nodes in a mobile ad hoc network are effectively acting as: servers, clients and routers; because of the nature of the device that may be used in a MANET the nodes may randomly move location, hence, making a predefined infrastructure impossible. The network's topology is therefore dynamic and as such its "shape" and thus its operation characteristics are unpredictable. Brayer (2004, pp. 1183-1196) states that:

"The routers are free to move randomly and organize themselves arbitrarily; thus, the network's topology may change rapidly and unpredictably. Such a network may operate standalone, or may be connected to the larger Internet. These networks do not require any prior investment in fixed infrastructure. Instead, the network nodes agree to relay each other's packets toward their ultimate destinations, and the nodes automatically form their own cooperative infrastructure."

Also as stated by Brayer (2004) is that the network should operate in a standalone fashion and be able to connect to a larger internetwork when required.

An area of research in MANETs is in relation to the routing of packets using intermediate nodes for multiple hop destinations, or more precisely the establishment of a viable and efficient (least cost) route for the packets. The difficulty lies in finding a series of nodes that can deliver the packet from the source node to the destination node. Although this has been accomplished to a certain degree, depending on the size, scale, density and the number of participating nodes within a geographical area the control overhead traffic could create such a burden on the network that actual data packets do not have the opportunity to be transmitted because the burden placed on the network is of such a level that the viability of the network in regards to end-to-end delay and general performance is so poor (Wu and Harms, 2002). This is further impacted on by the fact that the nodes can at any time move or 'fall off the network' (to stop communicating with the network) this could be due to a number of factors; an example may be due to power failure of the node or the node moves out of transmitting range of all other nodes (this is more common for nodes at the edge or border of the network). The nodes are also susceptible to a higher ratio of packet losses over that of a conventional wired network due to the characteristics associated with that of wireless communication networks. The current established protocols do not take into consideration the factors that could have caused the loss of the packet and assume that congestion is to blame and therefore acts accordingly to deal with the congestion; however, in a MANET failures other than congestion may account for packet loss.

2.3.2 Background of Mobile Ad-Hoc Networks

This section of the introduction to MANETs will look at the background and developments over the years in relation to MANETs, starting from the early 1970's when the idea of a MANET was first conceived to the research that has been conducted over the years and the current advancements of a MANET.

According to Cleveland (1993) in the early 1970's a project began at the University of Hawaii called the Area Locations of Hazardous Atmospheres (ALOHA) project, the project was started and led by Abramson. The primary focus of the project was to test and provide different networking capabilities in a combat environment. The ALOHA project used a packet radio to transfer data in a single hop network and according to Bakht (2005) used Carrier Sense Medium Access (CSMA) and a variation of a distance vector routing protocol. This project led to the work of the development of a multihop multiple access Packet Radio NETwork (PRNET) and was sponsored by the Advance Research Projects Agency (ARPA) which would later become the Defence Advanced Research Projects Agency (DARPA).

Project PRNET was first proposed by SRI International, SRI International was the first organisation to connect two networks with different infrastructures and topologies using a transmission control protocol. Website www.sri.com states that in 1976 SRI established the first connection between heterogeneous networks and communicated using the

Transmission Control Protocol (TCP). This enabled connectivity between a wired and wireless network.

PRNET used a form of data transmission used in radio to develop a wireless computer network. The radio used packet switching technology whereby a continuous data stream or a group of bits were transmitted in a packet. The grouping of the bits is what makes the radio a packet switching network and is often characterised by pulsating transmissions, this is important as it allowed multiple indirect connections between two devices that acted as though it was a direct connection even though multiple hops would of occurred.

The packet radio technology predates the Internet but was part of the motivation for the production of the Internet Protocol (IP) suite.

The early experimental trials that tested the packet radio network consisted of many scattered transmitting and receiving stations known as nodes, each node also acted as a store and forward station for routing the packets, the packets that carry the data as well as the control information are transmitted over the wireless packet radio network. Furthermore a packet may be produced at any node and can be forwarded to any node on the network in a series of hops.

Packet Radio Network experimentation started in 1977 with funding from DARPA. Most communication took place between mobile vans. In the original paper produced by (Kunzelman, 1978) reports on the experimentation and progress that took place on packet radio communications experiments for the period 1st August 1977 through to 31st October 1977.

Kunzelman (1978) “An expected part of the ARPA work was to demonstrate progress and give evidence of this networking capability. So TCP, spanning the PRNET and the ARPANET would be demonstrated in May 1977 between the SRI van and hosts at ISIC and SRIKL. On August 11, 1977, a TELNET as demonstrated between the van and the Naval Ocean Systems Centre in San Diego for Admiral Stanfield Turner (Dir. CIA) and William Perry (DDR&E). On September 19, 1977, a single LSI-11 microcomputer, running a multi-connection TCP, multiplexed four terminals through a packet radio to four different ARPANET hosts, essentially all of the ones running TCP servers at the time”

As of 1987 the Packet Radio Network supported 138 nodes, either packet radios or attached nodes. This technology was also based on a distance vector communications protocol for the routing of the packets. Routing protocols are discussed in chapter 4. This technology although usable did not take into account the scalability of the large numbers of nodes, their rate of mobility, nor the quality of service. Packets were often dropped making the network very unreliable. It was also a very slow means of transferring data in comparison to the more conventional networks. As the number of wireless hops and the distance between these hops increased the transfer rate of the network degraded. According to Goldsmith (2005), the data transfer rate for this network was less than 20 kbps and explains that there was not a strong market for this technology because of these low data rates compiled with the high cost of the technology and the lack of ‘killer applications’.

SUrvivable Adaptive RAdio Networks (SURAN) was launched in 1983 and used some of the existing technologies from the Packet Radio Network. SURAN enhanced the PRNET network and applied its application to producing radios that were smaller, cheaper and more resilient to electronic interference or electronic attacks. This project ran until 1992.

In the early 1990's the concept of commercial wireless networks was becoming more viable and wireless technologies began to emerge into mainstream society. Laptop, PDA and Tablet computers manufactured at this time began to incorporate wireless technology. Many manufactures of communication equipment started producing wireless hardware devices, PCMCIA wireless network cards are an example of this.

According to Bakht (2005) also in the early 1990's the theories and ideas behind the simulation and implementation of mobile nodes were being proposed at several research conferences across the world.

In 1995 the Global Mobile Information Systems (GloMo) project began, this project was funded by DARPA and ran until 2000. As stated by (Papavassiliou et al, 2002) the GloMo (Global Mobile Information Systems) project focused on the development of new ad-hoc wireless technologies.

GloMo proposed and implemented several new protocols for use in radio networks. Further information on GloMo can be found in several research papers, see for example (Leiner, 1996).

The Internet Engineering Task Force's (IETF) goal was to encourage the development of a standardised routing protocol. According to the IETF website (2009) the IEFT and groups within this taskforce were formed in 1986 with Phil Gross as the first Chair. This resulted in the development of proactive and reactive routing protocols in regard to networking and data communication.

Recent research and development into a protocol suitable for mobile ad hoc networks has continued. Simulators such as NS2, GloMoSim and OPNET are available to test the design of a protocol; this is a great advantage in testing and simulating how a protocol behaves because field testing and experimentation are very expensive and time consuming. Karygiannis and Antonakakis (2006) state that the research into mobile ad-hoc networks generally focuses on routing protocol to improve the performance, security and power consumption. The sophisticated simulation tools such as NS2 and GloMoSim allows researchers the ability to study MANETs without the need for physical devices which could be costly and time consuming for field tests.

2.3.3 Applications of a Mobile Ad hoc Network

The initial development of a mobile ad hoc network was funded by DARPA. The Mobile Ad-Hoc Network project was started with the aim that it could assist in various combat operations and was considered of great importance by the military. In a military situation where there is a hostile environment many technologies associated with the communication of data are considered. MANETs are such a consideration because of the ease of deployment, lack of cost and versatility in regard to the constantly changing topology.

The issues associated with security are paramount in that data from the source node to the destination node should not be intercepted and altered before it reaches the intended destination node. Also, the data should not be decipherable by unauthorised users intercepting the data as it travels to the destination node; the data could contain information such as strategic information regarding the location of infantry or resources. The messages sent to the destination node may contain details of the mission objective and if intercepted would eliminate the advantage of surprise. An enemy vehicle such as a tank could be tracked easily through a mobile ad hoc network, the information regarding the location, speed, direction and so forth could be relayed from nodes within the network pinpointing its current position.

Although the development into MANET's started with the intention that it would be used for military operations, a network with this lack of infrastructure that was proven to be reliable and secure could serve extensive applications, both military and non-military.

The government or local council could monitor your car, checking its location, for example, if it entered a zone that required an entry fee. Such zones have been established in London City and are being introduced in other cities that are prone to congestion in a bid to encourage the use of public transport. Average speeds could be measured from a start destination to an end destination and be used to work out if speed limits had been broken for that section of the journey, which could be very applicable in dense areas where mobile ad hoc networks would be easiest implemented.

The government are also looking at new ways to manage road taxation, it has been proposed that the longer the distance driven the more road tax should be charged, thereby not penalising car users who infrequently use their cars. A MANET to achieve this task may be best suitable; as such an infrastructure is currently unavailable for this task. To deploy the alternative wired infrastructure would be expensive and it would be advantages to make use of wireless technology such as Global Positioning Systems.

There is other research in regards to monitoring vehicles using a MANET infrastructure, and although the underlying principles and protocols used in MANET research are used in MANET research when using vehicles, the research community have labelled MANET research with Vehicles as Vehicles Ad-Hoc Network (VANET) in order to clearly distinguish the major differences, primarily being that the speed of the nodes in a VANET usually travel at a higher velocity than that of the nodes in a MANET. Also, VANETs tend to have different applications of use such as communication between vehicles, for example, emergency service vehicles. Flood and Blakeway (2010) explain furthermore the distinctions between MANETs and VANETs in, however, the underlying technology, routing protocols, communication protocols, limitations and obstacles that are faced by VANETs are the same of those that are faced by MANETs.

In recent years terrorist attacks have become a threat to society. If a main communications exchange was to be targeted the entire communication infrastructure could be destroyed. The terrorist would be able to disrupt all communication resulting in chaos. The Emergency Services would be unable to relay information with regard to the current situation at the scene back to its main control centre. An example of such scenario can be seen in Figure 10- A bomb blast in Lahore which is a photograph taken of the Pakistani city of Lahore and which clearly shows the communications infrastructure damaged after a bomb blast.

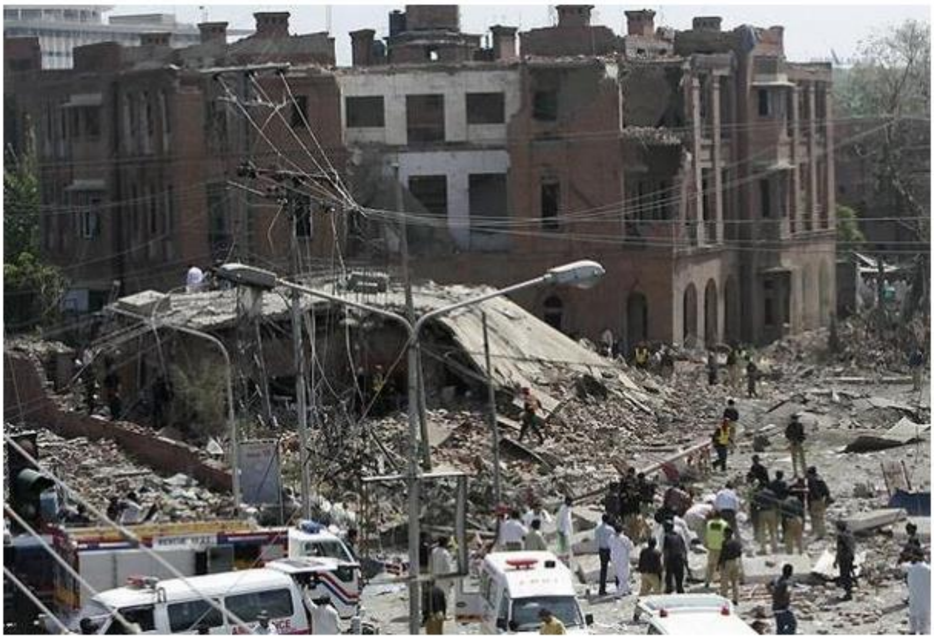


Figure 10- A bomb blast in Lahore

Restoring communication quickly would become crucial. A MANET infrastructure could be set up very quickly and with less financial resources than that of a network with an infrastructure. The installation of wired infrastructure to restore a communications network could take weeks or even months. Physicians for Social Responsibility (2006) states that: "Communications would be completely disrupted, as the electro-magnetic pulse from the explosion would burn out telephone land-lines, exchanges and cell-phone networks".

Another scenario might be used by armed forces. Figure 10- A bomb blast in Lahore shows a photograph of a battle field in Fallujah, Iraq. The figure shows the communications network intact and is operational, however, it might be prudent that a MANET be more suitable for the use of communications between allied forces rather than using a communications network that is more prone to eavesdropping or may be considered less secure.

Other possible implementations may be in an environment where no communications structure exists. Figure 11 - No Infrastructure shows a remote Island which is part of Lemay Island where no predefined communications infrastructure exists.

Figure 11 - No Infrastructure

MANETs can be seen to be applicable in many situations, as described earlier applications consists of situations when communications networks have been damaged, may be considered unsecure or where no predefined communications structure exists. Other applications consist of Local Networking or Personal Area Networking.

A common misconception when discussing applications of a MANET is the functionality of a MANET. There are many applications associated with a MANET and there could be many source nodes and many destination nodes, however, this is dependent on application. Also dependent on application is the size of the network, if the network is to scale, the density of the network and the rate of mobility. The common misconception by the MANET community is that a MANET must be able to serve all applications simultaneously. The view that this is a misconception is shared by (Argyroudis et al, 2007) who clearly take the time to define the literal meaning of ad-hoc in which they describe is derived from Latin and can be translated to English as: to this. In contemporary usage, it is often used as an adjective, translated to: for this purpose, to this end, for the particular purpose in hand or in view. Thus after considering the literal meaning and the many applications for a MANET, it is understandable that the MANET community expects a MANET to do everything, however, one-size does not fit all and the scenario or application of use must be a determining factor. Thus it is on this basis that when constructing the conceptual design and testing the implementation that it will be focused on a particular application with the ability to adapt to changing conditions within that scenario, before discussing the scenario further a definition of a cognitive network will be given that will bring together how the use of the two technologies can complement in order to meet the changing demands of a given application.

2.4 Cognitive Networks

A Cognitive Network (CN) can be defined as a network that is context-aware and is able to adapt to changes in the environment, thus the network is aware of the limitations and requirements of its application (Thomas et al, 2005). A CN should be able to foresee a developing problem and plan a change in the network before the problem occurs in order to mitigate the issues that may arise, an example may be to counteract against congestion or linkage failures. Therefore the CN should be aware of its environment and be able to adapt to changes in this environment.

A Cognitive Network (CN) is a network that is able to adapt to changes in requirements of an application or adapt to progressive detrimental network conditions that could hinder the application from meeting its objectives. CNs make use of several different technologies such as: Biologically Inspired Networking, Autonomic Networking, Self-Managing Networking, Machine Learning, Cross-Layer Design, Cognitive Radio Networks, Knowledge Representation, Adaptive Networking, Intelligent Networking and so-forth, because of the synonymies with these other technologies the definition of CN can be ambiguous. This research defines a CN as a network that is able to observe the state of the network, plan a change, execute the change, but most importantly, learn from the consequence of the change in order to make better informed future decisions. This view is shared by (Thomas et al 2005), in Thomas' research he states that there is not a common, accepted definition and continues to suggest a definition of "a cognitive network has a cognitive process that can perceive current network conditions, and then plan, decide and act on those conditions.".

There is also confusion between cognitive radio and cognitive networks and some associate the two as being the same, software defined radios as described by (Zvonar and Mitola, 2003) operate at the physical layer of the OSI model. Cognitive Networks operate across all layers of the OSI model.

2.4.1 Objectives of a Cognitive Network

This research attempts to answer several research questions in the field of cognitive networks. The structure of this research starts firstly with an introduction that highlights the growth of networks over the last decade and hints at current technology not being able to sustain this growth over the next decade. The following section will very briefly discuss how conventional networks operating in an interconnecting environment, such as the Internet before continuing this discussion with Wireless.

Once the need for a new type of network is established the research continues and discusses the often blurred definition of a cognitive network, followed by a brief history before continuing to applications of a cognitive network and then concluding with a section dedicated to future direction of cognitive networks.

Cognitive networks enable novel approaches that utilise the unused spectrum (which other channels other than the one being used is assigned) in an intelligent and coordinated way, without causing interference or disruption to other users operating in the same transmitting range. Research has shown that measurement studies have found the spectrum (or frequency range) is relatively unused across time and frequency and is explained in much more detail in (Kushwaha and Chandramouli, 2007). Cognitive approaches are being researched that will allow for dynamic access across all channels of the frequency depending on the conditions of the network and the requirements of the application.

A cognitive network is more than the ability of better utilising the frequency spectrum. A cognitive network is described as having characteristics of cognition, cognition is that which comes to be known, as through perception, reasoning, or intuition; knowledge (The American Heritage, 2009). This in essence means that a cognitive network is a computer communications network that consists of technologies such as machine learning, knowledge representation and network management. This is not unlike that of SDR (software defined radio); however, SDR operates at layers 1 and 2 of the OSI model.

Therefore the objective is to make a network adaptive to the needs of the application and for the network to be programmable. In order for this goal to be met, the network must be autonomous, i.e., no management by human administration. The network needs to be aware of its state and the needs of the application. This can only be achieved by the network possessing knowledge of the goals and how to accomplish these goals, thus the network needs to be able to make informed choices based and be able to respond to the dynamic changes in the network.

Currently, this has been accomplished to some degree, however, the network is considered reactive, in that the network adapts only when problems occur, and the network attempts to resolve these problems by adapting only once the problem has occurred, Mackay (2003).

Cognitive networks has been described as being hailed as the next holy grail of (and a potentially disruptive technology in) wireless communication, Mahmoud (2007).

2.4.2 Background of Cognitive Networks

Cognitive networks were first conceived in 2003 by Thomas, DaSilva and MacKenzie. Two years later in 2005 Thomas et al. published work that presented a framework for this novel type of adaptive network. The authors explain that the collection of elements that make up the network observe the condition of the network. This information in addition to the information gathered previously enables the network to make informed decisions based on the information and the conditions within the network, Mahmoud (2007), thus in essence the network can be said to be observing, acting, learning and optimising their performance.

2.4.3 Applications of a Cognitive Network

The overall objective of a cognitive network is to overcome some of the limitations of the current conventional networks, or more accurately, maintain the current level of service or application that the users have become accustomed to. Earlier in this research the unprecedented growth of the Internet is discussed and the need for current communications technology to evolve so that it can continue to offer the services and applications that are currently available. In addition to this over the last few years the Internet has seen the explosion of new media formats being readily available and this media is becoming an inherent application of this vast interconnecting networks. The media in question is that of streaming video delivery (or download) which is adding to the already propagating network traffic. Not only this, but, the introduction of latest video formats such as Blu-ray will mean increased file sizes, thus, data required to propagate the network evermore burdensome on the communications infrastructure.

This research has also discussed wireless infrastructure and the limitations that the current 802.11g technology in that it makes poor utilisation of the available frequency range in the spectrum.

New applications are undoubtedly going to take advantage of communications and the technologies that these communications infrastructures offer. The author of this research predicts the growth and the users of this technology will continue to grow at an unprecedented rate, in particular with wireless technologies. This is already apparent with the development of cloud computing. Overall, The Cloud now boasts more than 25,000 hotspots across the UK, Germany, Sweden, Denmark, Norway and the Netherlands (Meyer, 2008).

"The Cloud is a fast-expanding European mobile wireless broadband service provider with market leading positions in the UK and Sweden and now, as a result of this transaction, we are also the leading independent Wi-Fi operator in Germany," said Steve Nicholson, The Cloud's chief executive. "We predict that the mobile industry will continue to wrestle with poor performing data networks over the coming 12 to 18 months. We therefore see a significant opportunity for The Cloud as the imagination of the mass market is captured by the arrival of new multimedia Wi-Fi-enabled mobile devices throughout 2009."

This thesis aims to clarify the research aims and objectives in relation to the need for a Mobile Ad-Hoc Network with cognitive features. The report discusses the rationale behind the objectives and details two types of network approaches: Mobile Ad-Hoc Networks (MANETs) and Cognitive Networks (CNs) before discussing the rationale for a Mobile Ad-Hoc Network with Cognitive features. The report also highlights the impact and benefits that this will have in computer network communications when a Mobile Ad-Hoc Network approach is adopted. The report concludes with a summarisation of findings.

2.5 Chapter Summary

This chapter discussed the background, some history and application of data communication networks. The discussion began by looking at Traditional Wired Networks (Section 2.1), focused then shifted to wireless data communication networks (Section 2.2), a specific wireless communication network known as a mobile ad hoc network (Section 2.3) was then discussed. Finally cognitive networks (Chapter 2.4) were discussed. Next, Chapter 3 presents a review of current literature.

Chapter 3 Literature Review

This chapter will review the current relevant literature that has been published in regard to the cross-disciplinary areas mentioned, such as Mobile Ad-hoc Networks and Cognitive Networks. The review is presented in three sections, the first two are the disciplinary areas mentioned, and the third is literature that currently reports on the cross-disciplinary area: MANETs with Cognitive attributes. The review focuses on the key papers that are most relevant to this research. The reviewed literature is primarily from online scientific databases although literature also includes research textbooks. This chapter enhances the discussion of chapter 2 by looking at supplementary research analysis and aims to complete Objective 1 which is to discuss the additional elements required for cognition to be applied to a Mobile Ad-Hoc Network.

3.1 Mobile Ad-Hoc Networks

This section of the literature review will discuss the recent advancements in MANETs and identify the research areas in the broader field of MANET technology. The topic areas in this thesis are those that are currently being investigated by the Internet Engineering Task Force (IETF®) MANET research group. The topics consist of: limited power, issues associated with wireless communication, the dynamic environment, communications models, propagation, scenario models and the routing protocols available for the transportation of a packet from the source node to the destination node. This research intends to address: **routing, spectrum allocation** and **mobility** so these areas are the focus of the review. Routing has been chosen because research has shown that different routing protocols perform differently in different scenarios or topologies. Spectrum allocation has been chosen because the research shows that the unlicensed spectrum, although very limited, is poorly utilised which results in congestion. Mobility was chosen because of the nature of the network, devices are mobile and the dynamic topology place a large role in how the network is performing.

3.1.1 Routing

Rahman and Kemel (2011) present an algorithm that includes mobility metrics integrated into current on-demand (proactive) routing protocols although their research only shows results for the DSR (Dynamic Source Routing) protocol. The work details that one of the major challenges is link breakages due to the mobility, a view shared also by Chang and Chen (2002) as early as 2002. The algorithm proposed by Rahman and Kemel (2011) is applied at every node, the source node requests a route (the process continues for intermediate nodes), the route request for the given protocol (DSR in this case) has been modified to calculate a route life time (RTL), the route life time is based on metrics: velocity, radio range and hypothesised location after a given time period. The authors' results show that by adopting this algorithm packet loss is reduced by approximately 90% to that of the DSR protocol without the modified algorithm; however there is a minor delay. Results were obtained by running three simulations in NS-2 (Network Simulator 2) using fairly arbitrary values, i.e. 10, 20 and 30 nodes, no other information about the simulations is given, i.e. geographical location, simulation time, type of traffic. Unfortunately, the author also fails to mention the additional processing requirements and how this overhead may have an adverse impact on processing, battery depletion or other issues that are associated with MANETs.

Yang and Sun (2011) propose an adapted AODV (ad-hoc on-demand distance vector) protocol that is able to store multiple routes to a given destination at the source node, the authors have taken this one step further and implemented a mechanism for route priority using path selection entropy, which allows the adapted protocol to find the minimal node residual energy of each route in the process of selecting path by descending node residual energy. The work presents results from a simulation conducted in NS-2, however, although the claims of the results are that average end-to-end delay has decreased and packet delivery ratio is improved, this is based only on one simulation that consisted of arbitrary values, i.e. 50 nodes in a 1000x1000 meter square running for 10 minutes. This work, like that of Rahman and Kemel (2011) fails to discuss what overhead the proposed algorithm may incur in terms of processing and memory requirements and battery power.

One key area of research is minimising routing maintenance overhead, this theoretically will help in throughput and decrease end-to-end delay by minimising the amount of routing traffic on the network, this has been identified by Kumar et al (2011) who presents an algorithm that claims to minimise overheads when using on-demand routing protocols. The algorithm assigns nodes as a source node that creates an array data structure to store node identities of the nodes within the source nodes transmission range, the data structure is sorted in relation to the distance of the nodes from the source nodes. When a routing request occurs the source node selects the node that is closest from the data structure and this process continues. The destination node replies via the intermediate nodes using the least distance route. According to the research this algorithm helps stabilise routing traffic with increasing number of nodes, although the researchers have not performed an analysis, simulations or experiments to back up their claims.

In summary it can be seen that much of the research on routing concentrates on enhancing routing protocol design to decrease end-to-end delay, several approaches are undertaken from storing multiple routes at each node to reducing route maintenance traffic.

Unfortunately in much of the research, authors do not provide enough detail so that their experiments can be duplicated and their results compared.

3.1.2 Mobility

Some research has been conducted in the area of the mobility of nodes that are self-controlled in order to maintain a topology, such work is produced by Debnath et al (2011) who present an algorithm that allows the nodes to monitor their position in relation to their neighbours and take necessary action to maintain this topology by using actions such as: stop, rush and continue depending on the given situation. The fact that each node is working independently to maintain their relative position means that a distributed approach is being used. Priority levels are given to the nodes and those with highest priority are the nodes that the neighbours sought to maintain mobility by using the actions described earlier. This work is intriguing and although not stated in the paper appears to create a swarm effect with neighbouring nodes maintaining connectivity with higher priority nodes. The experiments performed are arbitrary and of very low scale (5 nodes); it would be interesting to see the effects for a much larger network with a higher population of nodes. Similar work is presented by Grundy et al (2011) however, their algorithm is based on a formal convergence analysis using bio-inspired principles, their algorithm 'Force-based Genetic Algorithm' also works by observing neighbours in a spatial radius and stores information in 'chromosomes'. Although this work is very similar to Debnath et al (2011) there is a greater emphasis and focus on genetics and bio-inspired systems. The authors claim that nodes with confined communications range converge quicker than nodes with greater communication range, which conserves energy. This work demonstrates how similar principles can be drawn by applying different philosophies. It is also interesting that GPS is not considered in either case, however, GPS has limitations (for example, clear line of sight to a global positioning satellite is required) and would have adverse effects on the battery life of the mobile nodes. In a MANET battery life is one of the more important considerations because normally each device will have a self-contained energy source, if GPS were to be used battery life would deplete much quicker causing a much more dynamic and less stable network with fewer links.

In addition to nodes being able to move to maintain links, research has undergone that attempts to predict future links and in particular link breakages by using spatial auto-correlation, work proposed by Jahani and Bagherpour (2011) demonstrates this by adapting the weighted clustering algorithm, originally proposed by Chatterjee et al (2002) and adapted by Karunakaran S. and Thangaraj (2008) so that it incorporates spatial awareness in order to predict mobility within a cluster. Their algorithm shows that performance is increased in-terms of stability of MANETs regardless of the mobility model. However, this claim has no substance in their research as only one scenario was modelled that consisted of 30 nodes in a 100m square. The simulation used was self-developed in C# so validating their results is difficult, in particular because a random walk mobility model was used.

In summary, research has been conducted and is on-going that looks at utilising the mobility of a node(s) in order to maintain links, some of this work is based on nodes within a localised area and GPS is not considered. Other work focuses predicted mobility by using spatial auto-correlation.

3.3 MANET Protocols

This section will explain what a communication protocol is and why it is required in a network. A protocol is a means of enabling communication of two or more machines on a network. The protocol determines the structure of the packet and both machines need to be using the same protocol in order to understand the definition of the packets that they have received. There have been hundreds of different communication protocols designed in a MANET network and much discussion and research has been conducted in this field within the MANET research community; however, few of these are real innovative protocols developed from scratch, they are more likely and adaptations of one of the early developed protocols such as AODV, TORA, DSR.

Each protocol will be taken in turn and discussion on how each protocol initiates the route discovery and maintenance reliable routes for the transfer of data across the network.

3.3.1 Evaluation of Existing Routing Protocols

There are a wide range of protocols available for use in a MANET simulation and many more under development. The protocols are generally classed in one of two specific groups: proactive and reactive. There is a third group which is a combination of proactive and reactive and forms the hybrid group. The examination and critical review of the available protocols will determine which protocols would be best suited for enhancement in order to make the route discovery algorithm more focused and efficient.

Once the protocols have been selected simulations will be performed that will test the protocol further than the current literature available has. Further tests will include how the protocol performs in different densities, different saleability and different networking technologies (i.e. P2P and Client Server), this is especially important as most simulations perform simulations assuming P2P, whereas real applications are more likely to use a more Client Server approach whereby a super node is responsible for generation or receiving most of the data.

This thesis will discuss several protocols and describe some of the innovative adaptations or enhancements made to the original protocols. Based on this research and discussion a choice will be made on which routing protocols will be used for this research. The research considers both reactive and proactive protocol design.

Therefore the purpose of this section of the thesis is to examine and detail the current literature available in regard to protocols available for MANETs. This section also discusses and critically analyses the research conducted in this field of research.

3.3.2 AODV

The purpose of this section is to give a brief overview of the AODV protocol including when it was first theorised and implemented. It will then continue to give a detailed operation of the protocol and how routes are discovered, maintained and updated. This section will conclude with an analysis of the fundamental workings of the protocol. Later sections of this thesis will conduct comparable experiments to try and establish if indeed the AODV protocol is scalable as the author claims or if the AODV protocol is more suited to a particular application.

It is also one of the few protocols that have been further developed from the theoretical stage to the implementation stage in form of the code being produced to enable experimentation under NS3.

AODV (Ad hoc On-Demand Distance Vector) was introduced to the MANET research community in the form of an internet draft in July 1997, Perkins (1997). Research on this protocol continued throughout 1997 – 2003 with various modifications and enhancements. The AODV protocol is claimed to offer quick adaptation to dynamic link conditions, low processing and memory overhead, low network utilization, and determines unicast routes to destinations within the ad hoc network (Perkins and Belding-Royer, 2003).

Quick adaptation to dynamic link conditions whereby AODV attempts to react to changes in the topology of the network, link changes occur when nodes move in and out of range of neighbouring nodes, when nodes first join and initialise themselves and when nodes unexpectedly leave the network. AODV attempts to adapt to these changes by each node on the network periodically checking the status of the link with its neighbouring nodes using 'Hello' packets. Should a link with a neighbouring node fail an error message is sent to the nodes that have been using that link to transfer data from the source node to the destination. This error message is not broadcast to the entire network informing all nodes of the change in link status, just the nodes that participated in forming the link from the sources node to the destination.

Low Processing and Memory Overhead are achieved by AODV only maintaining paths that are in use. If a link is not used within a period of time the path is not stored. This can be argued that this causes an additional overhead as if the path were to be reused at a later stage there would be a need to resend the route request packets again. However, AODV may argue that because of the dynamic nature of this type of network the link is not as viable in comparison to a standard wired network where nodes are less likely to move in and out of range, fall off the network (possible due to power failure), i.e. nodes on a wired network are usually static and have no constraints on power because they are not usually battery power.

Low network utilisation is achieved by AODV selecting the most effective and most reliable routes. This is achieved by using sequence numbers by each node, generally the higher the sequence number the more up-to-date the route is, therefore, being the most viable. This in effect causes low network utilisation as the link is more reliable, theoretically less route discovery should occur.

AODV uses unicast routes for destinations. This means that each route to a destination uses only one possible path. Should the route fail a new path needs to be established, there is not an alternative route until the route discovery packet has been propagated and resent to the nodes on the network. It should be noted however, that AODV does not blindly forward packets for route discovery. As in conventional wired networks and most ad-hoc protocols, a route discovery is broadcasted (a broadcast essential sends to all computers on the network) the difference with AODV over other protocols is that AODV makes use of the TTL (time to live) parameter, this is set relatively low and increments as the route is not discovered, thereby not flooding the entire network, just a localised area. The localised area increases if the route is not discovered. Theoretically the route request could be dispersed throughout the entire mobile ad hoc network if the route is not discovered.

Operation

AODV uses three control packets to establish and maintain the routes within the mobile ad-hoc network; they are RREQ (Route REQuest), RREP (Route REPly) and RERR (Route ERRor). This section will look at how these control messages are used in order to obtain the viable routes and maintain them during the use of the particular route. The control packets are transmitted using a user datagram packet (UDP). UDP is defined as a protocol that operates at the Internet standard network layer and provides transport and session layer protocols. A checksum and additional addressing information is added to the header of the UDP protocol. UDP does not guarantee delivery nor does it require a connection which makes it lightweight and efficient. The application using UDP must take care of all errors and data that require retransmitting.

This essentially means that the UDP does not initially require a connection and does not guarantee delivery of the packet. The method of using UDP generates a lightweight method of obtaining a route.

RREQ is used to initially obtain the route from source node to a destination. The route request control packet is broadcast to nodes within the network. The route request packets continue to transmit starting within a localised area and the area expands the localisation of that area until the destination node is reached or until an intermediate node is discovered that contains information of the route. The route request packet uses an algorithm based on the Bellman algorithm (Bellman, 1958), however, AODV has implemented sequence numbers to overcome the potential problem with the Bellman algorithm of counting to infinity, what this means is that the original Bellman algorithm contains a flaw whereby the route request packets could theoretically continue throughout the entire network infinitely. The use of sequence numbers and the time to live parameter overcomes the problem of count to infinity, thus, killing the route request if the destination is found, or killing the route request should the destination not be available. Sequence number is discussed in more detail towards the end of this section.

After analysing the results of packet delivery ratio, network load, end-to-end packet delivery delay, number of packets forwarded and number of packets dropped, it has been established that no one protocol is ideal for use in a MANET environment across different scenarios, node density, 'world size' i.e. the size of the simulated environment, and the range of the node transmission range (which is greatly affected by the power limitations of the node and obstacles in its place). For example, Destination-Sequenced Distance-Vector (DSDV) is considered non-scalable; however, it outperforms AODV and ZRP in scenarios that consist of a small world despite the density of the nodes within that world.

This poses the question, can the network automatically switch to another protocol depending on the factors discussed in the last paragraph. It also poses the question what are the overheads and are they too high to implement a network that switches from one protocol to another dependent on the variables in the network. A MANET network as discussed earlier is highly prone to change and thus being very dynamic in nature, this poses a further question, if a network can switch from one protocol to another on a MANET, how often should the variables be established in order to initiate a change in communications protocol, and furthermore, is it viable to change from one protocol to another subject to the information gather and how will this be accomplished? Will the network be inoperative whilst each node changes to the required communications protocol?

AODV is one of the popular protocols currently available in a MANET; one reason for this is that AODV protocol is highly scalable. I will discuss each of these adaptations starting with research conducted in 2005 to the latest research being conducted.

M-AODV which was theorised in 2005 where the authors (Shih-Hsien et al, 2005) attempt to help control redundancy, contention and collision associated with route discovery and broadcasting messages. This is achieved by controlling the broadcasting of the route request by partitioning the network into zones using a Zone-based Controlled Flooding (ZFC) scheme.

AODV-LRQ and AODV-LRT (Pan et al, 2005) were protocols developed based on AODV where the authors proposed an efficient approach to repair error links quickly. This used the parameters throughput increment and routing overhead comparison and drew conclusions by comparing these parameters. Simulations were run and by using the number of successfully transmitted packets in a given time as a metric and the number of route request packets transmitted as the cost, the authors proposed and defined a term called bonus gain that enabled them to capture the concept.

MAODV (Shih-Hsien et al, 2005) whereby the authors proposed a mobility prediction algorithm which was based on AODV. This algorithm requires that each node stores information on their neighbours nodes distance and by predicting the neighbouring nodes direction and speed. Assumptions are made that all nodes have the same transmission power. The author conducted simulations using NS-2 which showed that in comparison to AODV, MAODV had an end-to-end delay which was less, higher control overhead, lower packet delivery ratio and a higher average number of hops.

3.3.3 DSDV

DSDV is a proactive protocol and works by flooding the network with route request which creates an unnecessary overhead. The latter impacts on each node's resources. It has been proven that DSDV works better than most protocols (such as AODV) in simulations on small networks with a small number of nodes. This is due to the fact that DSDV is a table driven routing protocol and the table is updated periodically. Therefore, the route from source node to destination node is already known before data needs to be transmitted to a destination node(s). However, further simulations and evidence show that as the number of nodes increase within the network the unnecessary blind broadcasting of route request packets causes such an overhead that the network reaches a critical condition and fails (Ramesh, 2010, pp. 183-188).

3.3.4 TORA

The purpose of this section is to give a brief overview of the TORA routing protocol including when it was first theorised and implemented. It will then continue to give a detailed operation of the protocol and how routes are discovered, maintained and updated. This section will conclude with an analysis of the fundamental workings of the protocol.

The Temporally-Ordered Routing Algorithm (TORA) was initially implemented in 2001 for NS2 (Park & Corson, 2001). The protocol features distributed execution which allows the assignment of parts of the routing task to neighbouring nodes. Each node is also able to store multiple routes for each destination node.

The TORA protocol is distributed because the TORA algorithm is run for each destination that is required. Each instance of TORA is run independently which in effect achieves the distributed feature of this protocol.

Downstream Traffic is traffic propagated at the remote computer and arrives at the local computer. This can be thought as the number of packets entering the local computer. Upstream Traffic is propagated at the local computer and sent to a remote computer. This can be thought of as the number of packets leaving the local computer. This is usually requests.

TORA works by each node on the network setting a height variable. The height variable is not the physical height of the router or node but the rate at which the node is able to transfer upstream traffic in relation to transferring downstream traffic. This in effect creates a direction of which traffic can flow between two nodes. The higher router will transfer data to the lower router.

“TORA assigns directions to the links between routers to form a routing structure that is used to forward datagrams to the destination. A router assigns a direction ("upstream" or "downstream") to the link with a neighbouring router based on the relative values of a metric associated with each router.” (Park & Corson 2001)

Nodes cannot forward packets to nodes that are considered higher than themselves. This enforces a unidirectional method of data communication. (Park & Corson 2001) explain that this method of establishing routes forms a loop-free, multi-path routing structure.

This method of route establishment differs from the AODV protocol because there can be cases whereby the shortest hop route is not the route chosen to transmit the data from source node to destination node. This creates an additional burden on some routers as the packet will be transmitted through one or more intermediate nodes.

Routers, or nodes that route packets in a MANET that uses the TORA protocol only store route information about their neighbours (nodes that are within their transmission range) this limits the amount of information stored at each node thus limiting the overhead that is required when establishing routes for this type of network.

Route establishment is achieved by one of two means when using the TORA protocol, the first technique is a reactive technique whereby the source node sends out a route request for a destination node. This method of route discovery means that routes are stored for required destinations only, not for all destinations. This will limit the communications overhead by only obtaining routes that are required and not obtaining routes for all viable destinations (most of which will probably never be used). This methodology is especially important in a MANET because this prevents routes for destinations that may never be required being stored, thus preventing the periodic flooding of the network with route requests for all source and destination nodes. This not only helps limit bandwidth usage of the network but also conserves limited battery power of the nodes and utilises the processing capabilities of the nodes.

The second method of route discovery is proactive. This is whereby a destination will proactively obtain a route for a source; however, this method of route discovery is usually reserved for devices that form a gateway to a wired infrastructure such as a LAN or the Internet. These devices, because of their tasks are not likely to have the same constraints as the other nodes in the network; therefore, battery power and processing power are less of a concern. This type of device would also usually be static, thus the mobility issues for this device are also less of a concern.

TORA attempts to limit topology changes (route changes) to a small area of the network in order to limit the communications overhead.

3.3.5 Protocol Summary

Section 3.3.2 discussed AODV, Section 3.3.3 discussed DSDV and Section 3.3.4 discussed TORA. Each protocol has similarities but operate differently. Table 2 - Protocol Summary summarises each of the parameters for each of the protocols.

Parameter	AODV	DSDV	TORA
Source Routing	No	Yes	No
Topology	Full	Full	Reduced
Broadcast	Full	Full	Local
Update Information	Route error	Route error	Node's height
Update Destination	Source	Source	Neighbours
Method	Unicast	Unicast	Broadcast

Table 2 - Protocol Summary(Gupta et al, 2010)

This concludes the discussion on the operation of protocol design. Next the discussion focuses on Cognitive Networks.

3.4 Cognitive Networks

This section of the literature review will discuss the recent advances in Cognitive Networks and identify possible methodologies that could potentially be applied to a Mobile Ad-Hoc Network.

3.4.1 Cognitive Mobile Ad-Hoc Networks

This section of the literature review will highlight research in the area of Cognitive Mobile Ad-Hoc Networks. The current publications available in the research databases are of limited number; therefore all found literature is reviewed.

Younis et al in (2009) describe the concepts and challenges for automatic design and reconfiguration of cognitive MANETs. The authors present a framework that details constraints and impose design parameters that should be met. The authors also mention spectrum allocation, how this resource is used and how secondary users could potentially use licensed spectrum allocation. The research also indicates that current simulation software such as NS-2 and OPNET are not sufficient for the testing of MANETs with cognitive features and that an analytic model is required. The paper is very informative and presents areas of research that relate to tuneable parameters. Their research shows that by using tuneable parameters that traffic loss is reduced.

Vaman, D.R. and LijunQian (2008) put forward the advantages of combining sensor networks and MANETs, the authors clearly identify the goals of both types of networks and list the restrictions in place. Examples to real world applications are given. This paper although presented as a cognitive mixed approach of cross communication between the two networks fails to identify how the cognitive approach is adopted. As in Younis et al (2009) the authors state that spectrum allocation is restricted and that cognitive networks are able to sense the spectrum in order to utilise the spectrum more efficiently. The authors fail to detail how this is achieved.

Kennedy (2007) details a cognitive approach to MANETs in that uses a cross-layer approach and that attempts to predict route discovery. The author explains that reactive protocols are not always applicable to time sensitive applications and that proactive protocols are prone to issues with scalability. The author identifies a conceptual design whereby route prediction can occur before a link breakage occurs. This approach uses fuzzy logic and classifies such attributes like node speed and link quality, for example speed could be classified as slow.

Boutin et al (2009) agrees with the authors Younis et al (2009) and Vaman and Lijun Qian (2008) regarding the need for networks that better utilise the radio spectrum allocation and presents a new interference estimation technique for a smart antenna equipped MANET. The work attempts to show that radio spectrum can be shared with a primary (legacy) network as in Younis et al (2009) and that nodes within a MANET can share the spectrum with the assurance that the MANET will not interfere with the primary networks goals.

Further research in spectrum allocation and MANETs is conducted by Xiuhua et al (2007); however, the emphasis of this research is on spectrum handoff when sharing the allocation with another network. The author takes time to explain that a handoff usually occurs if the primary user of the network requires the bandwidth or if the performance of the network degrades. The author also states that in a MANET, a handoff would also occur when a mobile device moves out of range of a wireless node into range of another. Thus, the greater the mobility of the nodes in the MANET the more frequent that handoffs may occur. The author states that because of the frequency of handoffs there will be considerable delay in the network as each handoff occurs, this results in changes being required in each layer of the network protocol in order to accommodate the new wireless radio environments. The author continues to substantiate this delay by stating that the network protocols would have to shift from one mode of operation to another. The paper did not detail time measurements that each handoff may incur or give situations of where network viability may cease should the handoffs increase to a substantial number within a given interval. The author does present a model whereby the handoffs could be managed by a mobility management module; the author does not incorporate how delays will be handled with handoffs.

The authors Rosset et al (2005) do not discuss any specific cognitive feature of a MANET so unlike Younis et al (2009), Vaman and LijunQuin (2008) and Boutin et al (2009) spectrum allocation is not considered, however, there is mention of being able to control the bandwidth for QoS enforcement. The paper primarily introduces a middleware application called ActiveEdge-M and claims to support a number of application services to facilitate the creation of agent-based applications in a MANET environment. The author continues to claim that the middleware supports: “asynchronous messaging, controllable bandwidth consumption, self-management, survivability, high-level middleware services, and efficient role-based distributed knowledge management". The paper states that the middleware actively or passively monitors failures, track agent lifetime, move and restart agents as necessary, and as in Kennedy (2007) switch to alternative protocols as necessary. This paper is very ambitious in its approach, in being so ambitious and trying to cover so much, fails to give any detail of how the middleware actually meets any of the objectives that it claims. The paper is descriptive but provides a good overview of many of the services that middleware software could support. Further research into the ActiveEdge-M middleware was sought but it appears that since publication of Rosset (2005), research with ActiveEdge-M ceased as no additional literature was found.

Al-Fuqaha et al (2009) present a novel approach to spectrum utilisation with the use of cooperative mobile nodes. The mobile nodes that are cooperative move to establish or strengthen communication links. The authors assume that the cooperative mobile nodes are equipped with GPS and therefore are aware of their position. The research also presents and proposes a channel estimation algorithm. The algorithm uses a wavelet transform and flip-flop filter that allows for the monitoring of frequency and time domains to establish if the channel is highly utilised or not. Similar to Kennedy (2007), Fuzzy logic is applied; this is based on five rules defined in their paper Al-Fuqaha et al (2009). The authors have developed their own simulation software to validate the results. The results indicate that by applying cooperative nodes that move to form links and using channel utilisation techniques that bit error rates within the infrastructure can be significantly reduced.

Jin-Hee (2009) proposes a trust management system for cognitive mission-driven group communication systems in MANETs. The trust system is based on observing nodes energy levels and allows for a trust chain for multiple hops, the authors explain that current proposals for trust systems often only account for one hop. The research is based on foot soldiers with low mobility rates and describe various levels of trust assigned based on their cooperation and energy levels, an example given is that when a nodes energy level reduces suddenly that this may be due to the node counteracting an attack, thus requiring additional processing. Although this paper is informative, novel in its approach and much research is conducted the cognitive aspect fails to be addressed.

A Mechanism Design-Based Secure Architecture

Rachedi et al. (2008) starts by introducing a proposition for avoiding a single point of failure in a MANET when using the cluster based approach. The paper explains the limitations of this approach in terms of its inability to form clusters with a central head cluster and two trusted nodes that form a dynamic demilitarizes zone (DDMZ) in that this would increase the size of the cluster and thus reduce the total number of clusters. The author continues by stating a second limitation which is based on the selection of the cluster head and the two nodes that form the DDMZ. In order to perform the duties of a cluster head a monitoring program is required to run, thus using often scarce resources of the mobile node, such as battery power, processing power and memory. The limitation of such a selection is that nodes could report false information regarding their status in order to prevent being selected as the cluster head or one of the two DDMZ's. The approach that the authors in this paper have taken is to implement a system that offers incentives in being a cluster head or DDMZ so that the nodes cooperate and report accurate information in order for the selection process to be fair. This fair selection process helps build a robust cluster using a cluster head and at least two nodes acting as secondary cluster heads should a single point of failure occur. The incentives are that a node accumulates points for reputation, should a node have enough point to be considered reputable an intrusion service is offered. The authors have also implemented a catch and punish model. The criteria for selection are based upon strength, mobility and trust.

The paper details how the nodes are selected and does detail contingencies to protect the nodes designated as head clusters. More research is required to fully understand the implementation of such a proposal. Impressions indicate that an overhead initially

3.5 Issues that affect the performance of a MANET

As a result of the literature reviewed several factors that could adversely affect the performance of a MANET were identified, these are congestion, routing and link-breakages. The factors are often caused by certain characteristics of a MANET which are : Mobility, Limited Resources, Bandwidth, Energy Models and Lack of Fixed Infrastructure. To conclude this chapter a brief overview for each of these characteristics is presented.

3.5.1 Mobility

The very nature of a MANET is the mobility of the participating nodes within the network; because each node can be mobile it is difficult to allocate resources in order to serve the requirements of an application. A viable route that meets the requirements of a network may be valid for a short time because of the mobility of the nodes. For example, an intermediate node participating in the forwarding of packets for the intended destination node or the next intermediate node in a chain of intermediate nodes to the destination may move out of range of its neighbours thus breaking the link. Viable links have to be updated and maintained more frequently in a MANET because of this. Routing protocols play a vital part in this role.

3.5.2 Limited Resources

The nodes within a MANET as discussed in the last paragraph are mobile by their purpose; because the resources are often limited. The resources that need to be accounted for consist of battery power, transmission power (affecting transmission ranges), processing power and storage capabilities. Each of the resources will have an adverse effect on other resources. For example, if a node needs to buffer packets there are additional requirements on storage and processing resources, thus having a direct impact on battery power. It is beyond the scope of this research to address each of these items in detail, but in order to produce a conceptual model that aids in performance of the network, the limited resources need to be considered.

3.5.3 Bandwidth Limitations and Licence

Wireless communications operate within a specific range of frequencies. This range of frequencies is often split into channels and the wireless device selects and operates on this channel (and operates within that range of frequencies). In order for wireless devices to communicate both the sender and receiver should operate within the given range of frequencies (on the same channel). Normally the communications protocol that is being used determines the frequency band and how this band is split into channels.

The range of frequencies that a wireless device can operate is quite limited (normally in the 5GHz or 2.8GHz bands). If a wireless device wanted to operate or communicate outside of these bands a licence is normally required. Licences prevent the unauthorised use of sending data traffic within the band that the licence covers. The holders of licences are typically government enforcement agencies, government emergency agencies and the media (FM/AM radio) or television.

3.5.4 Energy Models

As mentioned in the earlier sections the devices that communicate in a mobile ad-hoc network are typically mobile devices. Therefore the devices normally operate for periods of time without an external power source and have to rely on their limited internal power source (i.e. battery). The nature of the device and the application(s) that the device runs determine how much power is drawn from the power source. Different devices will have different life-times (power depletion times), for example one might expect a mobile phone power source to last longer than that of a laptop – however this would depend entirely on initial energy state and application.

3.5.5 Lack of fixed Infrastructure

The lack of fixed infrastructure is also a limitation in mobile ad-hoc networks. There are no dedicated devices to handle the operation of the network, for example there are no dedicated routers, switches or other networking hardware to handle the flow of data. Therefore each device that operates within a mobile ad-hoc network has to undertake the roles of the network hardware to manage the flow of network traffic. One of the more common roles that each node has to undertake is the routing of traffic which would normally be handled by a dedicated router in a network with infrastructure.

3.6 Chapter Summary

This chapter has identified several factors that affect performance in a MANET and has focussed on the research of others. In this research several gaps have been identified, for example, some researchers use arbitrary values for their simulation designs without justification of why the values were chosen. Some of these parameters consist of number of nodes, mobility models and geographical area of the simulations. In addition to this many parameters are not discussed at all, for example transmission range, making duplication of their experiments difficult or impossible.

It has also been established that there is no standard benchmark or guidance for the configuration and running parameters of the simulations. Nor, is there any standardised simulation software used, albeit NS2 is more popular there are many alternatives and in some cases the research will create their own bespoke simulation software.

Chapter 4 Problem Analysis of Mobile Ad-Hoc Networks

The aim of this thesis is to present and implement a conceptual model that demonstrates the advantages that cognitive features offer when applied to a MANET environment. MANETs should be able to observe, learn, plan and execute changes in order to help alleviate against congestion and to prevent or recover from link breakages. The network can do this by monitoring and addressing changes in topology and by making better use of the available bandwidth. This will help the network meet the demands of the application and have a better chance of meeting the mission objectives. However, first expansion of the factors that hinder network performance which were identified as a result of conducting a literature review are discussed.

4.1 Problems affecting MANET Performance

The devices on a MANET may be mobile and are normally powered by a limited energy source. Should the energy source deplete the device on the network dies, which alters the topology of the network and may cause unstable or unusable links.

On data communication networks with a static infrastructure, the type of traffic discussed in the Chapter 2 can be accommodated by allocating bandwidth or prioritising data. This allocation and assigning priorities is not so easily accomplished on a MANET because of the constantly changing topology and link failures.

Furthermore, mobile devices are often restricted in processing power and memory capacity. From the other side, high processing may have an adverse effect on the lifetime of a node.

Therefore, there is a need for an adaptive network that can self-configure in the event of change or requirements in order to support a MANET infrastructure that accounts for different types of traffic, changes in topology, application requirements, and to make better use of bandwidth allocation.

The network should not only react to the change in network state or to the requirement of the application but also be able to recognise the change or requirement before they occur. Therefore this will create an adaptive network with cognitive features rather than simply making the network adaptive.

This view is supported by (Santivanez, 2007) which claims that networks are still not aware of their state or needs and do not have knowledge of their goals and ways to achieve them, nor are they able to reason for their actions.

4.2 Congestion

Congestion can be defined as excessive crowding and generally means that a blockage has occurred due to the high volume of instances trying to navigate through a given path. This definition works for many types of congestion from biological nasal congestion to vehicular traffic congestion. In data communication networks congestion occurs when intermediate devices (for example: hubs, switches and routers) or the destination receives data more quickly than it can process. There are mechanisms that exist in order to counteract congestion, for example, queuing algorithms and flow control techniques, however, inevitably if the flow of data does not slow enough for the node to be able to process the queue will become full and data will be lost, the data packet dropped (Song et al, 1997). If the incorrect flow control mechanism is used there may be poor resource utilisation and increased control traffic on the network, for example, the stop and wait flow control algorithm places control traffic for each block of data that has been received at the destination as an acknowledgement of the received data and the indication that it is ready for the next block of data. With stop and wait flow control the source will not send any further data until the acknowledgement has been received.

Stop and wait flow control is one of the simplest forms of flow control and other flow control algorithms operate by placing much less control traffic on the network. Examples include those algorithms that make use of a sliding window which is used to acknowledge many packets of data in one acknowledgement (Wang and Zhan, 2011).

In a MANET intermediate nodes are responsible for forwarding packets to the next node in the route or to the end destination; therefore congestion can occur at intermediate nodes along the route and at the intended destination. In this thesis, a node is defined as a connection point in the network and could be any device that is able to transmit and/or receive radio waves which are the electromagnetic representation of data.

There are various forms of traffic that traverse a data communications network, one type of traffic already discussed is traffic generated by applications, another type of traffic that has been touched on is control traffic as mentioned when giving an example of a flow control technique; this is in the form of packets that represent data intended for the control of the network. Other control traffic is generated by route discover or route request packets. Routing is discussed in section 1.3.

There are also various physical design considerations that limit the bandwidth, thus restricting the amount of traffic that can traverse the data communications link at any one time. Fibre optical medium for example has a higher bandwidth than coaxial or twisted pair cable and thus the capacity to transmit a higher volume of data at a given time. Wireless networks also have bandwidth allocation considerations, which are considerably more restricted as opposed to that of wired communication networks (although technically fibre optic is not wired the data does traverse a dedicated set path). The more restrictive the bandwidth for the medium, the less amount of traffic can traverse that medium at a given time; limited bandwidth with high data rates is a primary cause for congestion. The physical characteristics of wireless medium are discussed next.

4.2.1 Spectrum Allocation

Spectrum allocation is the allocation of bandwidth within a given range of frequencies. Wireless devices can transmit radio signals at limited positions of the electromagnetic spectrum depending on the technology and protocol being used. Bluetooth for example can operate within the frequency range of 2,400-2,483.5 MHz Estrin et al. (2000). Wi-Fi operates within this frequency too but also supports communication in the 5GHz frequency range depending on the protocol implementation being used. Wireless protocol implementations, in particular the 802.11 family are discussed in the next section.

In order to transmit data from a source node to a destination node there needs to be some mechanism of access to the electromagnetic spectrum. Current legislation, regarding the partitioning of the electromagnetic spectrum, states that there are unrestricted sections that are publicly and freely (in terms of financial cost) accessible. This is the unlicensed spectrum and operates in the 5GHz or the 2.4GHz range. Ranges outside of 5GHz or the 2.4GHz range require a licence and access to these ranges is restricted to the application or service that holds the licence. An example would be the United Kingdom emergency ambulance service which operates using 12.5 kHz channels between the range of 166.1 kHz and 166.85 kHz frequency band (OfCom UK, 2014).

Therefore, because there is much demand for spectrum and much of the spectrum is licensed there has been research conducted that attempts to use the unlicensed ranges more efficiently or impinge on the licensed spectrum without disrupting the communication of the primary licensed service.

Research conducted in spectrum allocation shows that much of the scarce unlicensed radio spectrum is poorly utilised (Zhen et al, 2010). Researchers have begun to address spectrum allocation issues by applying spread spectrum techniques that are found in CNs. Spread Spectrum is the technique of spreading the electromagnetic signal over a wider range of frequencies as a function of time, thus making better use of the spectrum. Spread spectrum techniques also make it more difficult to eaves drop, (Electronic Communications Committee, 2000).

There are generally two techniques for applying spread spectrum techniques, frequency hopping and direct sequence; these are discussed in more detail later.

This research attempts to address congestion by allowing the network the ability to switch to another frequency based on the requirements of the network. The decision the network will make should be cognitive, i.e. based on the network's previous experience. The option of changing the transmission frequency will be available for the network together with other options discussed below. Once network has decided to switch to another channel, the time and context associated for this decision will be saved in the managements' memory and later evaluated as successful or unsuccessful. The theoretical design of this procedure is discussed in chapter 5 and implementation is discussed in chapter 6. Next routing algorithms and link breakages are introduced.

4.3 Routing

In order to determine the best path from a source node to a destination node a routing protocol is used. Different routing protocols use different algorithms to calculate the least cost route. Two well-known algorithms that routing protocol designs are based on are the Bellman-Ford algorithm and the Dijkstra algorithm; and both attempt to compute the least cost path.

Two popular routing protocol designs for a MANET are the Ad-Hoc On-Demand Distance Vector (AODV) and the Direct Sequence Distance Vector (DSDV) routing protocols (Vyas and Chaturvedi, 2014).

Research shows that MANET communication protocols perform differently in different scenarios, for example, the Ad-Hoc On-Demand Distance Vector (AODV) routing protocol works better than Direct Sequence Distance Vector (DSDV) routing protocol in a highly scalable environment (for instance, when the number of nodes in the network and the geographical area occupied by the network increases), whereas the DSDV protocol works better in a small, less dense environment (Li and Mohapatra, 2002) and (Karygiannis and Antonakakis, 2006). Should the density or the scale of the network change it may be appropriate to change the routing protocol in use. Initially the research will look at the use of two protocols; it is the aim of this research to use a proactive protocol and a reactive protocol. Proactive protocols are known to perform better in less dense networks of a small scale, but performance of the network deteriorates as the density of the network or the scale increases (Vyas and Chaturvedi, 2014).

Depending on the routing protocol in use, different levels of control traffic are generated at different times. DSDV produces more control traffic which places additional traffic on the network which could contribute to congestion. This said DSDV has advantages when compared with the AODV protocol such as the validity of the routes. When using DSDV the routes are already established and are considered more reliable than that of AODV, in particular in smaller less dense networks (Istikmal et al, 2013).

This thesis shows that a network can be aware of its environment and through observation by the network select an alternative and more appropriate routing protocol that better serves the network in terms of performance (end-to-end delay and dropped packets) should the shape of the network change.

4.3.1 Link Breakages

When a node is in communication range of other node(s) they form links in order to collaborate on the network by forwarding data intended for the destination. The link is predominantly a means of establishing a connection to the communications medium that is shared by the other nodes. There are several reasons why a link breakage could occur; for example, a node may move out of communication range or the energy level of the node may deplete to a level below a transmit or receive threshold so that there is not enough power to transmit data or receive data. The node may lose power completely (such as a power failure or shutdown of the node), which would again cause a link breakage.

The nodes in a MANET are mobile and should a link breakage occur there should be the mechanism of a higher authority that is able to control other nodes within the network in order to re-establish a communications link to facilitate the transmitting of data from the source to the destination (or intermediate link). This should be possible when there is no alternative path to the destination. This research will allow control of such nodes by the network and the implementation is discussed in the design and implementation sections.

4.4 Simulation Software

In order to test the implementation of cognitive features in a MANET environment a network simulator will be used. Simulation software can be described as sophisticated computer programs that attempt to create a representation or model of a physical system or particular situation. In the case of MANET simulation software, the software attempts to model many aspects of how a MANET would operate in the real world using hardware with radio capability that are mobile.

The software attempts to create an artificial environment mapped as closely as possible to that of a real environment; such considerations of the real environment must also be simulated, these may consist of: geographical area, propagation model, radio frequency, radio range, number of nodes, initial node placement, movement of the nodes, transmission rates between nodes, how the connection is established, routing protocol and application protocol. The software will also operate a timeline of scheduled events, i.e., when a particular node will join or leave the network, when to transmit, whom to transmit to and so-forth (Blakeway and Merabti, 2010). All of which is *usually* set by the user prior to running the simulation. In this case, the network will be cognitive so some of the configuration will occur during the simulation run.

Most research conducted in network design makes use of network simulators because the cost and time implications of setting up a real network are high. In addition to this there are geographical constraints. Other consideration might be staff resources, lodging and cost of the equipment. In addition to this research experiments should be possible to reproduce which would be quite difficult without simulations.

Most researchers favoured NS2 (Blakeway and Merabti, 2010) and this trend continued for many years (Kurkowshi and Colagrossa, 2004). There could be many reasons for the popularity of NS2 but some key features are that, it is open source, it is well established and there is a lot of documentation available, there are many protocol designs implemented and there are tools that allow visual representation of the network and analytical tools for the analysis of results. In 2010 48% of the research community used NS2 and in 2004 this was slightly lower at 44.4% in order to validate their research. Other popular simulation software for modelling networks are Opnet, GloMoSim, MATLAB and Qualnet.

NS3 is intended to replace NS2 but currently adoption has been low and many researches are still using NS2, this may be because NS2 has been around for much longer than NS2 and there are many examples, tutorials and manuals. In regards to NS3 adoption is growing but there are limited tutorials and examples. In some cases the documentation that accompanies NS3 is incomplete.

OPNET was also considered and is much easier to use that NS2 or NS3, however, OPNET does not allow for changes to the network during the simulation run which is imperative for application of cognition.

In this research, despite the lower level of support, Network Simulator 3 has been selected because it has been implemented in C++ which allows the creation of many methods in order to apply cognition; new data structures available in C++ such as Vectors is also another advantage. Each object created maintains a pointer to that object; common objects can be aggregated to other objects. One of the difficulties in NS-2 was how different headers of a packet could be accessed. Network Simulator 3 provides easier access to different headers of the packet. NS3 also performs better than NS2 in the way that memory is managed. The C++ language allows much better memory management. There is also support for the visualisation of the network topology using Network Animator.

In order to ensure that the results can be replicated the code will be made publically available.

4.5 Addressing Problems Identified

In order to solve the problems identified new additional roles on some of the nodes will be established. Responsible nodes are selected that will be aware of the current network state in the local vicinity related to the source node they serve. It is planned to select the "best possible" candidate; the election process is discussed later in Section 5.5.1 Candidate Node Election and the battery life of the node, the distance and the predicted time in range will be taken into account.

Each selected node (called manager node) will take care of its own source node(s) by checking frequently the node's performance; if performance drops, the management node will issue a request for improving the performance in one of the three directions above based on the memory of previous actions or the manager node’s experience.

There may be multiple manager nodes; the requests from different nodes might contradict each other. The requested action might be recently performed, and the consequences are only about to be revealed. Thus, the establishment a network command centre (NCC) that will summarize all the current requests and compare them with the recent actions to either grant permission for an action or deny the action will be implemented.

The network performance is to be measured in the rates and total numbers of the received packets. However, the network is aware of the other information, such as dropped packet rates, send packet rates, end-to-end delays, etc.

The major outputs of the thesis consist of a theoretical architecture which consists of cognitive features applied to a MANET, where cognition is applied for all three directions discussed above and a cognitive MANET testbed implemented in NS3. The objectives set up in chapter 1 are met by constructing those two objects.

The set of theoretical procedures designed in this thesis are tested by the designed simulated MANET.

The cognitive MANET testbed is tested throughout its construction (discussed in detail in Chapter 6, 7 and 8) and after it is constructed, when it works as expected, the testing of the designed procedures will be carried out by running various simulations (the results are discussed in Chapter 9).

Various scenarios are used to test the model, each time before it is run; the main features are discussed and the expectations towards how the cognitive network will perform comparing to a MANET without cognitive features applied.

Chapter 9 presents results and the evidence, that the novel procedures together with the implemented network bring significant improvements in network performance.

4.6 Chapter Summary

This chapter has discussed the problems associated with a MANET there were identified in Chapter 3. In this chapter more detail how congestion within a network hinders the performance of the network, how limited is the unlicensed radio frequency spectrum, how routing protocol play a major part in the establishment of paths between source and destination and how link breakages occur because of the mobility of the nodes within the network were discussed.

Then discussed were the simulation software and a detail of the approach to be used in testing a MANET with cognitive attributes applied to mitigate against the factors identified. In the next chapter a design is presented which details how the cognitive attributes are to be applied to a MANET.

Chapter 5 Design of Cognitive MANET

This chapter begins with the discussion how most cognitive networks operate; the cognition cycle and the components that make cognition are discussed. Next the topology of a MANET is discussed (Section 5.2) so that the roles of the nodes available in a MANET can be explained. Some of the roles the nodes undertake are not typical in a MANET environment and have been created for this research, for example, a controlled node and a management node. This section also discusses Initial Node Placement, Mobility Models and Hardware. Section 5.3 continues the discussion of cognitivity and gives an overview of cognition before detailing how to select suitable candidates to become a management node and how to select the "best candidate" for this role. Also discussed is how to check relationship states between the selected candidate (the manager node) and the source node to ensure that the manager node is still a viable candidate for the source node.

Section 5.4 gives an overview of memory from a psychological point of view and how this relates to computer science. In this section attentiveness and information processing is discussed. The information intended to be captured (Section 5.5) relates to position, distance, velocity and packets is then discussed. Once the information that is captured is discussed, the discussion turns to Information Processing (Section 5.6), i.e. how to use the information that is captured. Time is spent discussing how the management node makes use of the information captured.

Next consideration is given to multiple managers (Section 5.7) and a Network Command Centre cycle is introduced to handle multiple requests from multiple manager nodes. This section also details how requests from managers are considered, authorised or denied.

The novelty of this design was briefly highlighted in section 5.8 before concluding with a chapter summary in section 5.9.

5.1 Cognition

Most cognitive networks implement a control loop which is often referred to as a cognitive cycle (Esch, 2012). The cycle allows a node participating in the data communication network the ability to imitate or mimic cognitive features with the intention of increasing the overall efficiency or performance of the data communications network.

The cognitive data communications network should be able to adapt to the changes in network state. In addition to this there should also be a way to predict changes of the network state, reflect on these changes and formulate a plan when required. If, as a result of performing a state prediction it indicates a drop in performance the network could try to determine the reason for the performance drop. For example, there could be an occurrence of a link breakage, congestion or the most appropriate routing protocol is not being used for this or the predicted network state.

If, for example, there were a hindrance (resistance, delay or obstruction) in the operation of the application the network should be able to formulate a plan to mitigate against the hindrance with an attempt to prevent the undesired level of service by considering the resources available and making better use of them.

There should also be a mechanism in place that will allow the recording of information, actions and the consequences of the actions to a data structure. This data structure could be thought of as 'cognitive memory'. The memory of past actions and the consequences of the actions that have been previously remembered should be used to improve the network performance by allowing the network to make better-informed future decisions. This experience gained and referred to, should be used in order to increase the overall performance of the network and make the network more resilient to congestion and link breakages.

Should it transpire that the performance of the network degrades (i.e. throughput reduces, dropped packets increases, sent packets reduces or received packets reduces) it could mean that the resource(s) are not being used efficiently or the requirements of the application are not being met. A cognition cycle will often use the data that has already been gathered to perform some problem analysis in an attempt to determine the problem or the reason for the problem.

Once the problem or reason has been determined the network can begin to formulate a solution in order to better use the resources or in order to better meet the application requirements.

When the solution is reached, the resulting actions are planned, scheduled and executed.

The cognition cycle in this thesis is intended to be assigned to a management node within the data communication network and is depicted in Figure 12 - Cognition Cycle. The cognition cycle is formed using a controlled loop, which consists of four stages: Observe, Learn, Plan and Execute (see Figure 12 - Cognition Cycle).

A management node will assume the cognitive attributes. It is most desirable that the node(s) selected to become a manager node are in direct communications range of the source node and that it is does not form part of the route for that source node (i.e. it is not forwarding packets for the source node to the destination or to the next intermediate node that forms part of the route for the intended destination), this is desirable because the management role will assume additional processing and storage requirements. There are additional desirables in choosing the management node, such as, a good remaining battery lifetime or a longer duration that the management node will be in range of the source node. The additional desired parameters of a management node are discussed in further detail in the Election Process section.

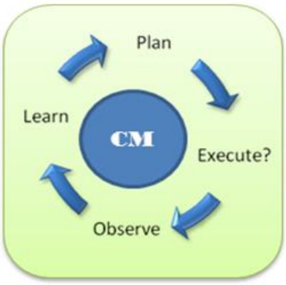


Figure 12 - Cognition Cycle

5.2 Topology

In order to perform a simulation the designer needs to produce a modelled world in order for the simulation to run. Many published research papers show that authors fail to provide justification of why the world that they designed was used. Many papers that show results based on simulations use arbitrary values, for example a geographical network area 400m by 400m or 160,000m^2 with no reasoning to why this area was chosen. Beraldi and Baloni (2003) state that their simulation consists of a geographical network area of 1500m by 1500m or 2,250,000m^2 and that each node within this region moves using a movement pattern based on the random waypoint movement model. Wang and Olariu (2004) state that their simulations consisted of nodes being distributed uniformly at random locations in an area which is either square or rectangle.

It can be argued that the testing of a protocol or network enhancement is generic and providing arbitrary values is adequate. Arbitrary values for the simulated world are also planned and random movements for each node are used. However, in addition to this models that attempt to model realistic scenarios with realistic objects, for example, a scenario that involves a search and rescue operation performed by military soldiers are created.

The results generated from the data communication network model being simulated could on one hand show that the said enhancement (for example that of a new or adapted routing protocol design) is more efficient than that of the original. However, a particular enhancement being tested against the original design in one scenario that provides positive results in regards to overall network performance may not show the same performance results in another scenario. Therefore, one should not assume that a newly designed protocol is better because it performed better in one test case.

This has been a potential limitation of the analysis when not using models with realistic world environments and when not running the adapted network design across several different simulation models that account for several scenarios.

Research conducted shows that the models that attempt to model the real world greatly determine the outcome of the results for the model being simulated, for example consider the work conducted by Pullin (2007) that tests the performance of three routing protocols. Pullin demonstrated that the DSDV routing protocol and the ZRP performed better in terms of the overall network performance over that of the more scalable AODV routing protocol; which has been highly tested and documented as a protocol that outperforms DSDV and ZRP in other simulations with different results. Pullin used a real world model and used real ship movement in the Irish Sea data for the MANET Evaluation.

Other real world simulations have been conducted by Kiwior and Lam (2007) where they describe the scenario that their simulation models are based on a representative laydown of 5 wide-body aircrafts and a land-based Tactical Operations Centre (TOC) which is applied to the Caspian Sea Scenario over an area of 750 n mi x 350 n mi. Each aircraft's flight path is specified by a centre.

Johansson et al (nd) ran simulations to test different routing protocol performance. Although these routing protocols have been tested many times before Johansson et al ran their model based on real-world scenarios. They describe a simulation model that is based on 50 people attending a conference, a scenario which models a group of 50 highly mobile people who frequently change position. The authors state that this could also represent other realistic scenarios such as a group of reporters that are covering a political event, a sport event or a stockbroker’s scenario whereby stockbrokers are in negotiation at a stock exchange.

Scenarios are also created which model real world scenarios and also consider the application of the network. More realistic non-arbitrary values are used in order to further test the effectiveness of the cognition applied to a MANET in regards to network performance.

As yet there is no consistent means of benchmarking the newly proposed protocols or enhancements to current network design for researchers in this field.

Before discussion of how to apply cognition to the network, the new structure the topology will form and the roles that the nodes will undertake is discussed, Figure 13 - A simple mobile ad-hoc network topology is used when explaining the nodes.

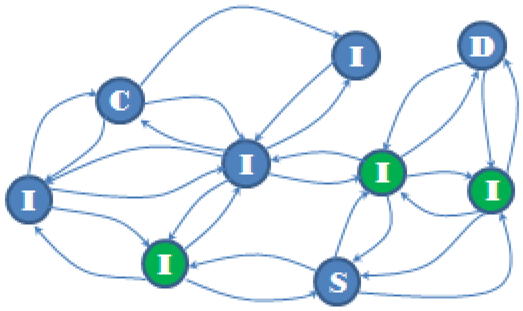


Figure 13 - A simple mobile ad-hoc network topology

A data communication network topology can be defined as the placement or arrangement of various hardware components that are used to construct the network, such as switches, nodes and the links between them (Groth and Skandier, 2005). Unlike traditional network topologies that are generally fixed, the mobility of the nodes participating in a MANET makes the network topology very dynamic and often unpredictable or difficult to predict. The mobility of the nodes is the primary reason for the dynamic topology but there are other attributes that factor to the dynamic nature of the topology such as battery life or communication range. Figure 13 - A simple mobile ad-hoc network topology depicts a snapshot of what a simple MANET topology may look like at a given moment in time. This snapshot is used subsequently to explain the fundamentals of MANET design and again when the elements that form the cognition cycle are discussed and how the cognition is integrated in to a node in order to perform some management of the network and the network element defined within the network.

5.2.1 Node Roles

The labelled circles S, D, I and C represent the participating nodes in the data communications network and the lines or links show the communication between the nodes, for example in Figure 13 - A simple mobile ad-hoc network topology the I nodes that are green are in direct communication range of the S node. Although all nodes participate in the network by forwarding data packets for the intended destination they could assume different roles. In Figure 13 - A simple mobile ad-hoc network topology S indicates the source node, D indicates the destination node, I indicates an intermediate node and C represents a controllable node (a controllable node is one that can be instructed to move to a new location in an attempt to form or strengthen communication links and is discussed in Section 5.2.1.4).

Section 5.2.1.1 Source Node

The source node is the node that has an application that generates data to be transmitted on to and traverse the data communications network for an intended destination. The destination is normally a single end point; however, there could be multiple destinations such as that used in multi-casting and broadcasting applications.

The data could potentially take many paths to arrive at the destination, although in this topology there are two paths from the source to the destination that consist of a single hop (or forwarded just once). Intermediate nodes could also become a source or a destination node for an application or mission should the need arise and is dependent on the scenario.

Section 5.2.1.2 Destination Node

The destination node is the receiver of the data and is sometimes referred to as a sink. The destination node passes the data received at the PHY layer to higher layers until it reaches the application layer. In order to know which application the data is to be passed to a unique local socket address is used. The local socket address is generated by the operating system when the application makes the request via the API. The local socket address is a combination of the local IP address and a port number.

Section 5.2.1.3 Intermediate Node

In Figure 13 - A simple mobile ad-hoc network topology the nodes in the network that are shown with the label (I) are intermediate nodes. An intermediate node acts as a router and is responsible for the forwarding of data that has been received to the destination, or to another indeterminate node that forms part of the path to the destination. This facilitates the communication between nodes that are not directly in wireless transmission range of each other.

The same mechanism is used for the Internet. However, unlike the Internet which uses dedicated and fixed routers, a MANET does not have dedicated or fixed routers. Thus each node participates in the network by undertaking the role of a router in addition to other roles it may have, i.e. a source node may also be an intermediate node to facilitate communication for another source node.

Section 5.2.1.4 Controllable Node

The node identified as (C) in Figure 13 - A simple mobile ad-hoc network topology, is a controllable mobile node. The controllable node would not normally be a source or destination node but a node placed into the network by a higher authority to assist the source node in regards to link breakages. A controllable node could be thought of as a robot or a drone. The primary reason for introducing the controllable node into the network is to facilitate the transfer of data within the network.

The placement of nodes in a MANET is vital for the formation of stable links. It is the aim of this research to present the feasibility of allowing idle nodes the ability to be instructed to move location to form reliable links. These types of nodes will be planted within the network for this purpose. It is envisaged that the nodes will be equipped with GPS and are aware of their position within the network; however, this research will also look at strategies whereby nodes without GPS can also be used.

Autonomous mobility of nodes has been made possible with the research into robotics, for example, ‘Big Dog’ developed by Boston Dynamics shows that advanced robots now have mobility, agility, dexterity and speed.

Figure 14 - Big Dog shows the ‘Big Dog’ robot and was taken from a video that demonstrated the characteristics of the robot. In the video the robot successfully negotiated snow, rough terrain and ice. This section of the video shows the agility of the robot as the robot successfully continued to walk after being kicked and stumbling on the ice.

It is beyond the scope of this thesis to detail robotics and their characteristics; however, in order to present a reasonable justification that proves that nodes can autonomously be moved to form reliable links and that the moving of controlled nodes is a viable solution.

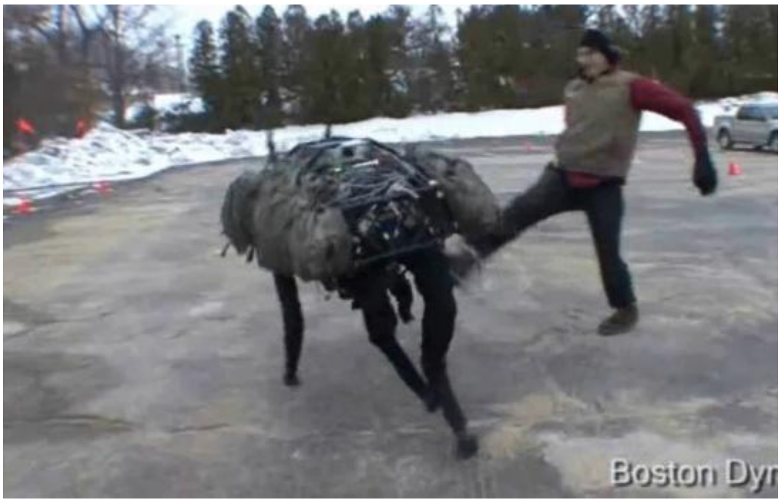


Figure 14 - Big Dog (Boston Dynamics, 2014)

The outputs are the main focus of this research. Alongside the outputs there also needs to be rules governing when the outputs are to be executed. The next section discusses the process for management node selection that will undertake the CM responsibility.

As part of the management the control mechanism has the ability to instruct nodes (controlled) to move from their current location to a new location. This is to facilitate stable links or create new links. This is to help with the traffic flow of the data.

The management node can also control the switching of channels or reallocate spectrum frequency for better utilisation of the bandwidth.

Finally the management node could instruct all nodes to switch from one routing protocol to another. The outputs are discussed in more detail after the Inputs section.

Combining these inputs as a functional control loop will allow the monitoring of the current state of the network, thus the network will be aware of its environment. Using these inputs and monitoring the network state over time will allow the management node the ability to predict changes in network state. This will facilitate the planning of changes in the network and produce various planning strategies based on the observation. It is envisaged that the outputs will facilitate on-the-fly changes in the operation or the infrastructure of the network.

The examples of the different applications that a MANET might be used for are discussed in chapter 2. One example was the implementation of a MANET for military purposes. In the military example a controllable node might be a node planted by a higher authority. The higher authority might be the person responsible for the control of the mission. There may be one, several or no controlled nodes depending on the application or scenario requirements.

Section 5.2.1.5 Management Node

The purpose of the management node is to learn in order to make better decisions to increase network performance. The management node is tasked with observing current network states and making comparisons with previous network states. The evaluation of current state against previous states are evaluated by the management node and based on this evaluation a management node might make a request.

One of the intermediate nodes within the communication range of a particular source node should be elected as a management (M) node. It is the management node that has the cognition applied to. The management node should process information, have the attention of the network state in its vicinity, have the ability to remember events that have occurred, and have the power to make judgements based on the evaluation of conditions and the reasoning of the current network state. The management node should also possess problem solving skills and decision-making. Thus the management node will have the implementation of the cognitive control loop.

The management node should listen to the local network state and the requirements of the application. Based on observations the management node should be able to make decisions on behalf of the source node. The role of the management node (intermediate node receiving management node status) is to receive information that relates to different resources and requirements. The management node also monitors the network and key resources in the data communication network.

Using the data gathered by monitoring the network, the management node should be able to plan and initiate changes in the network should the need arise. Any changes planned should be initiated depending on the requirements and the current network state. The observing, learning, planning and execution of the plan, are the stages of the cognition loop. This cognition loop results in the facilitation of the source nodes application requirements or mission objective.

5.2.2 Initial Placement

Now that the nodes and the roles of the nodes have been defined, discussion moves to detail how to position the nodes. Various placement models will be created to allow the flexibility of quickly testing other parameters within the network. The placement strategies allow the placement of nodes randomly within a predefined geographical area, place the nodes in a single location, place the nodes within a clustered group, place nodes within several clustered groups, allow the groups to adhere to fractional parts and finally an initial placement strategy that will allow the ability to place the nodes in a grid like formation.

5.2.3 Mobility

Various mobility models will be created that control how the nodes move within the world. Mobility models created will allow the testing of the parameters of the network, such as the transmission radius. In order to do this a mobility model that allows nodes to move along a predefined path (piece-wise linear motion) will be created. A realistic mobility model based on a realistic scenario will also be created. In order to compare the results of the simulation the mobility models that are used by other researchers in the field of MANETs will be used, therefore, the random waypoint model (RWP) will be implemented, which is a popular choice for many researchers and is well documented in much of the research literature available. The **R**andom **W**ay**P**oint Model (RWP) was initially proposed in 1996 (Johnson and Maltz, 1996).

There are tools available in most data communication network simulation software that will automatically create a movement model. In order to simulate a large number of nodes and have the nodes behave in an unpredictable manor a technique is used to generate the movement patterns for each node. According to Santi et al (2006) this model is commonly used in Ad-Hoc Network simulations to simulate random movement patterns for each node where nodes are allowed to move within a defined area. Each node is positioned randomly in the defined geographical area, it is then given a route line that is of random length, has a random angle and is of a random velocity (all within set boundaries). The end of the line signifies a way point and the process is repeated, thus creating another line, this is depicted in Figure 15 - Random Waypoint Model. This procedure is repeated until the node reaches its end position. This produces a zigzag route from the node start position to the node end position with way points along the way and is illustrated in Figure. At each way point the node will pause for a predetermined amount of time before proceeding to move to the next way point.

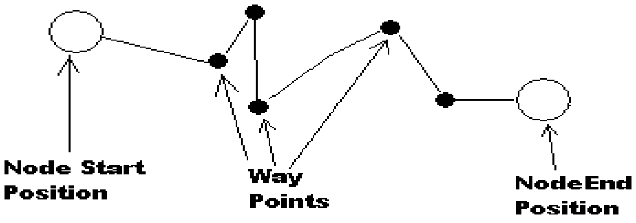


Figure 15 - Random Waypoint Model

There are some problems with the RWP model because of the paths that are constructed; it could be argued that the zigzag paths are unnatural and that the pause at each way point does not model realistic movement models. There are variations on the originally designed RWP such as the Markovian Waypoint Model (MWP), (Hyytia et al, 2006) which allows the next waypoint to be determined based on the current waypoint. The MWP also allows the designer the option to define the velocity distributions of each waypoint distance and to have random pause times at each waypoint.

In addition to this, a fine level of control is desired; changing the mobility of the nodes whilst the simulation is running is required, for instance, different velocities should be able to be incorporated. The aim is to create mobility models that are as realistic as possible so there will also be an attempt to apply slight variations between the nodes that intend to model real world objects such as people or soldiers, for example, not all soldiers walk at exactly the same speed, not all soldiers will walk the exact same path, not all soldiers will have the exact same stride size (step size) and so-forth. The variations are confined within boundaries, for example the speed of a walking solder will be approximately 1.4 meters per second, the variation therefore may be between 0.8 meters per second to 2.5 meters per second. It would not be realistic to apply arbitrary values as this could result in unrealistic walk speeds of say 25 meters per second.

In addition to this the smooth transitions for the models should be applied, thus avoiding discrete jumps from one point to another in the geographical area.

In respect to the controllable nodes, the aim to let the network dictate the mobility model for these nodes based on the requirements of the network. Rather than having the controllable nodes idle or stationary when they are not required, autonomous behaviour for each of the controlled nodes will be applied, this is so that they do not become too far separated from the other nodes in the network, for example, if the nodes in the network represented soldiers travelling South and the controllable nodes were stationary, the distance between the soldiers and the controllable nodes would increase as time passed. If at some point in the future a controllable node was requested to move to a new location to form or strengthen a communications link a stationary node would take longer than an autonomous node to get to the desired location if the autonomous node was positioning itself within the group of soldiers.

Similar to the other nodes, it is expected that the controlled nodes move in a realistic fashion, i.e. if the controlled node is modelling a person that is walking then they should move at realistic speeds with realistic transition.

5.2.4 Hardware

In order to maintain the realistic theme of the models the use of conventional hardware and technical standards will be used. The IEEE 802.11 protocol as discussed in Chapter 2 and the standards that this protocol offers will be adhered to, such as the maximum data rate of 11Mbps and the number of channels available. The operating frequencies of each channel will be in-line with the standards. The physical operation of the antennas will be based on the Friss propagation loss model with a propagation delay based on the constant speed propagation delay model. A non-quality of service Wi-Fi media access will be used which is typical for wireless networks. Of course, this will be set to Ad-Hoc mode.

The standard TCP/IP suite will be installed onto the protocol stack and use either the DSDV routing protocol or the AODV routing protocol. The network devices will communicate using the TCP/IP standards and various subnets to differentiate between different groupings of nodes will be used.

Already discussed are the mobility models and energy constraints; therefore, for each node an energy source that models a lithium battery will be installed. In addition to this correct operation (i.e. depletion) of the battery when a node is performing a given task (i.e. transmitting data) will be checked.

5.3 Cognitive Attributes

Already discussed is that methods in this network will comply with standards. Also presented are novel ideas that redefine the node roles within the networking topology which will be needed for the cognition cycle. Next, discussion continues by detailing how cognition is applied to the network configuration.

5.3.1 Cognitive Control

The management, regulation or control of the cognitive processes is known in psychology as cognitive control. Cognitive control represents the collective reasoning of observations, memory, attention, evaluation, reasoning and problem solving in order to meet a given task or goal. In the next sections the intention of how to implement the cognition cycle and how the cognition is used in order for the monitoring of network performance and make adjustments in the behaviour of the data communications network in response to external or internal network conditions is detailed.

The cognitive role is to be assigned to the management node(s), which is used to observe the current network state, predict future network states and control the resources in the network or how the nodes participating in the network use the resources.

In order to mimic cognition the management node will have the ability to capture information that is related to the current network state. The management node should have some form of memory in order to recall previous network states when evaluating the current network state.

The management node should be able to differentiate between attributes not related to the current network state and pay particular attention to the attributes that are associated with this network state.

After observing network state the management node should be able to formulate the reasoning of a particular network state and from this reasoning, if necessary, perform an action. That is, the management node should be able to perform some computation that allows a decided-upon solution to be reached. Then, the management node should be able to perform further reflection and form some judgement to how successful the solution was.

In order to select a management node an election process is performed to select the most suitable candidate to have the management responsibilities.

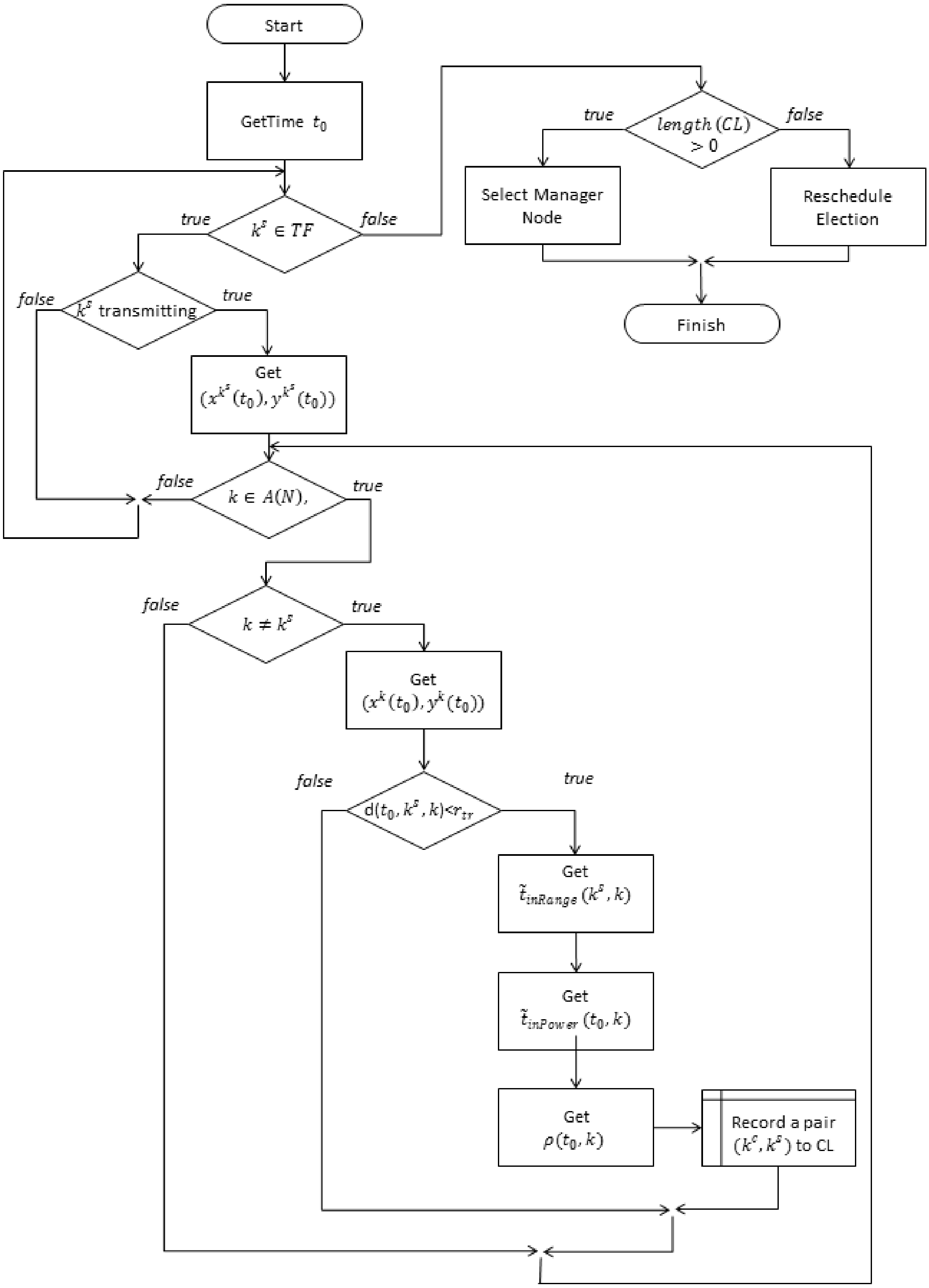
5.3.2 Candidate Node Selection

This section will detail the selection process for selecting suitable candidates to become a management (M) node. Consider Figure 13 - A simple mobile ad-hoc network topology, the figure shows that there are three nodes in direct communication range (neighbours) of the source node. These are coloured green. The election process will be initiated by each source node that intends to transmit application data. Several source nodes will not necessary indicate several managers because a manager node can manage multiple sources if required (if there is not a more suitable candidate).

Each candidate node on the network will have its own state based on the discussion earlier (i.e. energy level of the energy source, position and velocity). In this model a node cannot opt out of becoming a M node, therefore, all nodes within direct communication range of the source are potential candidates. Nodes outside of direct communication range are not considered because of the need for local proximity between source and management node for network states to be relevant. A network state in one area of the network does not mean a network state will be the same or similar in another area. The management node is making decisions on behalf of the source node based on current and past network states.

Figure 16 - Recording Candidates for Election depicts how a candidate list is built for all available sources on the network and this is explained subsequently.

Each source node will initiate a candidate selection process prior to beginning to transmit data. At this time the nodes that are designated source node(s) are checked. For each source node it is established if the node is still transmitting, if it is not the next node is checked and the candidate selection for this node conclude (i.e. not transmitting, no candidate selection required). If it is established that the source node is transmitting the current location (x and y position) of the source node is obtained. Then, for each node in the network distance is checked so that a list of direct neighbours can be built. For each direct neighbour of the source node, the predicted time in range between source and neighbour is obtained, as is the predicted life time of the neighbour and the current density of the neighbour. The current simulation time, the source node identifier, the neighbour identifier, the predicted time in range, the predicted time in power and the density is recorded. This neighbour is a potential management node and the values recorded will later be scored to establish which of the neighbours is the most suitable candidate to undertake the management role.



t_0 = time now
$rp_k(t_{n-1}, t_n)$= packets received by node k during time interval $[t_{n-1}, t_n]$
$\rho(k, t)$ = density of network for node k at time t.
TF is the set of source nodes' IDs involved in traffic flows
$k^s \in TF$ is the current source node ID
CL =candidate list; the set contains the pairs of IDs of candidates k^c and source k^s nodes

Figure 16 - Recording Candidates for Election

Once the possible candidates are selected, the process of dwindling the candidates will begin by checking their current states and their predicted future states. For example, a node that is closest to the source node should not be chosen with the assumption that this node will be in range longer than the other candidates because each node has its own velocity. However, the distance is considered to be one of the most important parameters when choosing the most suitable candidate node. This is because network state plays a key part in the decision making process of a management node, therefore local proximity should be maintained as long as possible, otherwise, the election process starts over.

For each candidate node, it will be predicted how long that node will be in range of the source node, depending on the source node and each of the candidate nodes positions and velocities. It would not be prudent for a node to be elected if the node cannot maintain its link for a period of time.

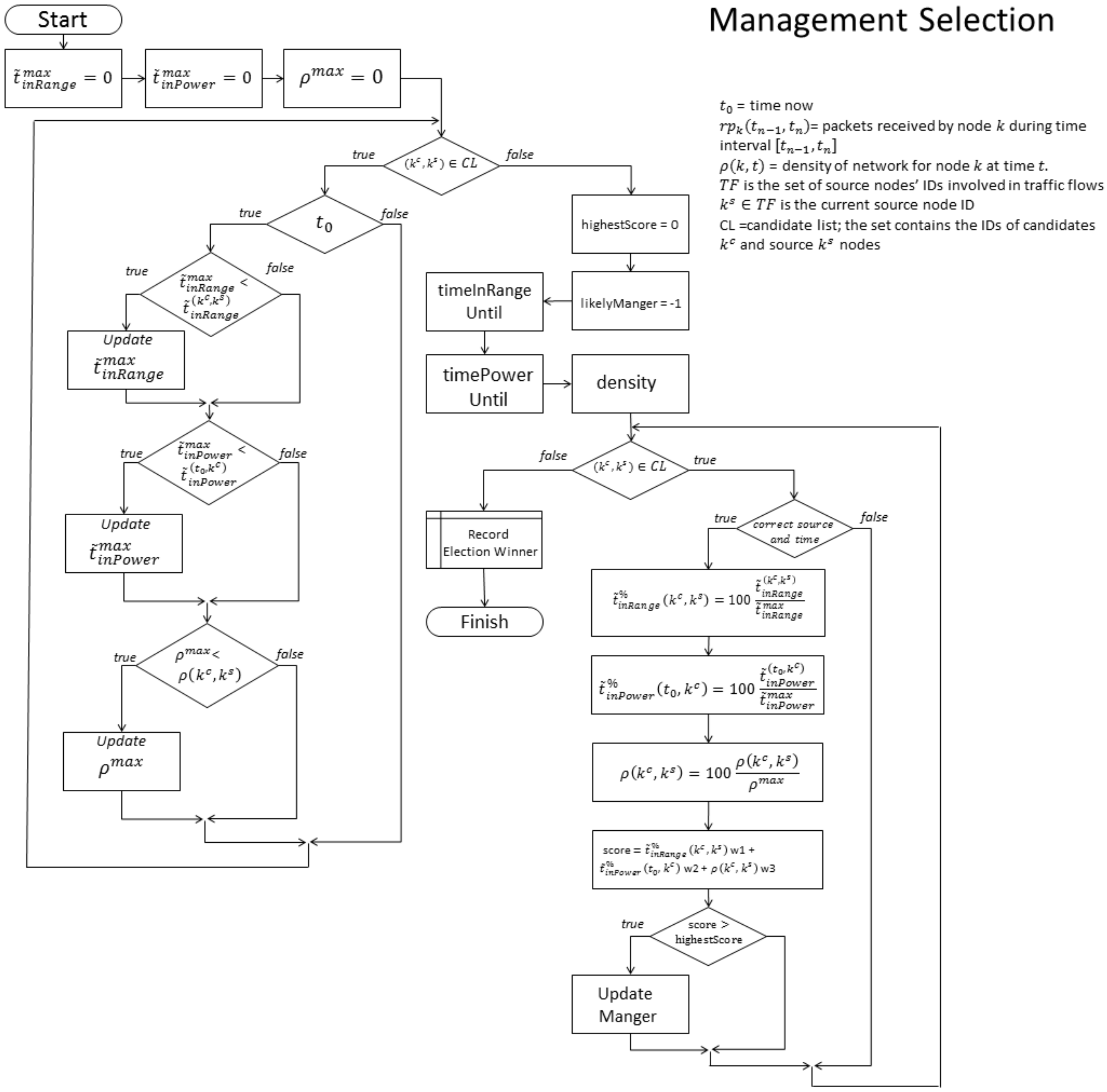
Next it would be unwise to choose a node that has limited power remaining, the current energy level for each candidate will be obtained. Similar to distance one cannot assume that the node with the highest energy level will be the node that is alive (in respect to power) the longest and this is determined by the current draw of power from the energy source. Therefore, the prediction of how long it is expected that the node will have suitable energy levels for by using previous knowledge will be undertaken.

The third consideration is the current density of the local network for each candidate; nodes of higher density will likely be receiving more packets (control and application), but use less transmission power.

5.3.3 Management Node Selection

Each of the considerations will carry a weighting that determine the importance of that consideration. This allows for the more critical constraints to be given greater priority when deciding on the most suitable candidate in the election.

Other considerations could have been also taken into account, for example processing power. However, this will prolong the election process and my cause unfair elections, and it would be unfair for a particular node to be constantly elected because its processing capabilities are greater. It would be unfair because being continually elected based on the processing power would have adverse effects on the life-time of that node because the battery levels would reduce with the additional processing. Figure 17 - Management Node Selection depicts how this could be achieved.


Management Selection
Start
$\tilde{t}^{max}_{inRange} = 0$
$\tilde{t}^{max}_{inPower} = 0$
$\rho^{max} = 0$
true
$(k^c, k^s) \in CL$
false
t_0
$\tilde{t}^{max}_{inRange} < \tilde{t}^{(k^c,k^s)}_{inRange}$
Update $\tilde{t}^{max}_{inRange}$
$\tilde{t}^{max}_{inPower} < \tilde{t}^{(t_0,k^c)}_{inPower}$
Update $\tilde{t}^{max}_{inPower}$
$\rho^{max} < \rho(k^c, k^s)$
Update ρ^{max}
highestScore = 0
likelyManger = -1
timeInRange Until
timePower Until
density
Record Election Winner
Finish
correct source and time
$\tilde{t}^{\%}_{inRange}(k^c, k^s) = 100\,\frac{\tilde{t}^{(k^c,k^s)}_{inRange}}{\tilde{t}^{max}_{inRange}}$
$\tilde{t}^{\%}_{inPower}(t_0, k^c) = 100\,\frac{\tilde{t}^{(t_0,k^c)}_{inPower}}{\tilde{t}^{max}_{inPower}}$
$\rho(k^c, k^s) = 100\,\frac{\rho(k^c, k^s)}{\rho^{max}}$
score = $\tilde{t}^{\%}_{inRange}(k^c, k^s)$ w1 + $\tilde{t}^{\%}_{inPower}(t_0, k^c)$ w2 + $\rho(k^c, k^s)$ w3
score > highestScore
Update Manger

t_0 = time now
$rp_k(t_{n-1}, t_n)$= packets received by node k during time interval $[t_{n-1}, t_n]$
$\rho(k, t)$ = density of network for node k at time t.
TF is the set of source nodes' IDs involved in traffic flows
$k^s \in TF$ is the current source node ID
CL =candidate list; the set contains the IDs of candidates k^c and source k^s nodes

Figure 17 - Management Node Selection

Figure 17 - Management Node Selection shows the initialisation of some variables to zero that will store the predicted time in range, the predicted time in power and the maximum density. All the possible candidates in the candidates list will be checked ($(k^c, k^s) \in CL$, here (k^c, k^s) is the pair of nodes' IDs of the candidate node and the source node, stored in the candidate list CL). Each candidate is checked that it is candidate for the current source of interest; each source will have a manager. The next checks are the predicted maximal

time in range, the predicted maximal time in power, and the density for the candidate being checked. If any of those values are higher than the current values stored in variables that were initialised the variables will be updated to reflect a new maximum value found. This process is repeated for all the possible candidates for a given source.

Once the highest values have been established, more variables are initialised to be used in the selection process. Again, each candidate in the list is checked and percentage values based on the maximal values previously obtained are calculated. Once the values have been calculated as a percentage each candidate is scored based on a weighted system. The candidate with the highest score becomes the manager.

5.3.4 Relationship Status

Quasiperiodically the Manager node will check its relationship state with the source node(s) that it is managing in order to assess the viability of the relationship. First the Manager node should check that the source node that it is managing is still transmitting data (i.e. it is still a source node). Similar steps are performed as in the election process to ensure that the current manager node is still a viable management node based on the current state of the source node and this manager node. This is done to try to preserve any knowledge gained during the period of management. This is depicted in Figure 18 - Overview of Management Role.

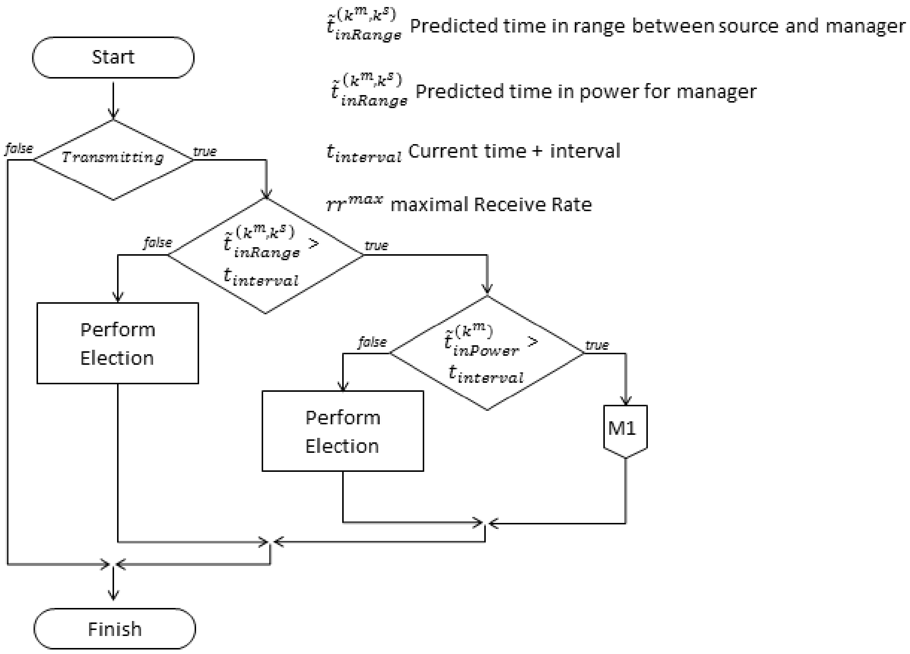

$\tilde{t}^{(k^m,k^s)}_{inRange}$ Predicted time in range between source and manager
$\tilde{t}^{(k^m,k^s)}_{inRange}$ Predicted time in power for manager
$t_{interval}$ Current time + interval
rr^{max} maximal Receive Rate
Start
Transmitting
false
true
$\tilde{t}^{(k^m,k^s)}_{inRange} > t_{interval}$
false
true
Perform Election
$\tilde{t}^{(k^m)}_{inPower} > t_{interval}$
false
true
Perform Election
M1
Finish

Figure 18 - Overview of Management Role

5.3.5 Network State

Once a manager node has been selected or a previous manager's relationship state is confirmed as viable, it should have the authority to make recommendations that will improve network performance should the performance of the network degrade. The manager node should be able to observe the network conditions in order to make this assertion. The management node will have the ability to remember previous network states, ascertain the current network state (rate of dropped, received, sent and forwarded packets) and predict future network states. This is depicted in Figure 19 - Current Network State.

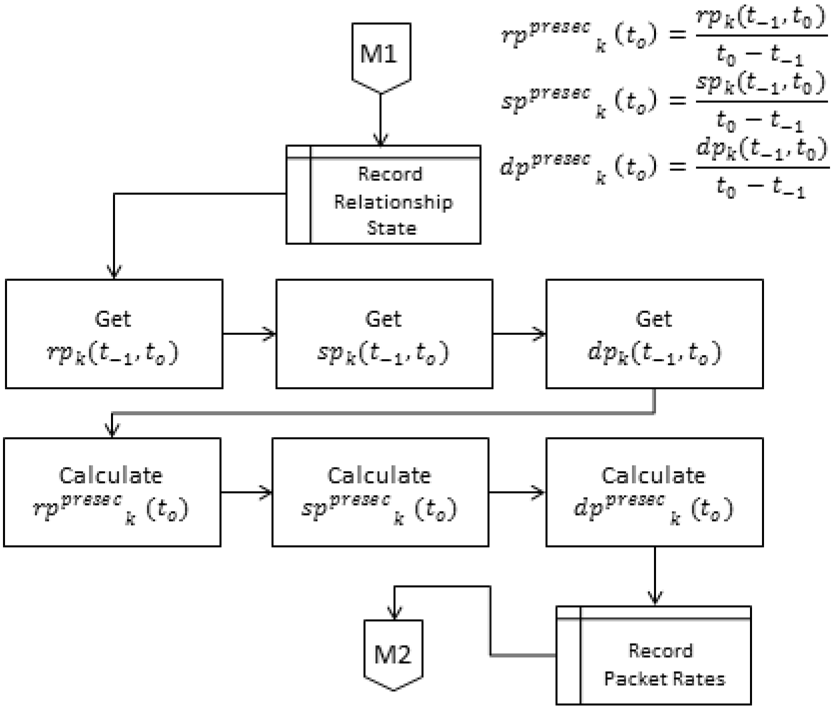

$rp^{presec}{}_k(t_o) = \frac{rp_k(t_{-1}, t_0)}{t_0 - t_{-1}}$
$sp^{presec}{}_k(t_o) = \frac{sp_k(t_{-1}, t_0)}{t_0 - t_{-1}}$
$dp^{presec}{}_k(t_o) = \frac{dp_k(t_{-1}, t_0)}{t_0 - t_{-1}}$
M1
Record Relationship State
Get $rp_k(t_{-1}, t_o)$
Get $sp_k(t_{-1}, t_o)$
Get $dp_k(t_{-1}, t_o)$
Calculate $rp^{presec}{}_k(t_o)$
Calculate $sp^{presec}{}_k(t_o)$
Calculate $dp^{presec}{}_k(t_o)$
M2
Record Packet Rates

Figure 19 - Current Network State

5.4 Memory

Memory can be defined as the ability to encode, store, retain and recall information and past experiences from a storage device. In humans the brain is the storage device for memory and is said to be a hugely complex organ and is the centre for higher-order thinking.

Computer programs also are able to encode, store, retain and recall information, a simple example might be a variable declaration in a programming language that can be used to store and retain values in the short term, computer memory is volatile and the contents are lost when the computer is restarted or powered down. The values can be accessed and updated by using the variable reference. It is envisaged that the use of arrays will be used to store memories of network states, decisions made and the outcome of the decisions for each manager node.

This will allow the management node the ability to encode (take the information about the network state and present in a suitable form for storing), store the memory of the network state and then retain and recall on the memory stored to evaluate past experiences.

5.4.1 Attention

In psychology attention can be defined as how one actively process specific information in their environment and ignore other information that is not important or relevant to the current situation. In this case, attention will be addressed by only encoding key information required by the management node and ignoring other information. For example, if the network is concerned with the receive rate of data packets that have been sent, it might ignore how many times a packet has been forwarded.

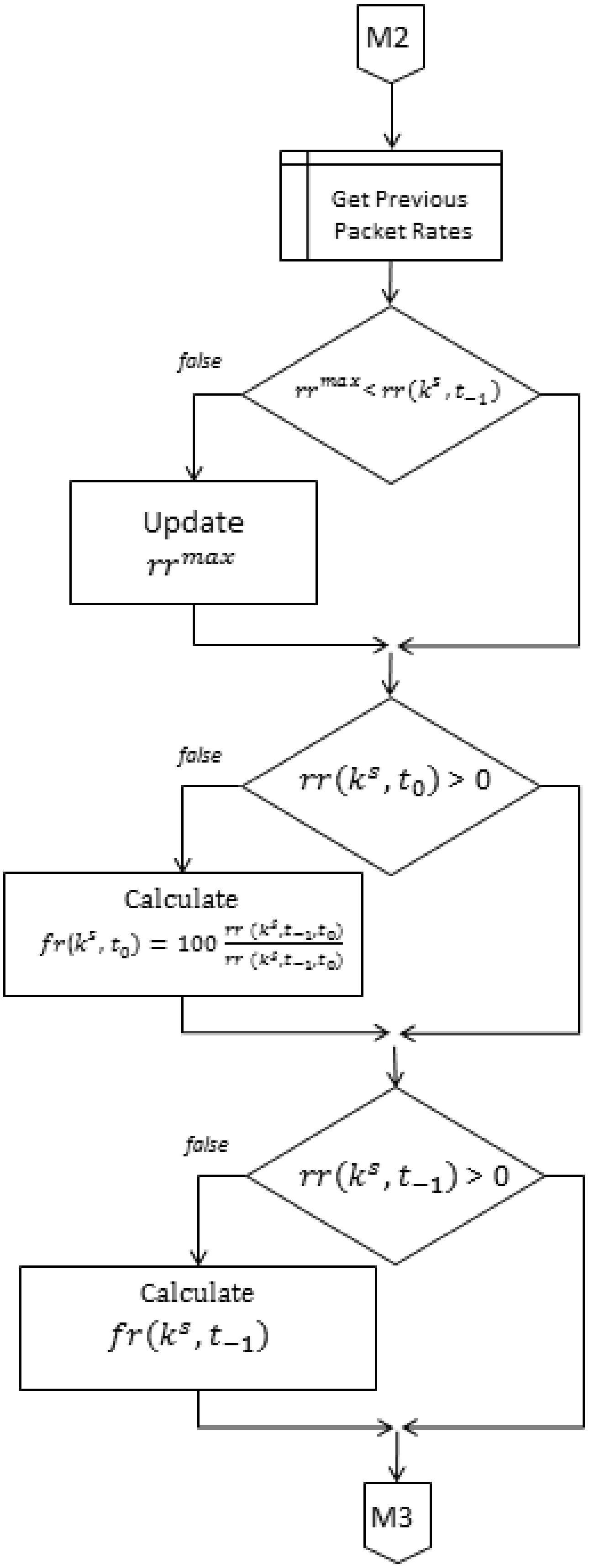

M2
Get Previous Packet Rates
false
$rr^{max} < rr(k^s, t_{-1})$
Update rr^{max}
false
$rr(k^s, t_0) > 0$
Calculate
$fr(k^s, t_0) = 100\frac{rr\ (k^s, t_{-1}, t_0)}{rr\ (k^s, t_{-1}, t_0)}$
false
$rr(k^s, t_{-1}) > 0$
Calculate $fr(k^s, t_{-1})$
M3

Figure 20 - Memory

5.4.2 Processing Information

According to psychologists in order for the brain to processes information it first needs to be stored and they give examples of types of memory such as sensory, short term and long term. This research models short-term and long-term memory. Each node is aware of its immediate vicinity (nodes in transmission range, local density, number of packets being sent); this information is the network state and can be perceived as short term memory. In order to establish if the network is performing poorly or to establish if the performance of the network is degrading the node should have access to previous states that have been captured (long term memories). Based on the knowledge just obtained (i.e. the current network state) and the previous network states stored in memory the node can evaluate consequences of past experiences. The proposed design is reflected in Figure 21 - Management Roles, which depicts the process of capturing and storing information.

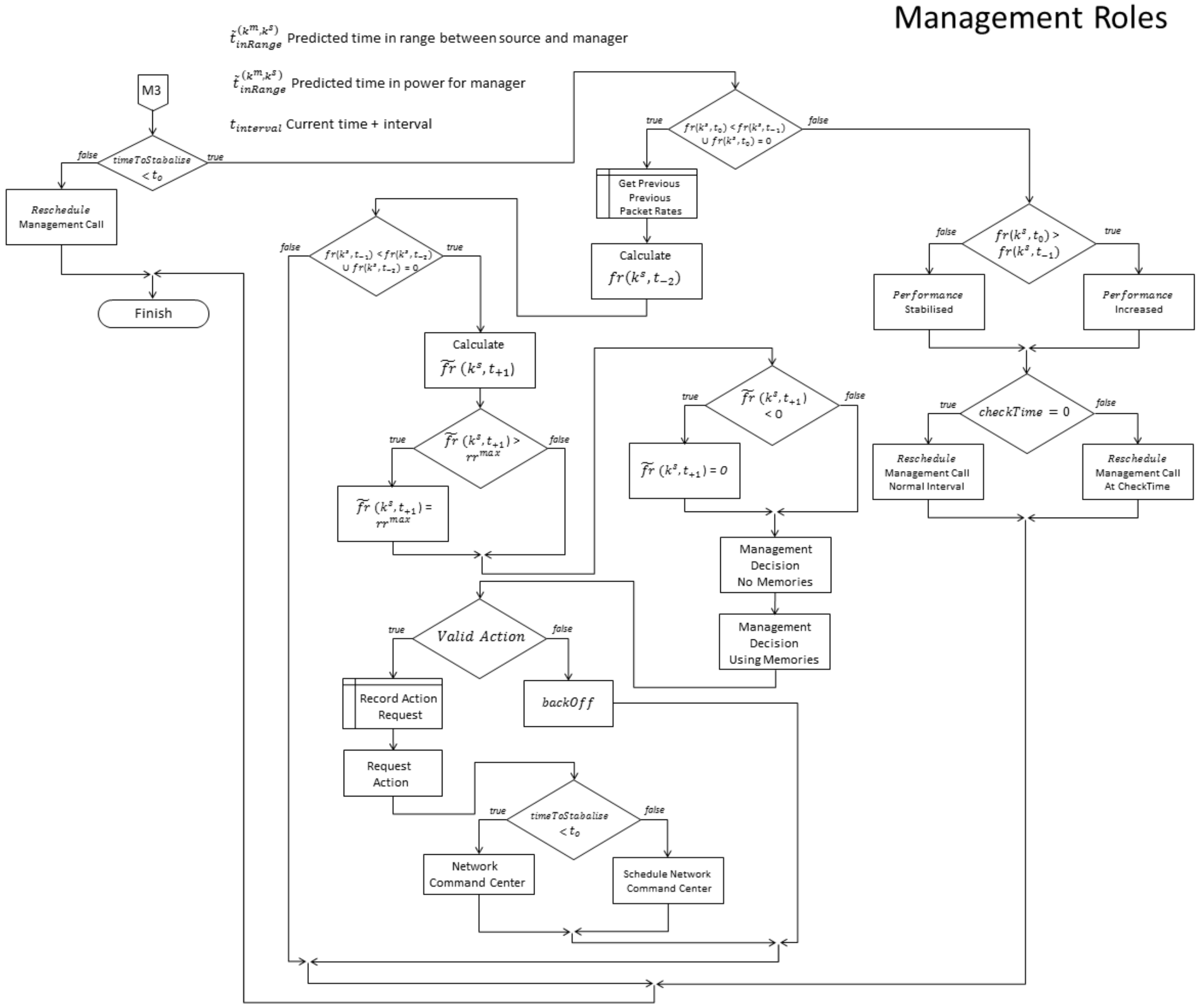

Management Roles
$\tilde{t}_{inRange}^{(k^m,k^s)}$ Predicted time in range between source and manager
$\tilde{t}_{inRange}^{(k^m,k^s)}$ Predicted time in power for manager
$t_{interval}$ Current time + interval
M3
$timeToStabalise < t_0$
false
true
Reschedule Management Call
Finish
$fr(k^s,t_0) < fr(k^s,t_{-1}) \cup fr(k^s,t_0) = 0$
Get Previous Previous Packet Rates
Calculate $fr(k^s,t_{-2})$
$fr(k^s,t_{-1}) < fr(k^s,t_{-2}) \cup fr(k^s,t_{-2}) = 0$
Calculate $\widetilde{fr}\,(k^s,t_{+1})$
$\widetilde{fr}\,(k^s,t_{+1}) > rr^{max}$
$\widetilde{fr}\,(k^s,t_{+1}) = rr^{max}$
$\widetilde{fr}\,(k^s,t_{+1}) < 0$
$\widetilde{fr}\,(k^s,t_{+1}) = 0$
Management Decision No Memories
Management Decision Using Memories
$fr(k^s,t_0) > fr(k^s,t_{-1})$
Performance Stabilised
Performance Increased
$checkTime = 0$
Reschedule Management Call Normal Interval
Reschedule Management Call At CheckTime
Valid Action
Record Action Request
backOff
Request Action
$timeToStabalise < t_0$
Network Command Center
Schedule Network Command Center

Figure 21 - Management Roles

The design is a continuation of Figure 20 – Memory, Figure 21 – Management Roles begins with the off-page connecter (pennant) labelled M3. Initially the management node will check to ensure that there has been enough time for the network to stabilise after a prior action has been performed; if not then the management node reschedules its evaluation for a future time based on the time to stabilise value.

If the network has had time to stabilise since the last action, or if there was no previous action the management node will check to see if the current receive rate is less than the previous receive rate or if the current receive rate is equal to zero.

When the current receive rate is not less than the previous receive rate there has either been an increase in network performance or the current receive rate is the same of as the previous receive rate. The node will establish if performance has stabilised or increased by using the network state recorded before the previous network state was recorded. Based on this information the node can establish if the network has stabilised or if performance has increased. Using this information the node will adjust the time at which next to call the management method (when to perform the next check on behalf of the source node). If performance increases the time at which next to call the management method will be increased, if performance has stabilised the time next to call the management method will remain unchanged.

Should it transpire that the current receive rate is less than the previous receive rate then there has been a drop in performance for the source node being managed. This may be a momentarily drop, therefore, the management node will determine this by checking the receive rate prior to the previous receive rate. If the previous receive rate is greater than the receive rate prior to the previous receive rate then no actions are considered at this time, however, the management node will reduce the time that the management method is next called.

In situations where there is a trend of performance degradation the management node will calculate the predicted receive rate if the trend continues, this is achieved by using the least squares approximation algorithm. The predicted receive rate is used when making a request for a change in the network. Based on the information gathered the management node will attempt to formulate a solution in order to increase network performance and is discussed in detail in section 5.6.4 Similar States.

If there are no valid solutions or the network is unable to accommodate a particular request the management node will be instructed not to make the same request again until a given time period has elapsed. The period of time is based on the action denied. When a valid course of action has been established the management node will record the current network state to memory and set a flag to show that the action is pending. The request is then put forward to the Network Command Center (discussed in section 5.7.1 Network Command Center).

5.5 Information Capture

This section details the information that is aimed to be captured with proposed ideas on how this information can be captured.

5.5.1 Position Information

One of the characteristics of a MANET is the mobility of the nodes within the network. As mentioned earlier it is envisaged that the management role is to be undertaken by a neighbouring node of the source node. Therefore, a mechanism is needed in order to get the position of any node that is of interest. In this case each source node and the candidate(s) for each source. Considering that the nodes in this network environment are mobile, the continuing monitoring of the positions of the nodes would be very advantages.

5.5.2 Distance Information

The distance between a source node and a management node should be known. If the distance becomes so great it could disrupt the ability of communication between the source node and the management node, and vice versa. If the signal strength is not strong enough to be interpreted by the receiver, communication between the nodes is lost.

5.5.3 Velocity Information

It is also desirable to know which direction the nodes are travelling and at what speed. If the source node and the management node are moving in opposite directions signal loss will occur sooner than if they were travelling on a similar trajectory. The time signal loss occurs is dependent on both of the current positions and the velocities of each of the nodes.

It is the aim to capture information relating to mobility for several reasons; one reason is for the election process. In the event that the management node begins to move away from the source node, or the source node begins to move away from the management node, there should be a mechanism in place where the management node can relinquish responsibility. This is important as over time, the management node will have obtained knowledge that is potentially transferable.

Should communication between the source node and the management node cease, this knowledge could be lost and will require relearning by the newly elected management node assigned.

5.5.4 Packets

The objective of any data communication network is to send packets from a source node to a destination node. Packets within a network can be sent, received, forwarded and dropped. An attempt to record all packet information in the network and use the packet information as metrics in determining if the network is performing well or not will be necessary.

This research is not evaluating routing protocol performance per se, so control packets are less of an interest in establishing how the network is performing. However, choosing the most appropriate routing protocol for a given scenario is of much interest, therefore, particular concentration will be on delivery ratios of application data rather than control traffic such as route requests or route replies.

It is desirable to record the number of packets sent and received for each node. Also considered will be the recording of the number of packets dropped.

5.6 Information Processing

The information gathered will be used at various times during the lifetime of the network. Already discussed is how the information may be used for the election process and also when checking the relationship status for the manager and source nodes. Next, consideration is given to how else this information might be used and focuses on the cognitive attributes.

5.6.1 Management Node

The management node will be responsible for the collecting of data as discussed above. The data will be carefully encoded in order to use as little resource as possible for the management nodes. However, memory size on mobile devices in recent years is vastly greater than prior. With SD cards available that store 512GB of data and transfer up to speeds of 95MB/s, which is 1,000 fold over the storage resources available in the last 10 years (Lee, 2014), the initial concerns of very limited memory capacity have been alleviated, however, the aim is to use as little of this resource as possible.

5.6.2 Infancy Period

In the introduction the Infancy Period of a management node was briefly mentioned, i.e. a node that has been newly elected and has no prior experience. If this is the case the Management node will monitor the localised state (within transmission range) of the network at periodic times (i.e. every 2 seconds). The manager will begin to build up a memory of states for those periods based on the information discussed before (i.e. number of packets sent from the source node). In addition to this learning stage the Management node will also be observing the performance of the network. It will still be allowed for a Management node to have the responsibility of attempting to increase network performance if network performance is dropping. This may have a detrimental impact on the performance of the network, however, it is believed that the network will learn from its mistakes and make better informed decisions in future Learning from ones' own mistakes is not a new idea and is well documented (Buckley, 2009) and (Duffy and Saull, 2008).

A manager with no prior experience shall base it decisions on packet ratio delivery between source and destination. In addition to this, local node density is taken into account by calculating a density reference for the manager node. In order for a Manager to make a decision during the infancy period, default decisions will be used.

If the Management node observes a drop in performance the Management node will initially assume that the right course of action is to switch frequency. This is an initial default course of action and has been chosen arbitrarily. The default course of action may be changed and is determined by received packet rates up to three intervals in time.

For example, consider three time intervals in seconds: 300s, 298s and 294s, if at each of these intervals the received rate for packets being sent is zero it is clear that at the last three periodic checks that there was no communication at all between the source node and the destination node. If this is the case it may be likely that a link breakage has occurred, therefore the manager will change from the default decision of switching channel to that of moving a controllable node. This is illustrated in Figure 22 - Infancy Period Decision 1.

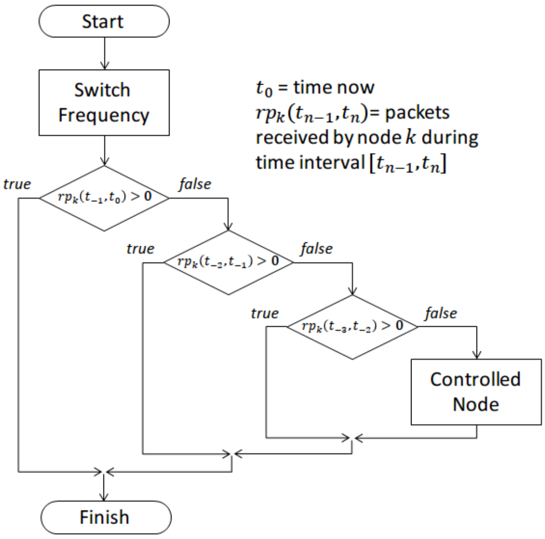

Start
Switch Frequency
t_0 = time now
$rp_k(t_{n-1},t_n)$= packets received by node k during time interval $[t_{n-1},t_n]$
true
$rp_k(t_{-1},t_0) > 0$
false
true
$rp_k(t_{-2},t_{-1}) > 0$
false
true
$rp_k(t_{-3},t_{-2}) > 0$
false
Controlled Node
Finish

Figure 22 - Infancy Period Decision 1

If there is a current flow rate between source and destination but the flow rate has decreased since the last check it could be attributed to the fact that the better suited protocol is not being used for this network state. To further reinforce this decision the density of the network local for the current node is obtained and if the density is low the action is changed to switch protocol.

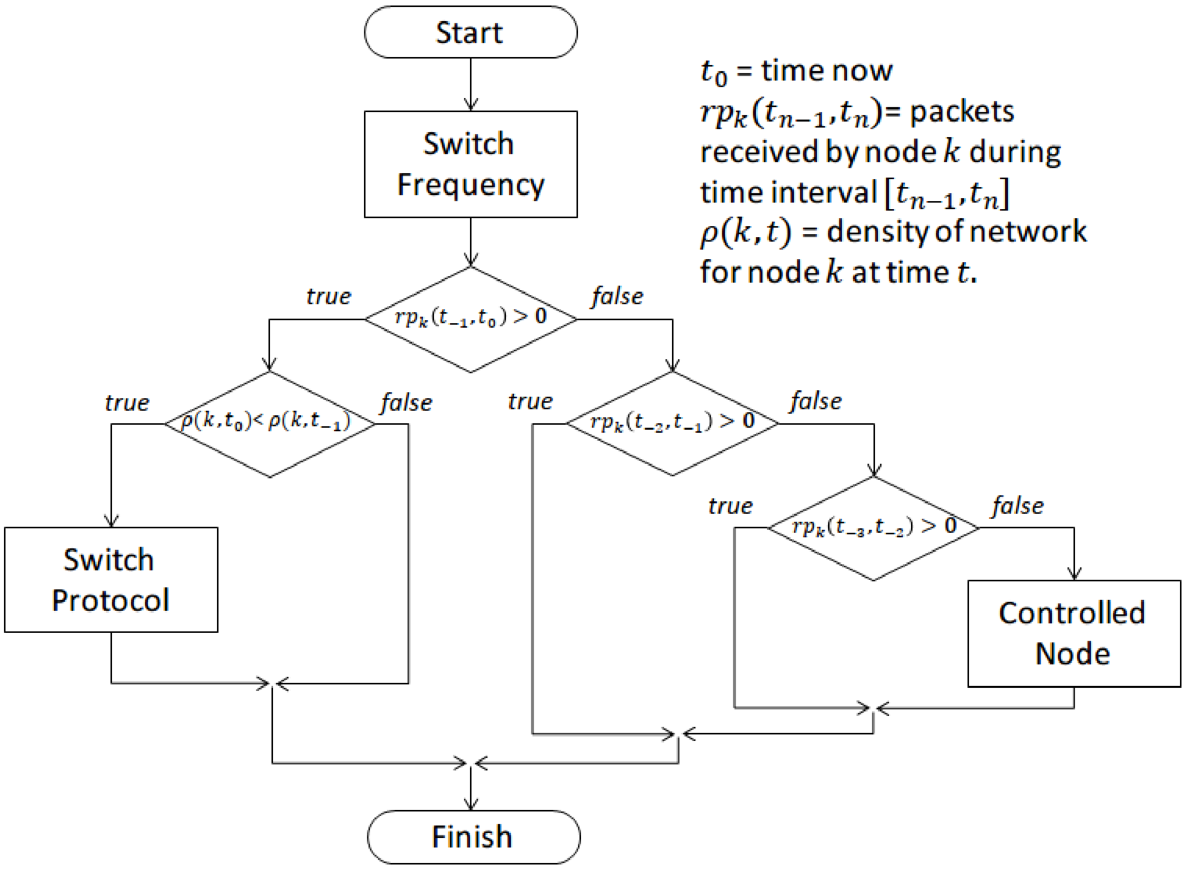

Start
Switch Frequency
t_0 = time now
$rp_k(t_{n-1},t_n)$= packets received by node k during time interval $[t_{n-1},t_n]$
$\rho(k,t)$ = density of network for node k at time t.
true
$rp_k(t_{-1},t_0) > 0$
false
true
$\rho(k,t_0)< \rho(k,t_{-1})$
false
true
$rp_k(t_{-2},t_{-1}) > 0$
false
true
$rp_k(t_{-3},t_{-2}) > 0$
false
Switch Protocol
Controlled Node
Finish

Figure 23 - Infancy Period Decision 2

Therefore based on the current state of the network and based on the limited knowledge (or lack of knowledge) the network attempts to choose a suitable course of action. Unfortunately, because of the lack of knowledge or limited knowledge this course of action may not be right. However, this does give the management node the opportunity to learn from its mistakes.

Each decision made by the management node should be remembered with the context for making that decision (i.e. the network state). After some time has elapsed, the management node should then reflect and determine if the course of action take for that given decision was appropriate.

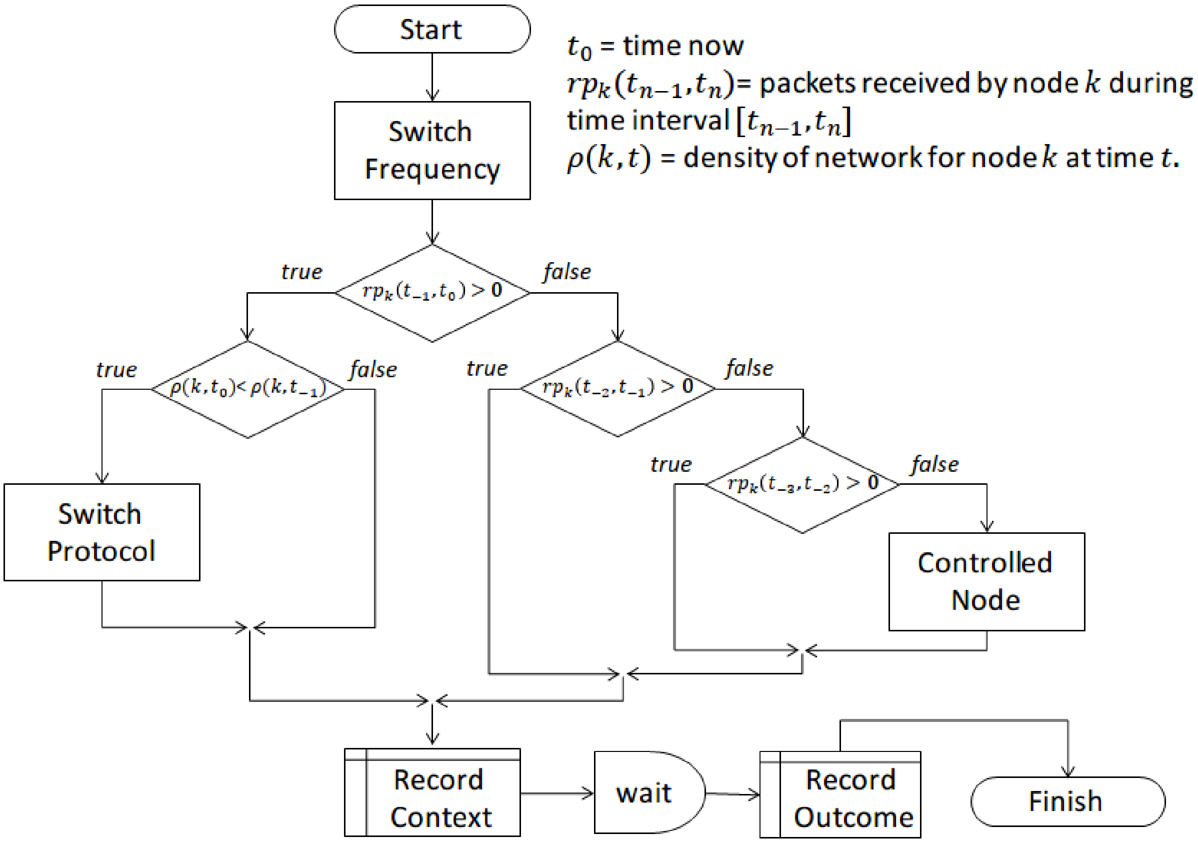

Start
t_0 = time now
$rp_k(t_{n-1},t_n)$= packets received by node k during time interval $[t_{n-1},t_n]$
$\rho(k,t)$ = density of network for node k at time t.
Switch Frequency
true
$rp_k(t_{-1},t_0) > 0$
false
true
$\rho(k,t_0)< \rho(k,t_{-1})$
false
true
$rp_k(t_{-2},t_{-1}) > 0$
false
true
$rp_k(t_{-3},t_{-2}) > 0$
false
Switch Protocol
Controlled Node
Record Context
wait
Record Outcome
Finish

Figure 24 - Infancy Period

As the management node becomes more experienced the manager node should not perform actions entirely based on intuition but instead, should recall memories of past experiences. Therefore the simple thought processes depicted in Figure 24 - Infancy Period will become much more complex.

5.6.3 Committing to Memory

Initial consideration was given in regards to the amount of memories that should be allowed to be accumulated by the management node. Earlier in this chapter it was stated that digital memory was becoming less of a concern because of new advances in technology, in addition to this, the human brain although must have a physical limit, is unlikely to run out of space in our lifetime (Reber, 2010). However, it may still be useful to limit the amount of information gathered by the management node in order not to overburden the mobile network device. Therefore, to avoid overburdening the node, a feature will be implemented so that a total number of entries (a limit) can be set for that node. This will allow limited memory capture / recall (i.e. most recent events) and full memory capture / recall (i.e. all events).

An overview of the information that is intended to be captured by the network has been given. In the case of management nodes, the information that is to be captured is the number of packets dropped that are sent by the source node it is managing, the number of packets that have been received from the source node and the number of packets sent by the source node. The management node should also record the current channel that it is operating on, the current routing protocol in use and the local node density. The current operating channel and routing protocol are known to the node; however, the node density will be calculated to give a density reference for the manager node by dividing the number of nodes within transmission range by the transmission radius of the management node.

5.6.4 Similar States

It might be unlikely that the same state will be identical to that of a previous state. As just mentioned the state will be determined by the number of dropped, received and sent packets, local node density, routing protocol and operating channel. Therefore, a way in which the manager can reflect on similar states will need to be developed.

It should also be accounted that some of the attributes captured being more or less important for a given state, for example, a high number of sent packets does not necessary mean that the packets are being received. Therefore, it is assumed that the number of received packets is of greater importance than the number of sent packets.

A weighting system for the attributes will be applied that relate to the packets. Using the weights with the number of packets for a given duration allows for a scoring scheme to establish how similar the state is in an attempt to choose a previous state that is similar to the current state rather than identical.

The management node should be able to choose a course of action that had a positive impact on the network in regards to performance from a previous state that is similar to the current state and dismiss other courses of action that show less similarity or if the previous course of action had a detrimental effect on the performance of the network.

Based on this knowledge the course of action will be updated from the initial intuition to a better informed course of action by using memory of past events and previous knowledge.

In addition to this there should be some mechanism of when the management node can evaluate current network state and make decisions based on prior state. If the manager is constantly monitoring the network the lifetime of the node would be depleted sooner. In addition to this, often mobile devices have a primary application(s) (i.e. a mobile phone) and it would not be desirable to tie-up the resources of the device at the expense of the device not being able to perform its primary application(s). It is intended to implement a scheme as depicted in Figure 25 - Decisions from Experience of Similar States and Figure 26 - Decisions from Experience of Similar States (b).

When the management node has determined that performance has decreased, an attempt to formulate a plan to prevent a further drop in performance will start. The management node initialises counters to keep track of the number of memory entries considered and the number of actions considered. Whilst there are memory entries to consider, and whilst an action is available the management node will check each memory entry.

backOffTime > t_o

Each memory entry will have a back-off time associated with it. The back-off time is calculated when a request is considered by the Network Command Center (Section 5.7.1 Network Command Center). The back-off time prevents a manager requesting the same course of action too soon, for example, if a manager node requested the use of a controlled node at 30 seconds but the node was unavailable because it is busy completing a prior request and would not complete this request for another 26 seconds then the back-off time for that request would be set to 56 seconds. Each time a back-off time is encountered which is less than the current time 1 is added to the strike counter. If the strike counter reaches three then all available actions are exhausted and no action will be requested by the manager node at this time, in this case the manager node will not call the management method again until time reaches or passes the minimum back off time, for example consider the following situation at time 30 seconds:

Requested Action	Back-Off time
Change Routing Protocol	38 seconds
Change Frequency	35 seconds
Move Controlled Node	56 seconds

In this case the management node would not call the management method until 5 seconds had elapsed. A call to the management method prior to the elapsed 5 seconds would result in three strikes and so would be pointless.

For each back-off time encountered a flag is also set to false for that action, for example if the back-off time for this current memory was 38 seconds and the action was to change routing protocol a flag for switch routing protocol allowed (spa) would be set to false.

backOffTime < t_o

When a memory is encountered that has a back-off time which is less than the current time the management node is allowed to consider this as a course of action.

performance < 0

The first consideration is if the action of the memory recalled did increase performance. Therefore, for each memory there is a performance metric to indicate if performance increased after implementing this action. If performance did not increase it may not be viable to follow the same course of action. Each action encountered that did not show any performance increases are summed up.

performance > 0

When a memory entry is encountered that has both a back-off time which is less than the current time and shows that performance increased the distance is calculated based on the state of the network at that time. Only distances for allowed actions are calculated and flags are used to keep track of which actions are not allowed and which actions are allowed, for example, a flag for switch routing protocol allowed (spa) is used to keep track of if switching protocol is allowed or not. Similar flags are used for move controlled node and switch frequency.

Once the distance has been calculated it is compared to the minimum distance already recorded. If this distance is less than the minimum distance already recorded (or it is the first distance calculated) the minimum distance will be updated and the requested action by the management node will be updated.

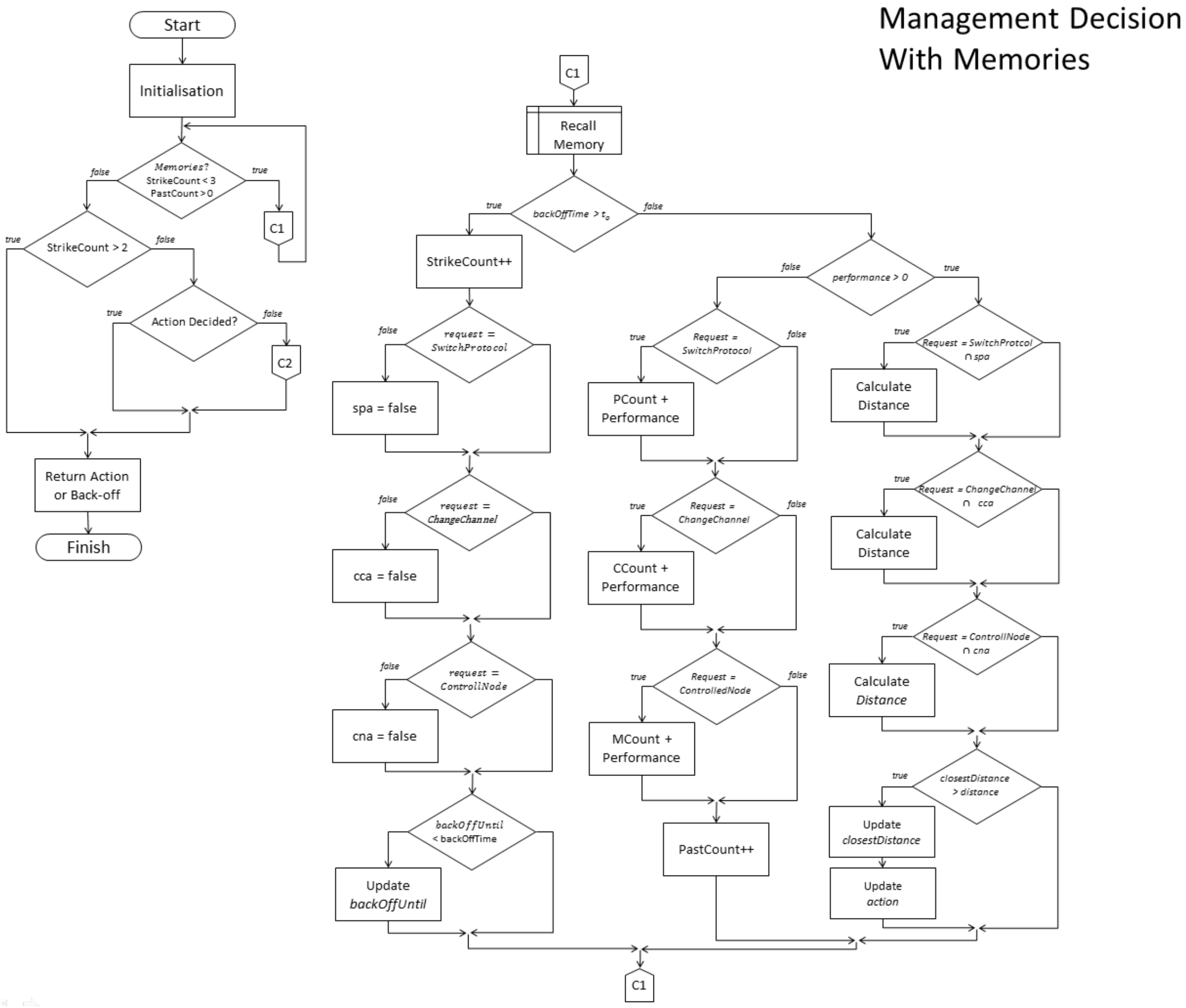


Figure 25 - Decisions from Experience of Similar States (a)

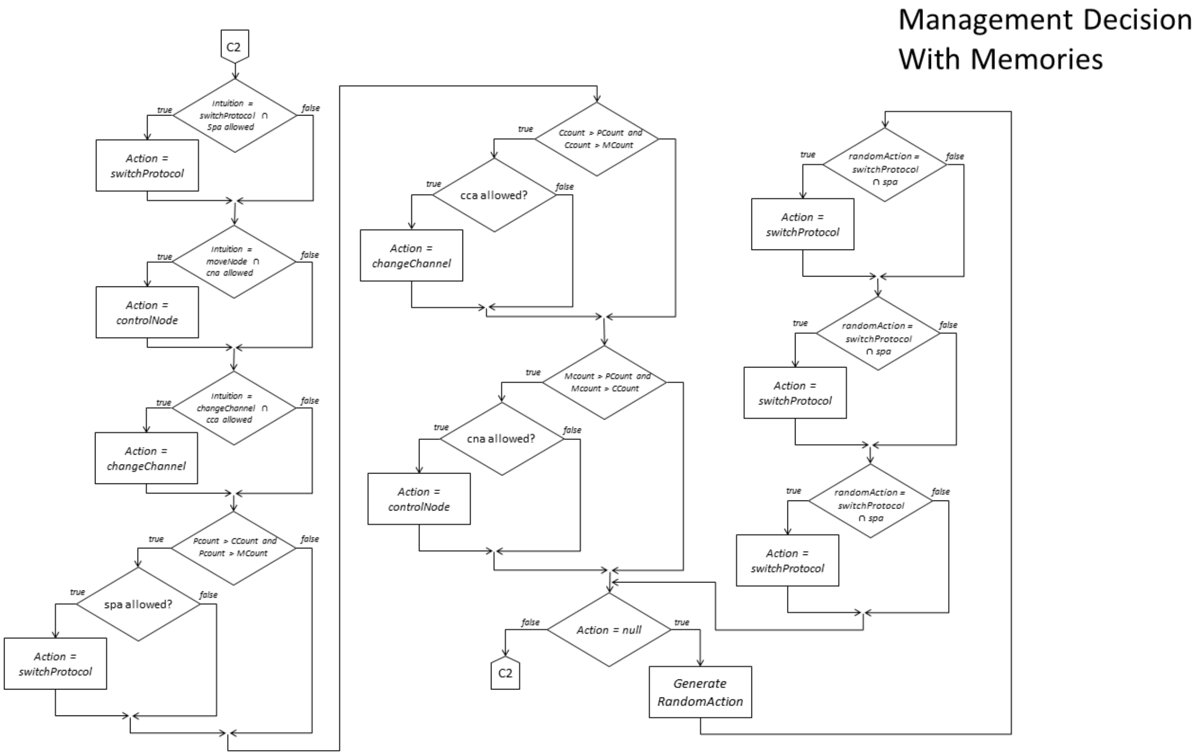


Figure 26 - Decisions from Experience of Similar States (b)

5.6.5 Cognition Cycle

The management node will perform a cycle similar to the one in Figure 27 - Cognition Cycle with some enhancements. The first of which is to allow the management node greater control over the timing of the cycle, and each manager to operate their own cycle times. Initially, an arbitrary value of 2 is assigned that dictates how frequently the cognition cycle is called. During operation, the network will have the ability to change the timing of the cycle. This flexibility has some advantages, i.e. when the network is performing well, the management node(s) may lessen the frequency of the cognition cycle, but when the network is performing poorly the source node may need additional attention.

In addition to this, it was stated that the management node will have the ability to reflect on the actions proposed. This will also affect the timing of the cognition cycle because it is not desirable for the management node to perform several actions until the previous action has been completed and reflected upon.

5.7 Multiple Managers

If there were only one source and one manager in the data communications network the manager would have complete authority and be accountable for any actions it imposed on the network. Realistic models will be produced and the scenario with one source and one manager does not meet most situations, therefore, multiple sources and multiple managers will need to be accounted for.

There should not be a position where a manager of a source node counteracts the action of another manager of another source node. This could present a situation where one manager changes from the current routing protocol and immediately after another manager makes the same request not realising this request has already been made. This causes an unnecessary overhead and in this situation results in the previous action being reverted. Another situation might cause a manager to make use of a controllable node in order to strengthen or re-establish a communications path. It could be that the controllable node is already undertaking a mission for another manager node; it would be unfair and unproductive for another manager to interfere with the initial request of the other manager. These are two examples but the same is true for changing operating frequency.

As more managers are added into the equation the situation worsens and makes the monitoring of cognition very complicated and difficult (previous decisions would be counteracted and overridden constantly). The application of cognition would inherently fail if there were contending management nodes with no authority.

5.7.1 Network Command Centre

In order to manage multiple requests from multiple managers a higher authority will be implemented called the Network Command Centre. The Network Command Centre will allow or deny requests of manager nodes based on the information they have from other manager nodes.

It is believed that it is the job of the management node to call upon the Network Command Centre each time it wishes to perform and action to increase the performance of a given source node. The Network Command Centre is less interested in source nodes and more concerned with overall network performance.

A manager will make a request to the Network Command Centre (NCC). The NCC will acknowledge the request. At this point the Manager node will not be able to make the same request again until the request has been granted or denied. Once an acknowledgement has been received by the manager node, that request is blocked.

Already discussed is how the manager node makes selections based on previous knowledge and experience; the manager should also be aware of outstanding requests so that it does not choose the same course of action and make the same request as the outstanding request.

5.7.2 NCC Cycle

The NCC will be called by manager nodes each time a manager node wishes to make use of network resources or make a change to the operation of the network. The NCC will not action these requests as they are made. Instead the requests will be queued until a suitable time has been identified in order to make the request.

When the NCC is called by a manager node, the NCC will check its own memory to ascertain if a previous action has been completed or if the network has had enough time to stabilise after the completion of the action. The NCC knows if enough time has elapsed to allow stabilisation of the network by checking the value stored in timeToStabilise, which is updated each time an action is authorised. If the previous action is on-going or if the network has not had time to stabilise the NCC should perform a recursive call at some time in the future (Manager Nodes are not allowed to continually call the NCC once their request has been acknowledged).

If the NCC is able to respond to the requests it will do so. The purpose of the NCC is to manage multiple requests and avoid contention from opposing Manager nodes, therefore, at the given moment there may be several requests.

A situation where multiple changes are made to the network at the same time needs to be avoided. Therefore, each time the NCC responds to the request it will choose the most suitable request. The NCC will do this by first keeping a tally of the most popular requests. The most popular request will be sanctioned and the relevant nodes will be updated to inform them if their request was granted or denied. Next to be discussed is each of the requests that could be made by the NCC.

5.7.3 Switch Channel

The NCC will get it current operating frequency and ascertain the next frequency. See Figure 7 - Channels assignment (Grigorik, 2013) for available channels and the frequencies associated with the channels. Once the NCC has the next frequency the NCC should instruct the participating nodes within the network to switch frequency. The NCC will then switch its own frequency.

In order to allow the switching of frequency from one channel to another, the network should be allowed time to stabilise. In order to do this the NCC will select a suitable back-off time, the back-off time will be used in two ways.

Firstly, the NCC will send the back-off time to the manager node(s) that made the original request; because the request made by the manager node was granted the back-off time will be used as an indicator for when to check if the action requested (and authorised) did or did not improve the performance of the network. This will allow the manager node to reflect on the action performed and ascertain if the action improved or decreased the performance of the source node that it is managing.

Secondly it will also be used by the NCC to allow the network time to stabilise. During times when the network is stabilising the NCC will still accept and acknowledge requests. However, it will not consider any request until the network has had sufficient time to stabilise.

5.7.4 Switch Routing Protocol

It is envisaged that switching routing protocol will work in a similar fashion. Firstly the NCC ascertains the currently operating routing protocol. The NCC will then instruct all the participating nodes on the network to switch to the alternative routing protocol. Next the NCC will switch its own routing protocol.

When to switch protocol is determined by the manager node after considering all available memory entries recorded (Section 5.6.4 Similar States).

Similar to change frequency, it is expected that a momentarily period where the network will need time to adjust to the change in routing protocol. Therefore the NCC will select a suitable back-off time to allow the network time to stabilise. Again, this back-off time is used by the manager node and NCC as described in the Switch Channel section.

5.7.5 Controlled Node(s)

Controlled nodes should be implemented in such a way that they help manager(s) strengthen or create new links when there is no communication path between the source node and the destination node. Should the most popular request be to move a controlled node the NCC will consider which source node(s) are in most need of the controlled node. This will be achieved by using the predicted receive rate of the destination node. The lower the predicted receive rate the more likely that a particular manager will be given control of a controllable node.

There could be zero or more controlled nodes so the first thing the NCC must do is check to see if there are any controlled node, if there are the NCC should query each controlled node in order to obtain their current status (i.e. they might be busy performing a previous task).

When the available controlled nodes have been identified the manager will locate the controlled node that is closes to the midpoint between the source and the destination.

The NCC will then ask the controlled node how long it will take to get to a location between the source node and the destination node. This is not as easy as this initially sounds because both the source node and the manager node may be in transit. Therefore the controllable node must be able to predict both the path of the moving source node and the destination node for given times in the future, and then calculate when it will reach the desired location with the controlled nodes current speed. The control node will inform the NCC when it will be in the desired position.

The NCC will then calculate if the source node and destination node will be in range at this time in the future by performing a similar projection for the source node and destination nodes trajectory. This is to determine if the source node and destination node will be in range at the given time in the future. If this is the case there is no point in relinquishing control of the controllable node.

If the NCC determines that a controllable node might be of some use for the manager node the NCC will authorise the request and information the Manager node. At this point the manager node can select the source → destination pair and instruct the Controllable node to move to a new location in order to reform the communication link.

The process will continue for other requests for the use of a controllable node until there are either no more requests or no more controllable nodes are available (i.e. they are all busy).

When the NCC grants the request for a controllable node the NCC will also send a back-off time so that the manager cannot make the same request again. The back-off time will be based on how long it takes the Controllable node to arrive at its destination. In addition to this the manager will evaluate how useful the use of a controllable node was, the time of evaluation is also based on the back-off time.

The back-off time is also used for when the NCC denies requests for manager nodes so that they do not make the same request again until the back-off time has elapsed.

5.8 Novelty

As discussed in the Introduction there is limited research at the moment in the field of Cognitive Network approaches applied to a MANET network. In addition to this, the methods in the simulation software that allow the modelling of a MANET with cognitive attributes do not exist. Therefore, creation and implementation of the proposed cognitive ideas and data structures as discussed above will need to be completed.

Although there is much research that compares routing protocol design in various scenarios or models, there is no evidence of research that shows comparisons between a MANET with cognitive attributes that are proposed in this thesis against other routing protocols where there is no cognition applied.

It is the aim of this research to establish if allowing the network to make decisions based on past experience will increase the performance of the network. In addition to this, particular attention is paid to specific applications (i.e. source nodes).

5.9 Chapter Summary

This chapter has discussed the design of a MANET with cognitive attributes which allows the network to adapt to changes. Also discussed is the topology and detail of the roles of the nodes (i.e. the source, the destination, the management and the controlled nodes).

Discussed next were the cognitive attributes which detailed a cognition cycle, how candidate nodes are established and how a candidate node is elected to become a manager node. The design is reinforced by using standard flowchart notation with standard mathematical notations.

Later in this chapter the discussion turned to memory, information capture and information processing. All of which will be used when the design is implemented and create a MANET with cognitive attributes.

The penultimate section of this chapter discusses how to handle multiple managers making many requests, some of which may be contentious by introducing and detailing the Network Command Centre.

The chapter closed by detailing the novelty of the work and this chapter summary.

Chapter 6 Implementation of the MANET Topology

This chapter presents a feasible solution to the challenges MANETs face as discussed in the previous chapters. The solution is based on the amalgamation of research from several areas which has been adopted and applied to a MANET.

This chapter explains the theory of the methods that were used in programming the simulation models. The detailed descriptions of the methods that were created can be found in Appendix 2. The methods that are used that are part of the NS3 library are not discussed because this documentation is available from the NS3 website. Although the code produced for this thesis call on and make use of methods or libraries that are included in NS3, most of the methods required by this simulation model were not available and had to be created and programmed entirely from scratch.

The simulation algorithms are split into three chapters: Chapter 6, Chapter 7 and Chapter 8. In this chapter the major settings of the network are discussed, beginning with the nodes used in this network model (Section 6.1), the implemented nodes that used the AODV routing protocol is explained as are: nodes that used the DSDV routing protocol, nodes that are allowed to be controlled, and nodes that are created on the intent of causing congestion.

Next, in section 6.2, detail is given to the implementation of the nodes into their world. Several placement strategies have been created (as discussed in Chapter 5), namely the placement strategies allow nodes to be placed in: random locations, a single location, two even groups, two fractional groups of nodes (i.e. a third in one group and two thirds in the other group), also added was an element of randomness in the placement of the groups, thus they did not occupy the same space but were grouped into clusters; grid formation and at specified locations.

Then discussion focused on the mobility models that were used (Section 6.3). Most of the models were implemented for this research and consists of: Piecewise linear motion with constant speed towards a given goal, motion along spirals, a diamond mobility model and a lake scenarios model. The Random Waypoint mobility model was also used. Next discuss shifted to the implementation of an autonomous movement model for the controlled nodes. Also discussed were some of the difficulties encountered when trying to apply more than one journey at a given time and this was overcame by implementing the **IsMoving** (p. 462) method.

In section 6.4 the techniques, formulae and algorithmic designs that were used and implemented to predict node movement in order to obtain future positions were discussed. In line with previous chapters the chapter concludes with a chapter summary (section 6.5).

6.1 Nodes

In chapter 5 (in section 5.2.1) key components of the network were detailed by explaining the basis network elements, namely the nodes, in this chapter the implemented design is discussed. The network model produced uses many types of nodes which include nodes that communicate using the AODV routing protocol, using the DSDV routing protocol, nodes that can be controlled by a management node and nodes that have been place in order to create congestion.

Here major variables used throughout chapters 6, 7 and 8 are introduced. N is the total number of standard nodes participating in the network; N_{CN} is the total number of controlled nodes (these are additional nodes used mainly to fix link breakages, discussed later; $N_{CN} \in \{0, 1, 2, \dots\}$). Time t is measured in seconds, the initial moment of time is taken to be $t = 0$, T is the total simulation time/duration and checking performance interval duration is t_{check}. Since NS3 is a dicrete simulator, current time is denoted by $t_0 \in \{0,..,T\}$.

As well as different type of nodes, the nodes can configure them differently, i.e. alteration of the transmission radius r_{tr}. The latter will be discussed in chapter 7.

6.1.1 AODV and DSDV nodes

Initially two containers of nodes are created, one for the nodes $A(N) = \{0,1,\dots,N-1\}$ that will transmit using the AODV routing protocol, the second container for the nodes $D(N) = \{0,1,\dots,N-1\}$ that will transmit using the DSDV routing protocol. Both containers are of the same size. Having both node containers in one simulation allows the comparison of the performance between the two routing protocols after the simulation is complete. For correct comparison, most of the setting applied to $A(N)$, are copied to $D(N)$; namely initial placement, mobility schemes, battery life, etc. However, they will never operate at the same frequency.

6.1.2 Controlled Nodes

Straight after creating $A(N)$ and $D(N)$, containers for the controlled nodes are created, again two containers $A_{CN}(N_{CN}) = \{N, N+1,\dots,N+N_{CN}-1\}$ and $D_{CN}(N_{CN}) = \{N, N+1,\dots,N+N_{CN}-1\}$, of the same size N_{CN}. Again, most of the parameters from one container are copied to another to compare them later. N_{CN} is allowed to be zero, in this case the network does not have access to controlled nodes.

6.1.3 Congestion nodes

The next containers contain congestion nodes and work the same as the methods previous defined. Please see Appendix 2 for the details of the methods that were used.

6.1.4 Additional Information

A global node container is also created in NS3, called allNodes that contains all nodes used in the simulation. The nodes are not moved from their original containers and the nodes can still be accessed using the original container descriptor. When a new node container is created the pointer reference from the original container is copied, this is useful when performing an action on all nodes and saves accessing each individual container separately.

6.1.5 Node Grouping

There are other grouping strategies that are used which are mostly incorporated into the placement strategies and are discussed subsequently; first however, splitting nodes into two even groups is discussed.

The **`SplitNodesTwoEvenGroups(p. 451)`** will split the total number of nodes into two even groups. The groups are kept track of by assigning the nodes to either a node container called Eagles or a node container called Sharks. This is done for each and every node in $A(N)$ and the $D(N)$ containers.

In order to perform simulations of a MANET a mobility model needs to be designed. This is to determine the movement of each node in a MANET, but in order for this to work, the initial placement for each node needs to be defined first.

6.2 Initial Placement

Several placement strategies were created during the development stage, the first of which allowed the placement of nodes at random locations within the world. The second placement strategy allowed the placement of all nodes in a single location and each node occupying the same relative space. The third placement strategy allowed the split of nodes into two even groups. The fourth placement strategy is a variation of the second and third and allowed the creation of two fractional groups of nodes (i.e. a third in one group and two thirds in the other group), an element of randomness is also added in the placement of the groups, thus they did not occupy the same space but were grouped into clusters; the range of the randomness is dependent on which group a particular node has been allocated to. The fifth placement strategy algorithm that was implemented placed nodes in a grid formation. The final strategy allowed the placement of each node at a specified location.

Most of the nodes are grouped by creating containers and this has been demonstrated earlier in section 6.1 and detailed in Appendix 2, when discussing creating node containers $D(N)$ and $D_{CN}(N_{CN})$ for the nodes that operate using the DSDV routing protocol, nodes $A(N)$ and $A_{CN}(N_{CN})$ that use the AODV routing protocol and the nodes that cause congestion. Now discussion shifts to how the nodes are placed into the simulation world.

NS3 allows full control over where and how to place the nodes in the simulated world. Each node in its turn is added to the list with initial (at time $t = 0$) positions that identify their x, y and z in meters, denoted by $(x(k,0), y(k,0), z(k,0))$, where k is the node ID number, $x(k,t)$ is the x-coordinate of the k^{th} node at time t, and so on. Since $D(N)$ and $D_{CN}(N_{CN})$ copy positions and motion of $A(N)$ and $A_{CN}(N_{CN})$, the coordinates of the k^{th} node will be exactly the same for all the containers. The **`DsdvPositionMatchAodv (p. 453)`** method is used to assign positions to the DSDV nodes that are derived from the current position of its corresponding AODV node. The method will loop though the container that stores the pointers for the AODV nodes, for each AODV node the position is obtained and then move the corresponding DSDV node to that position.

Some element of randomness is applied by using the rand() function that is included in the C++ math library. This enables prevention of all nodes being positioned in exactly the same location. The parameters when calling the random function are such that they only allow a random number within a given limited range, in this case between -40 and 20 meters. Thus this allocates nodes to positions that are not the same but in the same approximate location. Thus nodes can be clustered.

The models deal with the motion on the ground, which is assumed to be almost flat, so for simplicity only two coordinates are used to set a position of node k at time t: $(x(k,t), y(k,t))$.

One of the earlier simulations demonstrates this when assigning two groups of soldiers engaging in a search operation and navigating around lakes. The **`PlaceNodesTwoGroups (p. 451)`** method creates a list of allocated positions and is referenced by a pointer. The groups are by default a third of nodes in one group ($N_E = round(\frac{N}{3})$) and two thirds in the other group ($N_S = N - N_E$); however this is easily tailored to allow any fractional split of the nodes.

Each group of nodes is identified by storing each nodes identification numbers for each group in a container, the containers are called $Eagles = \{0, \dots, N_E - 1\}$ and $Sharks = \{N_E, \dots, N - 1\}$ – for no reason other than that is what came to mind at time of deciding what to call the groups.

Randomly positioned nodes are also used for some simulations. In this case the coordinates $(x(k,0), y(k,0))$ of the node are assigned random positions in range of $[0,150]$ meters in both coordinates.

Nodes can also be placed on a structured grid, for example a total of 9 nodes (the first 9 nodes in the container) can be placed at predefined positions which are top left, top middle, top right, middle left, middle, middle right, bottom left, bottom middle, bottom right. The placement pattern looks like:

```
x    x    x

x    x    x

x    x    x
```

Once the nodes have been placed into the world the nodes are assigned with a mobility model. The mobility models are discussed next.

6.3 Mobility Models

In order for Network Animator to function correctly there needs to be a mobility model assigned to each and every node in the simulation. Therefore, even nodes that do not move will require a mobility model attached to them – in order to achieve this, the Constant Mobility model is used. This method will assign a constant mobility model to all nodes in the network simulation. This method has been overloaded so that a collection of nodes can be passed to which to apply the mobility model to, thus not all nodes in the network. The overloaded method accepts a node container in its parameter list.

The designed and implemented **`MoveNodeTo (p. 454)`** method, which is never called directly but by other mobility methods and is used to update a given nodes x and y coordinates to the values passed into the parameter list (coordinates of the next goal). A mobility model object is created and the mobility model pointer of the node that is passed into the methods parameter list is obtained.

When considering the motion of each node there also needs to be consideration of the speed and the direction that the node travels at certain intervals during the simulation run. The speed could be variable, i.e. a node could increase or decrease their speed at any given interval. In this research piece-wise constant speed (varying velocity though) is used mostly, but the speed can be changed during the simulation when needed. Random variations to the speed for the nodes on different intervals can also be applied, so the motion is more realistic.

The mobility model should also allow for a node to join or leave the data communications network and at which interval in time that this has occurred.

6.3.1 Piece-wise linear motion with constant speed toward the goal

Once the pointer to the current nodes mobility model has been obtained a method can be used to get the current position of node k at current time t_0. This returns a vector that stores $(x(k, t_0), y(k, t_0))$ co-ordinates of node k. These co-ordinates are updated to the values passed in the parameter list, so that the next goal position $(x_g(k), y_g(k))$ on the node is set. The moment when the node will reach the goal position depends on the node's speed and the time when it will start the motion (the node might be still moving to the one of the previously set goals). The position of the node is then set by invoking the **`SetPosition`** method. At the same time its corresponding node in the other container (for the alternative routing protocol) will move in an identical fashion. For example if node 1 of the AODV container moves, then node 1 of the DSDV container will assume the same mobility.

Once node k gets a new goal position $(x_g(k), y_g(k))$ to aim for, a check occurs to establish if the node requested to move is valid, i.e. it exists. If the node does not exist the program stops with an assert message.

The current simulation time t_0 in seconds is obtained next. A node should not be able to move to two locations at the same time as this defies the laws of physics – albeit NS3 allows this. Therefore, if the node is already moving the walk is rescheduled. If the node is not moving a vector to store the current position $(x(k,t_0), y(k,t_0))$ of the node with ID k is created.

The **IsNodeMoving (p. 462)** method is used to check to see if a node is moving or not. The method accepts a node identifier to establish which node to check for movement. If the node is not moving a value of false is returned to the method that invoked this method, in this case the **NodeWalkTo** (p. 455) method. If the node is moving, when the node will complete its current journey is obtained from a dynamic data structure. The movement is then rescheduled to begin at a new time using this value. Information regarding when a node will complete its journey is obtained by calling the **GetTimeOfJourney** (p. 463) method.

Another vector called goalPosition is then created to store coordinates of the goal (x_g, y_g) as x and y values passed to the methods parameter list. The distance between the current and goal positions is calculated by using the Pythagoras theorem

$$d(k, t_0, x_g, y_g) = \sqrt{(x_g - x(k,t_0))^2 - (y_g - y(k,t_0))^2}$$

implemented in the **GetDistance (p. 465)** method. The method returns the distance between the two points as a double value.

The duration of the walk is calculated by dividing the distance by the speed of the object that is walking – small random variations are added to each node so that each node is not moving at exactly the same speed as this is unrealistic. The number of steps N_{steps} that the object will have to take in order to get to its destination is calculated by dividing the step size (stride size) by the distance. Step size (stride size) allows for different objects (i.e. soldiers or tanks) to be accounted for. It also allows the application of a realistic transition from the original position to the destination position and better models how an object moves.

How long each step will take t_{step} is calculated so that the position of the node at each step taken is known, and the time that the node will be in that position. This is calculated by dividing the number of steps by the duration of the walk.

Each move for x and y is calculated by applying the following formula

$$x = \frac{x_g - x_c}{N_{steps}}, y = \frac{y_g - y_c}{N_{steps}}.$$

A method called **`RecordMovement`** (p. 466) is invoked that records the node ID and when the journey finishes. This prevents the same node moving again whilst already in motion.

This is the information required for moving the node to a given location in steps. The method **`NodeMoveTo`** (p. 454) is called N_{steps} amount of times, where N_{steps} is the number of steps. The call to the method **`NodeMoveTo`** (p. 454) is based by creating a scheduled event. The time is calculated by adding $N_{steps} \ \times t_{step}$ to the original time of the original position.

6.3.2 Motion along spirals

During the testing phase of the simulations a defined and implemented mobility model, which is based on the Archimedean spiral was used because it allowed for verification that the parameters set were working as intended, for example initial node placements and defined mobility patterns. The Archimedean spiral has the property that the movement of a node is outwards from the starting point in successive turnings in a spiral as depicted in Figure 28 - Archimedean spiral.

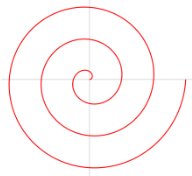


Figure 28 - Archimedean spiral

Motion along the Archimedean Spiral drives the node in an anticlockwise spiral and uses the following motion equations for the nodes for the next time point $t_1 > t_0$

$$x(k,t_1) = x(k,t_0) + \frac{t_1}{2\pi} cos(t_1); \qquad y(k,t_1) = y(k,t_0) + \frac{t_1}{2\pi} sin(t_1),$$

here $t_0 \in [0, \mathrm{T}]$ is the current time, step size $t_1 - t_0$ is 0.05 seconds, i.e. the time mesh for the mobility model is $\{0, 0.05, 0.1, \ldots, \mathrm{T}\}$.

To move the nodes clockwise along the spiral, the following formula is used:

$$\mathrm{x(k,t_1) = x(k,t_0) + \frac{t_1}{2\pi} cos(-t_1)}; \qquad \mathrm{y(k,t_1) = y(k,t_0) + \frac{t_1}{2\pi} sin(-t_1)}$$

6.3.3 Random Waypoint Mobility

The **`MobilityRandomWaypoint`** method allows the installation of a random waypoint mobility model onto the nodes. Random Waypoint is one of the most popular mobility models to evaluate MANET routing protocols. The nodes move randomly and with no restrictions. The goal position and speed are chosen randomly and independently of other nodes. To begin, an object is created in the object factory (a class that is part of NS3). Once the object is created an assigned minimum and maximum values for x and y attributes that restrict the movements of the nodes are used. For example, if the assigned minimum value of 0.0 and a maximum value of 250.0 for x and a minimum value of 0.0 and a maximum value of 250.0 for y the node would only be able to move within a 250 meter squared box.

A constant node speed value is then assigned. Next a list of allocated positions is created and is referenced by a pointer. Each node in turn is then added to the list with positions that identify their x, y and z co-ordinates.

Next the mobility model is installed onto each and every node inside a given node container. In this case the random waypoint mobility model is assigned to all nodes in the congestionNodes container.

The **`CourseChange`** (p. 466) method is a callback method that is called when a course change has occurred with a node using the AODV routing protocol. This method is only necessary when using the random waypoint mobility model as the path a node takes is not predetermined. The method ensures that the paired DSDV nodes move exactly as their AODV counterparts and was necessary so that the results between the two protocols can be compared.

When the method is called a pointer to a mobility model for the node that moved or changed course is passed to this method. Using this pointer, the position of the node that moved can be obtained by calling the **`GetNodePosition`** (p. 464) method. The position returned is used to set the position of the corresponding DSDV node by calling the **`MoveNodeTo`** (p. 454) method.

6.3.4 Moving in Diamond Pattern

The other mobility model that was used is the **`MoveNodeDiamondPattern`** (p. 460) mobility model. To further test the mobility and network parameters a diamond mobility model as shown in Figure 29 - Diamond Mobility Model was implemented. This was primarily used to ensure the antenna design worked as intended and allowed the direction of two distinct groups of nodes in a diamond formation. When nodes in each group would become out of range and in range of the nodes in the other group could be predicted using this model. Using a temporary clustered group of nodes also allowed the check that packets were being forwarded through intermediate nodes correctly. This mobility model is used in the first network model that was simulated and the scenario applied to it shows two clustered groups of nodes navigating around lakes. The results generated from this model are discussed in chapter 9.

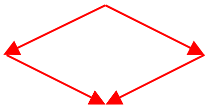


Figure 29 - Diamond Mobility Model

The **`MoveNodeDiamondPattern (p. 460)`** mobility model is intended to be used with groups of nodes that are defined in the placement strategy **`PlaceNodesTwoGroups`** (p. 451) – Eagles and Sharks. The method starts by looping through every node in the Eagles container and defines a movement that will move the node(s) from their current position to a new position that is to the left and below their current position. The node(s) in the shark's container are looped through next and defined a movement model that is similar to that of the eagle's container except they move to the right and down.

When the nodes reach their destination they then move back towards each other. This can be visualized as a diamond shape. This process is repeated and can be thought of as nodes zigzagging in and out. Figure 29 - Diamond Mobility Model illustrates this mobility pattern.

The actual movement is performed by the **`NodeWalkTo`** (p. 455) method. The **`MoveNodeDiamondPattern`** (p. 460) configures the mobility and then passes this information to the **`NodeWalkTo`** (p. 455) method.

6.3.5 Realistic Mobility Model

In Network Simulator 3 (NS3) there are no tools available for creating a realistic mobility model, however, x and y co-ordinates can be defined and a node can be assigned to this position at a specified time. This did not meet the requirements for creating realistic mobility models, therefore a model that allowed the identification of a specific node, the intended destination of the node, the speed at which the node would move and the stride size of the node was implemented. This allowed for the production of much more realistic mobility models for a given entity. Thus, a node could be created that represented a soldier and set the speed of movement to a value that is realistic to that node (i.e. walking, jogging or running). The stride parameter allows the nodes to move transitionally, which gave a smoother path or journey, thus no unrealistic discrete jumps that one may associate with teleportation.

The mobility model works out the number of steps that the node would have to take in order for it to arrive at its destination. In order for steps to be effective a mechanism was designed that allowed the scheduling of each step as an event. A second scheme was implemented as a variation on the first, which allowed the passing of angles rather than x and y co-ordinates. Both schemes calculated the time it would take to arrive at the goal, the number of steps that would have to be taken and the duration that each step would take.

The **`LakeScenario`** mobility model is intended to be used with groups of nodes that are defined in the placement strategy **`PlaceNodesTwoGroups`** (p. 451) – Eagles and Sharks.

For the first group of nodes (Eagles) each node moves from their current location to the location $(x = 0, y = 25)$ in meters. It would be unnatural for all the nodes to move at the same speed so random variations for each node were created, but they all move approximately 1.8 meters per second with a minimum speed of 1.4 meters per second and a maximum speed of 2.5 meters per second. A similar variation is applied to the x and y coordinates because each node should not occupy the same space. Once the speed and the destination have been obtained for the node, the time that this leg of the journey will take is calculated by calling the **`GetTimeOfJourney`** (p. 463) method. The value returned from the **`GetTimeOfJourney`** (p. 463) method is used as the value for when the next leg of the journey should start. Each leg of the journey is scheduled by calling the **`NodeWalkTo`** (p. 455) method.

The next leg of the journey sends the node to $(x = 100, y = 25)$, new random variations are generated for the speed and destination (thus the node could increase or decrease speed slightly for this leg).

The process is repeated for each and every node in the eagles group until simulation time has ended. Each leg of the journey x and y values are set based on their previous values. For each leg 25 is added to y so each node will continually head south. For x, a check is performed to see if the previous value of x was less than 20. If this was the case x is set to 100, if x was not less then 20 x is set to 0. This gives a zigzagging motion. For example, the nodes will head South East, then South West and repeat.

The same strategy is applied for all the nodes in the sharks group. However, they are sent in the opposite direction. Thus they will head South West, then South East and repeat.

This mobility model simulates nodes getting to a start position and then moving away from each other for a given time, before moving toward each other for a given time. Thus creating links and then breaking links.

6.3.6 Mobility of Controlled Nodes

The controlled nodes (both from $A_{CN}(N_{CN})$ and $D_{CN}(N_{CN})$) have two essentially different phases of motion: autonomious motion (when they are free, not claimed by any management node) and controlled motion (when them are called and requested to fix a link breakage).

Autonomous Motion of CN

The **`AutonomousControlledNode (p. 521)`** is designed to allow a node autonomous movement when not being requested by a manager node and when not currently performing a mission for a manager node. Although autonomy is allowed for the controlled nodes there should not be random movements, the nodes should assume good mobility in order to better aid the entire network.

It begins by getting the current simulation time t_0 by calling the **`GetTimeNow`** (p. 442) method. Determined next is if the controlled node k_{CN} is moving or not by calling the **`IsControlledNodeMoving`** (p. 463) method. If the node is not moving the controlled node will determine the best place to move to the goal by applying the following formula:

$$x_{center} = \frac{\sum_{k=1}^{N} x(k, t_0)}{N}, y_{center} = \frac{\sum_{k=1}^{N} y(k, t_0)}{N}$$

The controlled node will access the $x(k, t_0)$ and $y(k, t_0)$ coordinates of each node k in the data communications network for the last recorded positions. A running total is kept of the x_c and y_c coordinates for each node for this given period. Then the current controlled nodes' position is obtained by calling the **GetControlledNodePosition** (p. 527) method $(x(k_{CN}, t_0), x(k_{CN}, t_0))$. A goal position (x_{center}, y_{center}) is set by calculating the center point of the world which is based on the summation of the $x_k$and y_k coordinates. The summation is divided by the number of nodes in the network.

Next the distance $d(k_{CN}, \overrightarrow{x_{center}})$ between the current position (x_c, y_c) and the goal position (x_{center}, y_{center}) is obtained by calling the **GetDistance** (p. 465) method. Next the number of steps that will need to be taken for the given distance is calculated.

$$N_{steps} = \frac{d(k_{CN}, \overrightarrow{x_{center}})}{|v_{CN}|}$$

Where $|v_{CN}|$ is the current speed of the controlled node.

Once the goal position, the current position and the step count has been obtained each x_{step} and y_{step} is calculated by applying the following formulae

$$(x_{step}, y_{step}) = (x_c + \frac{x_{center} - x_c}{N_{steps}}, y_c + \frac{y_{center} - y_c}{N_{steps}})$$

Then the controlled node is moved by invoking the **MoveControlledNodeTo** (p. 454) method.

Controlled Motion of Controlled Node

The **MoveControlledNode (p. 522)** is designed to select a controllable node for a manager node. There could be a situation where a management node is managing multiple sources, therefore first the management node needs to establish which source node is in most need of the use of a controllable node; the latter is done by calling the **GetSourceNodeForControlledNode** (p. 525) method.

For the source node, it needs to be determined which of the sources destinations is receiving the fewest data packets with the aim to strengthen the link between the source node that is performing worst for this manager and the destination that is performing worst for the selected source. This is achieved by calling the **`GetDestinationNodeForSourceNode`** (p. 525) and passing the identifier of the source node.

A controllable node identifier (that is not busy) which is the closest to the midpoint between the source and the destination is obtained next by calling the **`GetClosestControlledNode`** (p. 526) and passing the source node identifier and the destination node identifier. Next a vector is created that will store the x, y and z coordinates of the controllable node selected by calling the **`GetControlledNodePosition`** (p. 527) method.

At this point the values that represent the source node, the destination node and the current selected controlled nodes position have been obtained. This information is used to determine the future position at which the controllable node will be at the midpoint between the source node and the destination node, this new position is return and stored in a vector called cNodeNewPosition by calling the **`GetControlNodeMidPointXY`** (p. 527) method. If the controlled node cannot get to the midpoint within a given time constraint the returned value will be equal to the controlled nodes current position.

Since the future position that the controllable node will need to reach has been obtained, a calculation is performed to determine how long it will take the controllable node to get there by calling the **`GetTimeOfJourneyControlledNode`** (p. 464) and pass the current controlled nodes position, the controlled nodes goal position and the speed at which the control node moves.

Next two checks are performed; the first is to ensure that the controllable node will reach its target position within the given time constraint (i.e. a possible constraint is the remaining duration of the simulation). This is the case if the current position of the controllable node is not the same as the position stored in cNodeNewPosition.

The second check is based on predicted positions of the source node, the destination node and time of which it takes for the controllable node to reach its intended new position. The predicted position of the source node at the time it would take in order for the controllable node to get to its intended goal is obtained by calling the `PredictPostion` (p. 469) method and pass the source node identifier and the time of which the journey takes. The same process occurs for the destination node. Next the distance between the source node and the destination node from the predicted positions is calculated by calling the `GetDistance` (p. 465) method and pass the predicted source node and destination node positions. The value returned from `GetDistance` (p. 465) is used to check that the distance is within the transmission radius by calling the `InRange` (p. 468) method and passing the distance. This is done because the use of a controllable node should not be allowed if the source and destination are in predicted transmission range before the controllable node gets there.

If the controllable node will get to its intended goal within the given time constraint and if the controllable node will get there before the source node and destination node are in transmission range, the manager node is allowed control of the node. In order to control the node the `ControlledNodeWalkTo` (p. 457) method is called and passed the control node identifier, the position that the node should move to, the current step size (stride size) and the speed at which the controlled node moves.

The final action performed by this method is to return the time it takes the node to reach its destination to the invoking method.

The `TimeAllowedToMoveControlledNode (p. 473)` method obtains the current simulation time, sets two variables named nodeMovingUntil and backOffUntil to the duration of the simulation run. Each controllable node is checked in turn to see if it is moving by calling the `IsControlledNodeMoving` (p. 463) method and passing the appropriate identifier for the controllable node. The method returns 0 if the node is not currently moving or a value that indicates when the node will be finished moving. If the node is moving and the time at which the node will be finished moving is less than the current value in back off time, the back off time is updated to reflect when a node will be finished first. Once each node has been checked the back off time is returned to the invoking method. Zero indicates that one or more control nodes are not busy, a value greater than zero reflects the time when a node will first become available.

The method `RecordControlledNodeMovement(p. 473)` is used to record a movement for a controlled node. In particular, there will be times that the manager node or the NCC will need to check if a controlled node is moving or not at a given time. In order to know if a node is moving or not, the node identifier, the current time and the time that the journey concludes is recorded to a dynamic data structure. This data structure is used each time a controlled node is assigned mobility or if a request has been made by a manager node to move a controlled node.

6.3.7 Applying two or more Mobility Models

During the testing phase of the mobility models it soon became apparent that one could apply a mobility scheme to a node and then apply mobility to the same node for the same period of time. This caused unpredictable mobility patterns and the discrete jumps that should be avoided when implementing the stride feature. It is unrealistic for an object to move in two directions or to two different positions at the same time and doing so caused the simulation to produce data that was unexpected and unrealistic. Therefore, a constraint is implemented for all the mobility models that first checked the node was not in transition from the origin to the destination, thus if the node was moving at a given time and a further mobility model was applied the simulation would warn generate a warning that the node was already moving and instruct the application of the model to wait before applying the next mobility model. The system also reported how long in seconds the current journey would take and at what time the node would arrive at its destination. This resolved the unexpected and unrealistic data situation.

Discussed so far is the initialization, node configuration, placement strategies and mobility models that are used in the simulations runs. Also mentioned was that a node should not move if it is already moving as this would create discrete jumps. Due to the nodes not having the ability to teleport from one location to another nodes are prevented from moving if they are already moving. To achieve this, the mobility state of the node is obtained by calling the **`IsMoving`** (p. 462) method.

The above Archimedes mobility model allows for simulation runs using non-arbitrary values but is not a realistic mobility model. Albeit the Archimedean spiral was useful for testing that the parameters of the network were working as intended it was difficult to think of a realistic scenario where a node would move according to the Archimedean spiral.

The Diamond mobility model was a little easier to relate to a real world situation or scenario and allowed the navigation of nodes around certain obstacles, in this case lakes.

6.4 Predicting future positions

Throughout simulations design future positions of the nodes are predicted. The **`PredictPosition`** method accepts a node ID k and a future time t_1 in order to attempt to predict that node's position at the future time passed. It is assumed that the node is moving at a constant velocity $\overrightarrow{v(k)} = (v_x(k), v_y(k))$ and the current time is t_0.

The method begins by creating three vectors, currentNodePosition $(x(k,t_0), y(k,t_0))$, previousPosition $(x(k,t_{-1}), y(k,t_{-1}))$, and predictedPosition $(\tilde{x}(k,t_1), \tilde{y}(k,t_1))$. Four local variables are created to store a previous time value t_{-1}, an x co-ordinate of the velocity $v_x(k)$, and y co-ordinate $v_y(k)$ and the prediction time t_1. In contrast to the exact values tildes are used to denote all the predicted values.

The vector currentNodePosition is populated to the nodes' current position by calling the method **`GetNodePosition(p. 464)`** and passing the method the ID of the node of interest.

An index to a dynamic data structure is created called myPositions, unlike normal circumstances where one would point to the start of the data structure, the position of this pointer points to the end of the data structure. This is done so that the last known position of the node is obtained as this helps improves the likelihood that the predicted position is correct. The predicted position data structure is looped through until the node ID is located. Once the correct node has been identified the recorded x co-ordinate $x(k,t_{-1})$, and y co-ordinates $y(k,t_{-1})$, is copied in to the vector called previousPosition. Also copied is the time t_{-1} that the node was at those co-ordinates in to the previousTime variable. The sum of the currentTime added to the futureTime in the predictedTime variable is stored.

Before the calculation of the nodes predicted position, the predicted velocity for the given node is calculated. To calculate the predicted x component of the velocity the following formula is used

$$\mathrm{v_x(k)} = \frac{\big(\mathrm{x}(k,\mathrm{t_0}) - \mathrm{x}(k,\mathrm{t_{-1}})\big)}{(\mathrm{t_0} - \mathrm{t_{-1}})}$$

Where $\mathrm{v_x}$ is the velocity for co-ordinate x; $\mathrm{x(t)}$ is the current x co-ordinate, $\mathrm{t_1}$ is the current time and is the previous $\mathrm{t_0}$ time. To calculate the predicted y component of the velocity the following formula is used

$$\mathrm{v_y(k)} = \frac{\big(\mathrm{y}(k,\mathrm{t_0}) - \mathrm{y}(k,\mathrm{t_{-1}})\big)}{(\mathrm{t_0} - \mathrm{t_{-1}})}$$

Where $\mathrm{v_y(k)}$ is the velocity's co-ordinate y for node k; $\mathrm{y(t)}$ is the current y co-ordinate of the node's position; $\mathrm{t_0}$ is the current time and t_{-1} the previous time. Then calculated is the predicted position of the node by using the formula

$$\tilde{x}(k,t_1) = \mathrm{x}(k,\mathrm{t_0}) + \mathrm{v_x}(k)(\mathrm{t_{-1}} - \mathrm{t_0});\ \ \tilde{y}(k,t_1) = \mathrm{y}(k,\mathrm{t_0}) + \mathrm{v_y}(k)(\mathrm{t_{-1}} - \mathrm{t_0})$$

Where $\tilde{x}(k,t_1)$ is the required predicted co-ordinate x and $\tilde{y}(k,t_1)$ is the predicted y co-ordinate. The values are stored in the vector predictedPosition and this vector is returned to the invoking function.

As per normal scope rules the vectors currentNodePosition, previousPosition and predictedNodePostion no longer exist after the returned vector.

6.5 Chapter Summary

This chapter presented and discussed the configuration of the nodes that are used in the network simulator, specifically discussed were the types of nodes that are used (AODV, DSDV and congestion). Node grouping was then discussed and section 6.1.5 explained how the nodes were grouped into containers called Eagles and Sharks. After grouping strategies the discussion turned to how nodes are placed in the simulated world and several examples of initial placement strategies were given (random locations, a single defined location, even clustered groups, and proportional clustered groups). Next the discussion turned to mobility models and the models that were implemented for this thesis were discussed which consisted of a piece-wise linear motion model with constant speed towards a given goal, a mobility model based on the Archimedean spiral, a diamond mobility model and a realistic lake scenario model. In addition to this the random waypoint model was discussed that is available as part of the NS3 simulation software. Controlled nodes were also discussed and explanations given for how the controlled nodes have both autonomy and can be instructed to move to a given location. Difficulties in applying more than one mobility model were discussed and how these difficulties were overcome with the introduction of a method to check to see if a node is moving before apply a new mobility model. The chapter concluded with discussion on how the future location of nodes at a given time could be predicted.

All the basic components of a MANET are now set with the exception of the transmission configuration. The nodes should communicate; this is discusses in the next chapter.

Chapter 7 Implementation of the Network Configuration

This chapter progresses on the design and implementation of a MANET testbed, started in chapter 5. The aim is to achieve Objective 2 (To design a suitable testbed, whilst ensuring that the solution is rigorous, transparent, and replicable for the testing of the scientific theories) and get a testbed ready. The chapter begins by discussing the node hardware (Section 7.1) and discusses the Physical Layer and the Media Access Control layer, the transmission radius, the protocol stack, addressing and interfacing, routing protocols and channel assignment.

Next to be discussed is how traffic flows are created between a source and destination pair (Section 7.2), in this section the configuration of the nodes that are used to create congestion are also discussed.

Most of the analysis is on traffic analysis (i.e. packets); therefore in section 7.3, discussion of how performance is measured is given including a discussion of how received, sent and dropped packets are recorded. Furthermore, the discussion details how received, sent and dropped packets rates are calculated before concluding the section by discussing how this information is recorded for analysis once the simulation has run.

In section 7.4 the energy model that was used which represents a self-contained lithium battery is discussed. Realistic initial energy levels are set and current is drawn when required.

Section 7.5 is dedicated to how callback methods are set up to log information during the simulation run. Also discussed is the batch file system that was implemented which allows the scheduling of simulation runs (each run took several hours to complete; the more complex runs were started at night and left running ready to be completed at some point the next day).

The chapter concludes with a chapter summary (section 7.6).

7.1 Node Hardware

In order for a node to communicate with another node there needs to be some form of networking technology applied. This technology operates based on two layers of the TCP/IP model. This allows the creation of an interface to the wireless communication medium by implementing physical hardware and creating a media access control protocol. The layers are depicted in Figure 30 - TCP/IP and OSI model. The implementation details and the characteristics of the antenna and media access control are discussed next.

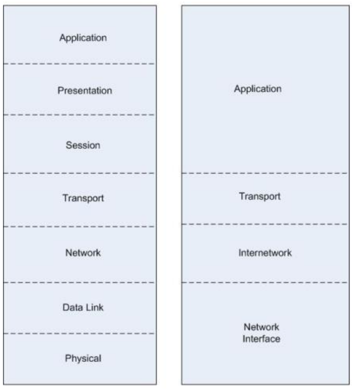


Figure 30 - TCP/IP and OSI model

Once the nodes $A(N)$ and $D(N)$ are created, methods to create the network devices (p. 444), the stack (p. 444), assign IP addresses (p. 444), create an interface to the network medium (p. 445), assign initial placements of each node (p. 451), assign a mobility model (p. 454) and install an energy source that represents a lithium battery (p. 529) are called.

7.1.1 Physical Layer and MAC

The physical layer contains the characteristics associated with the physical hardware (the network interface card and wireless transmitter). The Data link layer contains the logic that allows the physical hardware to make a connection to the network.

The physical operation of the antennas is based on the Friis Propagation Loss Model which is based on a formula derived in 1945 by radio engineer Harald T Friis. Originally the equation was

$$\frac{P_r}{P_t} = \frac{A_r A_t}{d^2\,\lambda^2}$$

According to NS3 documentation the equation that is used is that based on an isotropic antenna with no heat loss $\mathrm{A_{isotr}} = \frac{\lambda^2}{4\pi}$. Which alters the formula to

$$\frac{\mathrm{P_r}}{\mathrm{P_t}} = \frac{\mathrm{A_r A_t}}{(4\pi \mathrm{d})^2}$$

Modern extensions made to the original equation which means that the formula becomes

$$\mathrm{P_r} = \frac{\mathrm{P_t G_t G_r}\,\lambda^2}{(4\pi \mathrm{d})^2 \mathrm{L}}$$

where: $\mathrm{P_r}$ is the reception power (Watts), $\mathrm{P_t}$ is the transmission power (Watts), $\mathrm{G_t}$ is the transmission gain (unit-less), $\mathrm{G_r}$ is the reception gain (unit-less), λ is the wavelength (metres), d is the distance (metres) and L is the system loss (unit-less).

This is the formula used for the antenna and the values used are a transmission gain of 4.5, a reception gain of 4.5, and a reception power of 1. This gives a maximum transmission range of 100 metres, which is in line with the specification of the 802.11b protocol used in the simulation design.

The wavelength λ in this mode is calculated as $\frac{c}{f}$, where c is equal to the speed of light in a vacuum or approximately equals 299792488 metres per second; f is the frequency in hertz which is set based on the operating channel of the antenna.

This propagation model is valid for network simulations in free space and is described as being approximately as the region for $d > 3\,\lambda$.

The propagation delay is based on the constant speed propagation delay model. A channel is created that operates within the 2.4GHz range and is used by the 802.11b protocol. The standards and protocol design have been discussed in chapter 3.

A non-quality of service Wi-Fi media access control is used which is typical for wireless networks and this is set this to ad hoc Wi-Fi mode.

The PHY and MAC are then integrated on to each device on each node. To keep track of the devices a dynamic data structure is created with smart pointers to the nodes.

The device installed on each node is based on that of an isotropic antenna. NS3 classes are used to implement and configure the antenna. Various parameters are set that alter the operation of the antenna, for example values for the energy detection threshold, the transmission and receiving gain, the t_x power levels and the noise model are set.

Next the MAC layer is configured and set the mode of operation to mode.

What remains is to select a frequency for the antenna to transmit and/or receive electromagnetic radio waves. This may change during the simulation but the initial value is 2412Hz which is channel 1 of the 2.4Ghz range using 802.11b wireless transmission protocol.

The physical and mac layer configuration are aggregated onto the nodes in the aodvNodes container and pointers to the aggregated objects are stored in the aodvDevices container.

Next the method **`CreateAodvStack`** (p. 444) is called to create a stack for the devices.

7.1.2 Transmission Radius

The **`TestRadius`** (p. 536) method allows the testing of the configured transmission range r_{tr}. It creates two nodes by calling the **`CreateAodvNodes`** (p. 443) method and one traffic flow between them by calling the **`CreateTrafficFlows`** (p. 474) method (traffic flows are discussed below). Next the two nodes are positioned at the same location by calling the **`MoveNodeTo`** (p. 454) method. Next a conditional "for" loop is created in order to schedule one of the nodes to move away each second. The transmission radius is known because each second a packet is sent. Once no more packets are received the transmission radius is known by observing the output to the terminal.

The **`Set21mRadius`** (p. 536) method reconfigures the characteristics of the antenna so that the transmission radius is up to 21 metres. This allowed for the testing of link breakages much sooner (during the testing stage) as opposed to the more realistic 100 metres. The energy detection threshold is set to -61.76, the CcaMode1Threshold to -61.76, the Tx Gain to 4.5, the RxGain to 4.5, the Tx Power Levels to 1, the Tx Power End to 16, the Tx Power Start to 16, the RxNoiseFigure to 4 and myRadius to 21 so that the **`InRange`** (p. 468) method works according to the new configuration.

The **`Set100mRadius`** (p. 536) method reconfigures the characteristics of the antenna so that the transmission radius is up to 100 meters. The energy detection threshold is set to -61.76, the CcaMode1Threshold to -61.76, the Tx Gain to 14.5, the RxGain to 14.5, the Tx Power Levels to 1, the Tx Power End to 16, the Tx Power Start to 16, the RxNoiseFigure to 4 and myRadius to 100 so that the **`InRange`** (p. 468) method works according to the new configuration.

7.1.3 Protocol Stack

Once the devices are configured and are associated with the nodes the protocol stack is created. The stack allows the use of existing routing protocols available in the NS3 implementation. Two distinct stacks are created, which allow the differentiation between the nodes using the AODV routing protocol and the nodes using the DSDV routing protocol. Initial work consisted of creating a single group of nodes and then switch the stack when required; however, a limitation of NS3 is that an object cannot be destroyed once it has been created. Therefore when a stack is installed on a node it cannot then be deleted and replaced by another stack. To overcome this limitation two distinct sets of nodes $A(N)$ and $D(N)$ were created. This proved to be advantageous as it allowed quick comparison of the performance of the two routing protocols and the cognitive switching of the routing protocols.

7.1.4 Addresses and Interface

What remains is to assign IP addresses to the protocol stack and then to aggregate them on to each network interface. Two distinct networks were created so that AODV would not interfere with DSDV and vice versa. The AODV nodes operate on an address range from 10.2.0.0 and the DSDV nodes operate on an address range from 10.1.0.0. The AODV nodes also operate on a different frequency to that of DSDV so that there were no media contention issues.

Once the stack exists, IP addresses are assigned to each node. This is achieved by calling the **`CreateAodvAddress`** (p. 444) method. The **`CreateAodvAddresses(p. 444)`** method allows a base IP address (from address or start address), and an IP subnet to be created. This configuration is stored in the aodvAddresses container. In this case the aodv nodes operate on network 10.2.0.0. Therefore the IP addresses for the aodv nodes start at 10.2.0.1. The subnet mask is set to 255.255.0.0 this allows the creation of more than 256 IP addresses for this group of nodes should it be required. Once the address configuration is set the addresses are assigned on to the nodes by calling the **`CreateAodvInterface`** method (p. 445). The **`CreateAodvInterface (p. 445)`** will aggregate the IP addresses to the interfaces associated with the aodv devices in sequential order.

The **`GetIPAddressFromNodeId(p. 537)`** method takes a node identifier and returns the IP address for that node. First a pointer is created to a node memory address and assigned the pointer with the address for the node identifier for the current protocol that is in use. Next a pointer for an Ipv4 class object is created and assigned to the object that is aggregated to this node. Using this pointer a type Ipv4InterfaceAddress is created and assigned to the pointer of the address of the node. Once the pointer to the Ipv4InterfaceAddress has been obtained the local IP address for this node can be obtained. This is returned to the invoking function.

7.1.5 Routing Protocols

The purpose of the **`GetProtocol(p. 483)`** method is the return the routing protocol that sent a particular packet. First, a string variable called protocol is created, this string variable is returned to the method that invoked this function.

In order to determine the protocol that sent the packet the address is searched for the value 1 or 2 at a specific location. In order to do this the IP address (Ipv4Address object) must be converted to a string object which is done by calling the **`ConvertIPAddressToString`** (p. 537) method.

A string object is essentially an array of characters, thus the array operator [] can be used to specify a particular position in the string. Due to the fact that the IP addresses start with 10.x.y.z where x signifies the network that sent the packet the address at position 3 can be searched – arrays start at 0 in C++ so fourth character is the character of interest. If the fourth character is 1 then the protocol that sent the packet is DSDV. If the fourth value is 2 then the protocol that sent the message is AODV.

A caveat is added into the algorithm because the protocol that sent the packet should not be returned if the packet is not generated by an application. Therefore, the address is checked for the value 255. If the value 255 is found in the IP address the protocol variable is set to "route discovery". This allows the invoking function to deal with the packet accordingly.

In addition to being interested in sent packets, when a packet is dropped is also of interest. Therefore, similar call backs to handle the dropped packets are created.

```
Config::ConnectWithoutContext("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/Mac
TxDrop", MakeCallback(&CogMan::MacTxDrop, this));

Config::ConnectWithoutContext("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/Phy
RxDrop", MakeCallback(&CogMan::PhyRxDrop, this));

Config::ConnectWithoutContext("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/Phy
TxDrop", MakeCallback(&CogMan::PhyTxDrop, this));
```

Depending on the stage at which the packet was dropped a call is made to the appropriate method to handle the dropped packet. If for example the packet was dropped at the MAC layer the **MacTxDrop** (p. 485) method is called. If the packet was dropped by the receiving node at the PHY later the **PhyRxDrop** (p. 485) method is called. Finally if the packet was dropped by the sending node at the PHY layer the **PhyTxDrop** (p. 485) method is called.

The above `PhyTxDrop`, `MacTxDrop` and `PhyRxDrop` methods work in a very similar way so they are discussed together. The appropriate method receives a copy of the pointed to address for the packet and places this address in to the variable p of the parameter list. A copy of the packets memory address (pointer) is created and placed in a variable called q.

The headers of the packet are then looped through until the IPv4 header is located. Once the correct header has been located an object is created. The IP source address and IP destination address is copied from the packet header and placed in variables src_ip and des_ip respectively.

The data type for the variables src_ip and des_ip are of object IPv4 address. In order to process the address it must first convert to a string object, this is achieved by invoking the `ConvertIPAddressToString` (p. 537) method and passing the IPv4 address object. The string is searched for the value 255 so that a data packet and a network control packet can be differentiated. A function to record the event of the dropped packet is invoked by calling the `RecordDroppedPacket` (p. 485) method.

7.1.6 Channels

The `SetChannel(p. 534)` method will set the channel of the node identifier that has been passed to that of the channel number that has been passed. In order to set the channel number the pointer to the physical layer is needed, the pointer to the Wi-Fi physical layer can then be obtained. Using the pointer to the Wi-Fi physical layer, the pointer to the yans Wi-Fi physical layer is obtained. The yans Wi-Fi physical layer pointer is used to set the channel number by invoking the `SetChannelNumber(p. 534)` method which is a member function for the Yans Wi-Fi Class.

The `SetChannelControlledNodes(p. 535)` method works in the same way as the `SetChannel` (p. 534) method with the exception that the node identifier that is passed accesses the controlled node container rather than the node container.

7.2 Traffic Flows

The `SetupPacketReceive(p. 478)` method accepts an IP address and a pointer of a node. The method creates a pointer to a sink object that is the local application socket. Next a local address is created by using the IP address passed to this method and the default port number. This is then bound to the sink object.

The sink object creates a callback to the method that specified. In this case when this packet is received at the destination the `ReceivedPacket` (p. 478) method is invoked. Therefore, the sink will trigger the `ReceivedPacket` (p. 478) method when a packet has been received at the destination node specified.

The `CreateTrafficFlows(p. 474)` method is where AODV and DSDV traffic flows are constructed. A dynamic data structure is populated for each source and sink (destination) with the time that the source should start transmitting and when the source should cease transmitting. The transmission rate and packet size have already been set in the `ScheduleEvents` (p. 441) method.

Once the data structure has been populated the flows are scheduled by calling the `CreateTrafficFlow` (p. 474) method.

This method will accept traffic flow information and create a traffic flow for the AODV nodes by calling the **`CreateAodvTrafficFlow`** (p. 474) method. Similarly it will create a traffic flow for the DSDV nodes by calling the **`CreateDsdvTrafficFlow`** (p. 475) method.

The **`CreateAodvTrafficFlow`** (p. 474) method takes the destination node identifier that was passed to the parameter list and obtains that nodes IP address. Using the IP address and the port number an Internet socket address is created. This address allows the destination to forward the data to the correct location. Using this address a local socket address is created based on the remote address and the UDP. The sockets are kept track of by creating a sink pointer, this pointer points to the interface address of the destination node k^d and also allows the setting of a callback for each packet pointer – this is achieved by calling the **`SetupPacketReceive`** (p. 478) method.

When the receiver application should start and finish is specified by using the startAt and StopAt variables values that were passed to the parameter list.

The method then moves forward to create the source application. The remote address created prior is used when configuring the sender of the UDP packets. The data rate of the source node is also set here. The application to a given source node that is specified in the methods parameter list is installed. Similar to the receiving application, when the source application should begin and when the source application should stop is set. These times are identical to the times of the destination application that is to receive the packets. Therefore the source application starts and finishes at the same time as the receiving (sink) application.

The **StillTransmitting (p. 476)** method accesses the myTrafficFlows vector to see if a node is currently transmitting or supposed to be transmitting. The current simulation time is obtained by calling the **GetTimeNow** (p. 442) method. Next, the myTrafficFlows vector is stepped through to locate the node of interest, based on the source node identifier k^s that was passed to this method. If the source node identifier is found a check is performed to see if the source node is currently scheduled to transmit data by comparing the assigned start time (when to begin transmitting) and the stop time (when to stop transmitting) with the value returned from **GetTimeNow** (p. 442). If the node is schedule to transmit a value of true is returned to the method that invoked this one, otherwise false is returned.

Once traffic flows are established a density reference point is created. In this thesis density ρ is the quantity of nodes in a given area, in this case the area is the transmission radius r_{tr} of a node.

7.2.1 Congestion Nodes

CreateCongestionNodes (p. 477) method creates nodes that are intent on transmitting traffic to interfere with the source node(s) traffic. A default packet size of 1000 bytes is set and a physical mode using DSSS at a rate of 11Mbps. Fragmentation for frames that are below 2200 bytes is disabled, turn off request/clear to send for frames below 2200 bytes and set the non-unicast rate to that of the unicast rate. The reception gain is set to 0 to prevent antenna gain being added.

The energy detection threshold is set to zero so that sending devices fail to detect a signal which will prevent them backing off. The transmission gain is set to a value that is higher than that of the source node(s) so that the strength of the signal generated is higher than that of the source node(s).

In order to avoid complete packet loss the control of the rate at which the congestion node(s) generate traffic is set by setting an interPacketInterval variable. In addition there is also the ability of specifying when to start transmitting the packets, when to stop transmitting the packets and the number of packets to transmit.

Areas of congestion are created by placing the nodes within the simulated geographical area at the desired locations. The congestion nodes have a restricted transmission range of up to 100 meters which is consistent with the other nodes.

This completes the node configurations; discussed next is how network performance is measured.

7.3 Performance

This section is also essential to meet Objective 2 (To design a suitable testbed, whilst ensuring that the solution is rigorous, transparent, and replicable for the testing of the scientific theories). This section will also be useful for later objectives, since discussion now turns to the way to compare simulations performance. The sections that follow discuss the control of sent, dropped, and received packets.

7.3.1 Received Packets

In order to keep count of the number of packets received the `ReceivedPacket(p. 478)` method is invoked each time a packet arrives at its destination. Count is kept of the total number of received packets by incrementing a variable called packetsReceived. Also the size of the packet is obtained and this size is added to a variable called bytesTotal which allows the recording of the total number of bytes received during the simulation run. This method also invokes `PrintReceivedPacket` (p. 478) and passes the pointer to the packet and the pointer to the socket.

The purpose of the `PrintReceivedPacket(p. 478)` method is to output to screen when a packet was received, which node sent the packet and which node received the packet. For testing purposes the distance between the two nodes engaged in the communication is also outputted.

Source IP Address

To get the source IP address a SocketAddressTag object is created called tag. The packet pointer that was passed is then used to search the packet for the tag (the Socket Address field in the header of the packet), if this is found the tag from the packet is copied in to the tag object. An Internet Socket Address object called addr is then created and assigned the value of an IP address which is obtained as a result of invoking a method called `GetAddress` that belongs to the tag object.

Destination IP Address

In order to obtain the destination address a pointer to the socket that was passed in to the parameter list of this method is used. Using the address that the socket is pointed to the GetNode method is invoked. The value returned from get node is used to get the ID of the node. The ID is then used get the pointer to a node by invoking the Get method that is part of the node container class.

The node pointer is sent when the GetObject<ipv4>() method is invoked; this returns a pointer to an IPv4 object (aggregated to that node). There may be more than one network device installed on any given node, therefore next, the IP addresses associated with the first device is obtained, this is stored in an object called iaddr. There are typically two IP addresses associated with one device, the local address and a loop back address. In order to get the local address that the device is using the GetLocal() method is invoked for the iaddr object created. This value is stored in an Ipv4Address object called addri.

A method called **`RecordReceivedPacket`** (p. 480) is called in order to store the IP addresses of the source and destination and the time that the packet was received. The **`RecordReceivedPacket`** method accepts a source IPv4 and a destination IPv4 address and stores them in variables defined in the parameter list for this method, in this case src and dst respectively.

Next the method calls the **`GetProtocol`** (p. 483) method and passes the value in the dst variable. If the **`GetProtocol`** (p. 483) method returns the value "aodv", the totalAodvPacketsReceived variable is incremented by one. If the return value is "dsdv" the totalDsdvPacketsReceived is incremented by one.

The source IP address, the destination IP address and the time at which the packet was received is recorded to a data structure called packetReceived. In addition to recording the information into the packetReceived data structure, the values are also recorded to a dynamic data structure called allPackets with the additional value 'R' to indicate that this is a packet that has been received.

Finally as part of the cognition that is discussed and implemented in chapter 6; also recorded is if the packet that was received belongs to the current protocol that has been selected by the management node. If it was the variable totalSwitchingReceived is incremented by one and the information is added to the packetReceivedBySwitching data structure.

7.3.2 Sent Packets

The `SendPacket(p. 481)` method is called every time a network device transmits a packet onto the network. The method begins by creating two data structures called scr_ip and des_ip that will hold the source and destination IP addresses from the packet header.

Next pointer is created to hold a pointer to a packet. This is the packet pointer that has been passed into the parameter list of this method.

An index is created that will allow the looping through each header of the packet. Once the IP header is reached the IP source address is copied to scr_ip and the destination IP address is copied to des_ip. The destination IP address of the header is used to determine if the packet has been sent by an application or is generated to control the network. Checks are performed to see if the packet is a network control packet, i.e. a packet transmitted by an address resolution protocol or a route request packet. Or if the packet is generated by an application.

In order to determine the origination of the packet the address is searched for the value 255. In order to do this the IP address object must first be converted to a string object which is accomplished by calling the `ConvertIPAddressToString` (p. 537) method and passing the IPv4 Address object. Once it has been converted to a string object the methods associated with the string class can be used, one such method is find.

The string object is checked for the value 255, if the adjacent characters 255 are found in the string object it can be assumed that the IP address of the packet was not intended for a specific destination. An address with the value 255 indicates a broadcast or route request packet.

The interest in determining the purpose of the packet is for logging purposes, broadcast or route request messages are not logged but packets that contain application data are. Obtaining part of the IP address allows the differentiation between these packets.

This method then calls on the **`RecordPacketSent`** (p. 482) method so that the sent packet information can be recorded to a dynamic data structure for the analysis of the results.

The purpose of the **`RecordSentPacket (p. 482)`** method is to keep count of the number of packets sent by the AODV and the DSDV routing protocols and to record the packet sent to various dynamic data structures.

Firstly, the current simulation time is obtained by calling the method **`GetTimeNow`** (p. 442), which is stored in a local variable called timeNow in preparation to be stored in the dynamic data structures.

Secondly, there is a need to establish which routing protocol sent the packet, in order to obtain this information a method called **`GetProtocol`** (p. 483) is invoked, the destination IP address from the IP header of the packet is sent to this method. If the packet was generated by the AODV routing protocol one is added to the current value that resides in the scalar variable totalAodvPacketsSent. If the packet was generated from the DSDV routing protocol one is added to the current value that resides in the scalar variable totalDsdvPacketsSent.

The IP address of the source node that generated the packet, the intended destination IP address and the time at which the packet was sent is recorded to a dynamic data structure called packetSent. Packet sent information is also recorded to a dynamic data structure called allPackets and passed an additional value 'S'. The purpose of this data structure is to record all packet information regardless of if they are sent, received or dropped.

If the packet sent was sent by the current routing protocol in operation the packet information is also recorded to a dynamic data structure called packetSentByCurrentProtocol.

The **`GetCurrentSentPacketsFromSource (p. 488)`** method takes a node identifier (nodeId) into the parameter list. This identifier is used to get the IP address of the source node of interest by calling the **`GetIPAddressFromNodeId`** (p. 537) and passing this method the node identifier. The return value is stored in a local object Ipv4Address object called nodeIP.

An iterator called vectorIndex is set up next to step through the **`packetSentByCurrentProtocol`** data structure. The vectorIndex is set to point to the end of the data structure.

When stepping through the data structure the IP address of the source node in the data structure is checked to see if it matches the IP address of the node identifier that was passed to this method. If it does, the time that the packet was received is checked, if this time is before the timeUntil value that was passed to the **`GetCurrentSentPacketsForSource`**. If the checks determine that the packet was received during the period specified for the node of interest a variable called ps is incremented by one. The next packet in the **`packetSentByCurrentProtocol`** data structure is undergoes the same checks. Each entry in the data structure is checked until a point in the structure is reached that is outside of the time period that is being checking or when the start of the data structure is reached. The value of ps is returned to the method that invoked this method.

7.3.3 Dropped Packets

The **`RecordDroppedPacket(p. 485)`** method works in a very similar way to the **`RecordSentPacket`** (p. 482) method. First, the current simulation time is obtained by invoking the **`GetTimeNow`** (p. 442) method, the return value is stored in a local variable called timeNow, which is used when the information is recorded to the dynamic data structure.

Next is the need to establish which protocol dropped the packet, this is achieved by calling the **`GetProtocol`** (p. 483) method. If the protocol that dropped the packet was AODV one is added to the current value that resides in totalAodvPacketsDropped. If the protocol that dropped the packet was DSDV one is added to the current value that resides in totalDsdvPacketsDropped.

If the packet dropped is a data packet (with a payload) the source address, destination address that were passed in to the methods parameter list with the value from the timeNow variable are recorded to dynamic data structure called packetDropped. The same information is recorded to a dynamic data structure called allPackets with the additional parameter 'D' which indicates that the packet being recorded is a dropped packet.

The next check is to see if the packet dropped was a packet currently being transmitted by the current routing protocol, if it is, also recorded is the packet information to a data structure called packetDroppedByCurrentProtocol.

7.3.4 Received/Sent/Dropped Rates

The **`GetCurrentDropRate(p. 490)`** method returns the current drop rate for the protocol that is in use. The systemState vector is stepped through, starting at the end and moving towards the beginning. Each entry is checked to see if it matches the current protocol that is in use. If it is, the number of dropped packets is recorded in a local variable called currentDroppedPackets and the time that the entry was recorded is stored in a local variable called currentTime. Stepping through the vector continues until the next entry for the current routing protocol is found, once found the same information is copied into variables previousDroppedPackets and previousTime.

The dropped rate $dr_k(t_0)$ is then calculated per second at time t_0 for this source node k by applying the following formula,

$$dr_k(t_0) = \frac{dp_k(t_{-1}, t_0)}{t_0 - t_{-1}} = \frac{dp_k(t_0) - dp_k(t_{-1})}{t_0 - t_{-1}},$$

Where $dp_k(t)$ is the number of dropped packets at time t, and thus $\mathrm{dp_k(t_{-1}, t_0)}$ is the number of dropped packet per time interval $[\mathrm{t_{-1}, t_0}]$. The value calculated is returned to the invoking function.

The **`GetCurrentReceiveRate`** (p. 491) method returns the current receive rate for the protocol that is in use. The systemState vector is stepped through starting at the end and moving towards the beginning. Each entry is checked to see if it matches the current protocol that is in used. If it is, the number of received packets is recorded in a local variable called currentReceivedPackets and the time that the entry was recorded is copied to a local variable called currentTime. Stepping continues through the vector until the next entry for the current routing protocol is found, once found the same information for this entry is copied to variables previousReceivedPackets and previousTime.

The received rate $rr_k(t_0)$ is then calculated per second at time t_0 for this source node k by applying the following formula,

$$rr_k(t_0) = \frac{rp_k(t_{-1}, t_0)}{t_0 - t_{-1}} = \frac{rp_k(t_0) - rp_k(t_{-1})}{t_0 - t_{-1}},$$

Where $rp_k(t)$ is the number of received packets at time t, and thus $\mathrm{rp_k}(\mathrm{t}_{-1}, \mathrm{t}_0)$ is the number of received packet per time interval $[\mathrm{t}_{-1}, \mathrm{t}_0]$. The value calculated is returned to the invoking function.

7.3.5 Recording Performance Stats

The **`WriteAODVvsDSDVStats (p. 539)`** method allows the comparison of how the network has performed using the AODV routing protocol verses the DSDV routing protocol and in comparison to the dynamic switching of the routing protocol.

The method begins be creating a pointer to the allPackets vector, a pointer to the packetSentByCurrentProtocol vector, a pointer to the packetReceivedByCurrentProtocol vector and a pointer to the packetDroppedByCurrentProtocol. All pointers are set to the beginning of the relevant vectors.

Next counters initialised to keep count of the amount of packets that have been sent, received and dropped for each routing protocol (for within a given interval, i.e. 2 seconds). Variables to keep count of how many times the switching of a protocol had a positive and a negative effect on the performance of the network are also initialised.

The allPackets vector is stepped through starting at the beginning, each entry is checked until the end of the vector is reached. Each entry contains packet information and is checked to establish which routing protocol is in operation (for that packet) and the reason why the packet was recorded (i.e. it could have been sent, received or dropped). The relevant counter for each packet is incremented, for example, if the routing protocol in operation was DSDV for that packet and the packet is recorded as being dropped, a counter labelled **`dsdvD`** is incremented. Thus, for each packet one of the following counters is incremented:

```
aodvS routing protocol is aodv, packet is recorded as being sent.
aodvR routing protocol is aodv, packet is recorded as being received.
aodvD routing protocol is aodv, packet is recorded as being dropped.
```

There are similar counters for the dsdv routing protocol: dsdvS, dsdvR, dsdvD.

This method accepts an interval into the parameter list. This allows the grouping of statistics into manageable chunks of time (i.e. every 2 seconds). Therefore as the allPackets vector is stepped through there is a pause at each interval in order to perform some analysis and then write the results to a data file.

Three new counters are created (for each interval). The vector packetSentByCurrentProtocol is stepped through (for the same interval, i.e. the first interval could be 0-2 seconds). How many packets were sent by the currently operating protocol for this interval is counted. The same is done for received packets and dropped packets.

How many packets were received by the AODV routing protocol for this period is compared against the DSDV routing protocol for the same period. A counter called goodA is incremented if the AODV routing protocol received more packets for this period. If the DSDV routing protocol received more packets for this period a counter called good is incremented. If both routing protocols received the same number of packets for this period a variable labelled bad is incremented. This information is written to file as depicted in Table 3-Packets 65 - 85 seconds, which is a snapshot of this information from 65 seconds to 85 seconds with an interval set to 5 seconds (with four traffic flows and 80 nodes).

time	AODV S	AODV R	AODV D	DSDV S	DSDV R	DSDV D	Swi S	Swi R	Swi D
65	2797	380	456	3196	1078	758	1747	380	574
70	3127	2403	436	4366	1913	754	4318	1895	572
75	3977	3745	978	4363	1893	365	3977	3745	374
80	3765	3080	875	4228	1809	487	3764	3080	752
85	4038	742	387	4345	1864	455	4225	1490	356

Table 3- Packets 65 - 85 seconds

After each interval, the interval based on the parameter that was passed to this method, update the running totals for the number of received packets for DSDV, AODV and dynamic switching. The dsdvS, dsdvR, dsdvD, aodvS, aodvR, aodvD counters are also reset to be used for the next interval.

The method concludes by outputting to the terminal an overview based on the running totals. If the total number of packets received by dynamically switching protocols is higher than the running totals for DSDV and AODV for this simulation run know that dynamically switching routing protocol outperformed both AODV and DSDV and network performance increased. If the total number of received packets by the dynamically switching of routing protocols is equal to or greater than both the DSDV or the AODV total, switching has performed at the same rate as the better performing protocol. If the number of received packets is less than the better performing protocol but greater than the least performing protocol performance is good, but could be better.

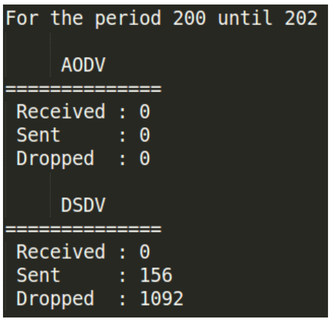
If the dynamic switching protocol is not performing as well as either the DSDV routing protocol or the AODV routing protocol the management node has in this case made a bad decision in its course of action.

The **`PrintAllPacketStats`** (p. 542) method accepts a time period to check all packet statistics for. The time period is determined by specifying a start time (fromTime) and a finish time (untilTime). An iterator called vectorIndex is initialised to the end of the data structure called allPackets. Next counters are initialised to zero to record how many packets were received, sent and dropped by the routing protocols in use; these are dsdvReceived, dsdvSent, dsdvDropped, aodvReceived, aodvSent and aodvDropped.

The allPackets data structure is stepped through until the beginning of the data structure is reached or until the loop is broken (explained subsequently).

An inner loop is created to step through the data structure until a finish time specified by untilTime is reached – packets that were received after the untilTime are of no interest. When this period is reached the inner loop completes but the outer loop continues.

The time recorded in the data structure is compared with the fromTime passed to the parameter list. If the time recorded in the data structure is less than the fromTime passed the outer loop ceases and statistics are displayed to screen for that period, this output is depicted in Figure 55 - Packet Statistics for a given period in time.



```
For the period 200 until 202

    AODV
==============
 Received : 0
 Sent     : 0
 Dropped  : 0

    DSDV
==============
 Received : 0
 Sent     : 156
 Dropped  : 1092
```

Figure 31 - Packet Statistics for a given period in time

Each packet that is checked within the given time period could have been recorded by any of the networks configured, for example a network that causes congestion, the network operating on the DSDV routing protocol or the network operating on the AODV routing protocol. The data packets that are using either DSDV or the AODV routing protocol are the packets of interest (control packets have already been filtered out). Therefore, for the packet currently being checked the protocol is obtained by calling the **GetProtocol (p. 483)** method and passing the destination IP address for the packet being checked. The **GetProtocol (p. 483)** method returns a string value of either “aodv” or “dsdv”. Each packet that is recorded is flagged with ‘S’, ‘D’, or ‘R’, which allows the differentiation between sent, dropped and received packets. If the protocol returned is aodv and the flag is set to ‘S’ aodvSent is incremented by one, if the flag is set to ‘D’ aodvDropped is incremented by one. If the flag is set to ‘R’ aodvReceived is incremented by one. The same is done for the dsdv protocol and dsdvSent, dsdvReceived or dsdvDropped using the flag method just described.

7.4 Energy Models

As mentioned previously the nodes within a mobile ad-hoc network are normally powered by a self-contained energy source, i.e. a battery. NS3 features a basic energy source implementation that allows the installation of one or more energy sources onto each node. The initial energy source is set to that of a lithium-ion battery with a maximum charge of 580 Kilowatt hour (kWh). In order to correctly model draw on the energy source the WIFI radio energy model that is included with NS3 is used. This model has four states, transmitting, receiving, idle and sleep. This allows the draw of power from the energy source when data is transmitted or received. The default values for power consumption are based on the CC2420 radio chip with a supply voltage of 2.5V and currents at 17.4mA for transmitting, 18.8mA for receiving, 20uA in sleep mode and 426uA in idle mode. According to Hyunwoo et al this model allows researchers to accurately model power consumption in a wireless data communications network (Hyunwoo et al, 2009).

A method that allowed the remaining energy of a particular node to be obtained was also created and is discussed when the management node and further cognitive features are discussed.

The **`SetBatteries (p. 529)`** method creates an energy source container with an initial value. The source container is populated with one or more energy sources (in this case one energy source). The energy sources within the energy container are then aggregated to a node.

Each node has its own energy source container with one or more energy sources in that container aggregated to that node. The energy source container does not contain energy sources that are used by more than one node.

To keep track of the energy sources that are aggregated to the nodes an aodvEnergySources container is populated to hold the addresses of all the energy sources.

A new container is created called aodvDeviceEnergyModel that allows the use of an energy model; in this case the Wi-Fi radio energy model is used, the TxCurrentA is set to a double value of 0.0174. This energy model is then aggregated to the aodvDevices and the aodvEnergySources and holds the memory addresses for all the energy sources that are using this model.

The **`RemainingEnergy (p. 530)`** method also creates trace sinks that trigger a method when an event happens. In the case when the energy changes it will call the **`RemainingEnergy`** (p. 530) method and the **`TotalEnergy`** (p. 530) method.

The **`TotalEnergy (p. 530)`** method prints to screen the remaining energy of a given node. The given node is defined in a callback method and is called when an event occurs, for example when a packet is sent.

This method prints the total energy consumed by a given node. The given node is defined in a callback method and is called when an event occurs, for example when a packet is sent.

The **GetPredictedTimePowerUntil (p. 530)** method will accept a node ID and return a predicted time of how long the energy source(s) attached to that node will last.

The method begins by calling the **GetRemainingEnergy** (p. 531) method and storing the current remaining energy in a local variable called remainingEnergy. Next, the current simulator time is obtained by calling the **GetTimeNow** (p. 442) method and stored in a variable called elapsedTime. A third local variable called usedEnergy is initialised and set to a value derived by subtracting the remaining energy value from the initial energy value. How much energy the node is using per second is calculated by dividing the used energy value by the elapsed time, the calculated value is stored in energyUsedPerSecond. Finally, the predicted remaining time for this energy source (how long the nodes battery will last) is calculated by dividing the remaining energy by the energy used per second. This gives an approximation in seconds of how long the node will be able to function.

Three functions are used which are **GetHoursFromSeconds** (p. 442), **GetMinutesFromSeconds** (p. 442) and **GetSecondsFromSeconds** (p. 443) to display on screen the time before predicted power depletion of the nodes energy source(s) in a time format, i.e. hours, minutes and seconds.

The function returns the total number of seconds that the node has before power depletion to the invoking instruction.

The **GetRemainingEnergy (p. 531)** method accepts a node identifier that is used to get the energy source(s) associated with that node. Once the energy source(s) are located for the node the **GetRemainingEnergy (p. 531)** method for that class is called to get the remaining energy. This value is returned to the invoking instruction.

In order to do this two iterators are created, the first iterator points to the start of the sources container (a container that holds the pointers to the energy sources) and this is called sourceBegin. The second iterator points to the end of the same container.

All the pointers in the source container are stepped through in order to locate the correct energy source pointer for the node identifier that was passed to the parameter list for this method. To achieve this a pointer to an energy source object called myEnergySource is created. The value assigned to this pointer is the value in iterator sourceBegin. Next, a pointer to a node object is created and a copy of the pointer for the node that is using the energy source pointer just obtained. This gives the node pointer for the current energy source being checked. The node pointer is used to invoke a method to get the identifier assigned to that node pointer. This is compared against the node identifier that was passed to the method. If there is a match the **GetRemainingEnergy** (p. 531) method is called, this method is part of the energy source class and returns the value to the method that invoked this method. If there was not a match sourceBegin iterator is incremented by one so that it points to the next pointer in the energy source container. This process is repeated until a match is found.

The **SetRemainingEnergy (p. 532)** method is used to set the remaining energy equal to that of its paired node in the other container. This method is called when the routing protocol is switched to ensure that the energy levels of the corresponding nodes are operating at the same level.

The method begins by comparing the current protocol against a string value. If the protocol is equal to AODV then the protocol has just switched from DSDV, therefore the node energy that is operating on the AODV protocol is set equal to the same node identifier from the

DSDV container. If the current protocol is DSDV then before the switching of routing protocol AODV was being used. In this case the current energy level of the DSDV node is set to that of its corresponding node from the AODV container.

This method will set the remaining energy for a node in the AODV node container to the same value with the same node identifier of the DSDV container.

It works by creating a pointer to an energy source that is attached to the AODV node id that is passed to the method. It also creates a pointer to an energy source that is attached to the corresponding node in the DSDV container.

The value that represents the remaining energy of the energy source attached to the DSDV node is obtained and this value is used to set the remaining energy of energy source for the AODV node.

This method is not called directly but from the **`SetRemainingEnergy`** (p. 532) method.

The **`WillBeAlive`** (p. 533) method returns true if the current energy source for the given node identifier that was passed is predicted to have enough energy for transmission/reception at the next point that the **`management`** (497) method is called. The method begins by getting the remaining energy of the energy source that is attached to a node (based on the node identifier that was passed) by calling the **`GetRemainingEnergy`** (p. 531) method. The **`GetRemainingEnergy`** (p. 531) method returns a double value which represents the current energy left for this node; this value is stored in remainingEnergy. Then a variable called elapsedTime is created to store the current time by calling the **`GetTimeNow`** (p. 442) method. How much energy has been used is calculated by subtracting the remainingEnergy value from the initialEnergy value. Energy consumption per second is calculated by dividing usedEnergy by the elapsedTime. To get how much power is required until the next call to the `Management` (p. 497) method the energy consumption per second is multiplied by the time remaining until the next call. RemainingEnergy > powerNeeded is returned which will result in the return value of true if the remaining energy is greater than the power that is required. Otherwise false value will be returned.

The `RemoveDepleatedNodes` (p. 534) method creates two node containers called deadNodes and liveNodes. The remaining energy level of each node is checked in turn by calling the `GetRemainingEnergy` (p. 531) method. If the node currently being checked has not depleted its energy source a copy of the pointer of that node is copied into the liveNodes container. Otherwise, the pointer of that node is copied into the deadNodes container.

7.5 Logging

7.5.1 Logging Receiving Packets Information

In this section other packet information is discussed. In the `ScheduleEvents` (p. 441) method several call back instructions are created in order to make a call back to a function of choice when certain actions or events occur. For example the following instruction creates a call back that allows the capturing of when a packet has been transmitted from the physical layer of a Wi-Fi network device.

```
Config::ConnectWithoutContext("/NodeList/*/DeviceList/*/$ns3::WifiNe
tDevice/Phy/PhyTxBegin", MakeCallback(&CogMan::SendPacket, this));
```

Rather than create separate call backs for each node the Greek asterisk symbol meaning ‘little star’ is used. In NS3 (and many other computer science applications) the asterisk is a wildcard and means all. Therefore callback is connected without any context to the `PhyTxBegin` method, each time this method is called a callback is invoked which results in one of the methods being called. The method that is called is **`SendPacket`** (p. 481). A pointer to the packet is passed into the parameter list of the **`SendPacket`** (481) method.

The **`GetCurrentReceivedPacketsForDestinationFromSource (p. 486)`** method returns the number of packets received for a destination node k^d (dNode) that were sent from a specific source node k^s (sNode) for a given duration (timeFrom → timeUntil).

All packets received are stored in a dynamic data structure called packetReceivedByCurrentProtocol with the time the packet was received, the IP address of the node where the packet originated and the IP address of the node that received the packet (not intermediate nodes). Therefore before searching the data structure for the correct source and destination the IP addresses must be obtained for the source node identifier and the destination node identifier. This is achieved by calling **`GetIPAddressFromNodeId`** (p. 537) for each identifier.

The data structure (which is sorted by time) is searched for appropriate packets. When an entry that contains the source node IP address, the destination node IP address and the time that the packet was received is within the given time period is located, a counter called pr is incremented by one. The searching of the data structure completes when a packet is encountered that has been sent earlier than the fromTime or when the beginning of the data structure is reached (whichever comes first).

The value stored in pr is returned to the invoking method.

The **`GetCurrentReceivedPacketsForSource`** (p. 486) method is used to get the number of packets received at the destination(s) that was sent by the source node identifier passed to this method. The method also accepts a fromTime and a timeUntil that allows a period to be specified in order to get the number of packets received at the destinations. This allows for performance analysis when actions have been issued by the management node. Global variables were used initially to keep track of the source and destination but is became apparent quickly that there could be multiple sources and multiple destinations for each source.

Therefore, a dynamic data structure that is used to record all packets received by every destination that is transmitted by the current routing protocol being used is stepped through. The number of packets that were received and that were sent by the source is tallied, by initializing a counter called pr to 0 and incrementing by one each time. The data structure is stepped through starting at the end. Each packet that is encountered the time that the packet was received is checked. If the packet was received before the value passed in the timeUntil parameter list, a check is conducted to ensure that the packet was indeed sent by the source node that is of interest, this is achieved by checking the recorded source IP address against the IP address of the node identifier that was passed. The IP address of the source node is obtained by calling the **`GetIPAddressFromNodeId`** (p. 537) method; the return value from this method is stored in a local Ipv4Address structure called nodeIP. If the IP address of the source node matches the IP address of the node identifier that was passed to this method the time that the packet was received is checked to establish if it was before the timeUntil value that was passed. If this is the case, the packet being checked was received during the period specified for the node of interest and the pr variable is incremented by one. If not the next packet in the packetReceivedByCurrentProtocol data structure is checked. Each entry in the data structure is checked until an entry in the structure is outside of the time period that is being checked or when the start of the data structure is reached. The value of pr is returned to the method that invoked this method.

7.5.2 Logging Dropping Packets Information

The `GetCurrentDroppedPacketsFromSource` (p. 489) method takes a node identifier (nodeId) into the parameter list. This identifier is used to get the IP address of the source node of interest by calling the `GetIPAddressFromNodeId` (p. 537) and pass this method the node identifier. The return value is stored in a local object Ipv4Address object called nodeIP.

Next an iterator called vectorIndex is created to step through the packetDroppedByCurrentProtocol data structure. The vectorIndex is set to point to the end of the data structure.

When stepping through the data structure IP address of the source node in the data structure is checked against the IP address of the node identifier that was passed to this method for a match. If there is a match, the time that the packet was received is checked to establish if it was before the timeUntil value that was passed to the `GetCurrentDroppedPacketsForSource` (p. 489). If the packet being checked was received during the period specified for the node of interest the pd variable is incremented by one. The next packet in the `packetDroppedByCurrentProtocol` data structure is checked. The data structure is stepped through until a point in the structure that is outside of the time period that is being checked is reached or when the start of the data structure is reached. The value of pd is returned to the method that invoked this method.

7.5.3 Other Logging

The `ScheduleEvents(p. 441)` method contains the code to schedule events within the simulator. This method mostly calls on other methods to perform actions relating to the setup, configuration and operation of the network. For example, methods to: create nodes, assign mobility models, set the default packet size and data rate, create the traffic flows, etc.

This method also calls on the FlowMonitor class in order to keep track of all the packets and data flows within the network. This information is written to disk when the simulation ends.

The Network Animator class that creates a file that can be run in the Network Animator visualization tool is also used.

Towards the end of the `ScheduleEvents (p. 441)` method the simulation stop time in seconds is scheduled. Immediately after the scheduling the simulation is started. When the simulation finishes various data files are written to disk such as received packet information, dropped packet information and sent packet information. Also written to disk are the dynamic systemState data structure so that when decisions were made by the management node within the network can be reviewed, analysed and if the actions made any change in regards to network performance.

The folder for the trace files is automatically created by this method based on the folder name that is passed from the command prompt if the folder does not already exist. This allows the running of several simulations and saves the data to be analysed to different folders of choice without having to alter the code. It was imperative to introduce a batch system to manage the runs as each simulation run required several hours to complete and was often left unattended. An example of a batch file is:

```
mkdir 4_20R
./waf --run "final_5
    --numOfNodes=20
    --numOfTrafficFlows=4
    --numOfCNodes=0
    --dataRate=11Mbps
    --cognition=false
    --simulationTime=600
    --radius=100
    --logFile=T1
    --mobilityModel=R
    --initialProtocol=aodv
    --cNodeSpeed=1.8
    --nodeSpeed=1.8
    --recordPositionInterval=5
    --folderName=4_20R
" > 4_20R/output.txt 2>&1
```

When any method or function ends the flow of control is returned to the place at which the method or function was invoked (called). Therefore when this function completes program control returns to the main function in order to execute the next instruction.

The setLogging method allows the turning on or off of certain logging components such as when methods are entered or left, output variable data, packet drop information, packet sent information, packet received information, information relating to the route a packet has taken, distance between node information, device information, IP address information, traffic flow information, election process information, node destination information, node position information, energy source information, distance information from a given node to all other nodes, distance information for a given node for all other nodes in range of the source, routing information and node information. Implementing this method was imperative because of the mass amount of information generated by each of the simulation runs. For example with five of the logging components switched on and thirteen switched off one of the output files generated was in excess of 1.7Gig and consisted of forty six million, three hundred and seventy five thousand and twelve lines.

The final purpose of the setLogging function is to allow for debugging in steps.

The ConfigureNetAnim method allows the changing of certain parameters for Network Animator. This method is used primarily in assigning a different colour for each group, for example the Eagles group are assigned the colour blue and the Sharks group the colour green. This method is also used to differentiate between AODV nodes, DSDV nodes and Controllable nodes.

The **`DoStats (p. 538)`** method is called when the simulation ends. The method outputs to screen the total statistics for each routing protocol. The outputs for AODV are the total payload packets sent, the total payload data dropped, the total control data sent, the total control data dropped and the total payload data received. The same information is printed for the Dsdv protocol.

The **`PrintNodeInformation`** was mainly used during the testing phase and allowed the ability to output information to the terminal for the node pointer passed the following: the global node id, the number of applications, the number of network devices, the number of network interfaces and the local IP address for each interface.

7.6 Chapter Summary

In this chapter the node hardware for network communication were discussed. This consisted of discussions relating to the physical layer and the mac layer of the ISO model. Particular focus was given to the characteristics of the antenna and how the configuration of the antenna used in this research modelled conventions. The transmission radius was discussed including the configuration required for a radius of up to 21 metres and a radius of up to 100 metres.

Next the discussion focussed on the protocol stack and how an interface was created. The discussion detailed how the IP addresses were split into sub networks and assigned to each of the nodes.

The routing protocols AODV and DSDV were discussed and how they were implemented onto the protocol stack. Next operating frequency (channels) was discussed.

After the configuration of the physical and mac layers emphasis shifted to traffic flows and section 7.2 Traffic flows detailed how sources and sinks were selected. The discussion detailed how initially one source would transmit to one destination but with later configurations, multiple sources with multiple destinations.

In section 7.3 Measuring Performance, it was explained how received packets are the primary metric to measure the performance of the network. This explained how packets were sent, received, forwarded and dropped and how this information was used in the operation of the network.

In the final sections of the chapter it was explained how all packets are logged and how this information is used to provide statistics that relate to the overall operation of the network.

All the required components of the MANET testbed have been presented; designed, created and implemented; the testbed was tested by simulations. On each step of implementation the code was checked to ensure it produced correct and replicable outcomes. Together with chapters 5, 6, chapter 7 proves to meet Objective 2 (To design a suitable testbed, whilst ensuring that the solution is rigorous, transparent, and replicable for the testing of the scientific theories). The next step is now related to the design and implementation of cognitive attributes to the MANET.

Chapter 8 Implementation of the Cognitive Application

This chapter progresses on the design and implementation of a MANET testbed, started in chapter 5. The aim is to achieve Objective 3 (To enhance the design of the MANET to incorporate cognitive attributes) and get a testbed ready. The chapter begins by reminding the reader of the admissible cognitive actions (Section 8.1).

Discussed next is how the novel idea has been implemented and discussion includes the Manager Node Election process (Section 8.2). The state of the Network is determined in section 8.3. The implemented roles associated with the Management node are discussed in section 8.4 and discusses how the management node determines a course of action, requests the action, record the request, records outcomes as a result of the request being authorised. Also discussed is how the Management node uses its memory to consider previous actions and perform reflections.

In section 8.5 discussion turns to the role of the Network Command Centre beginning with its main functionality, how stabilisation is implement, how actions are granted or denied and how a suitable controllable node is located and used.

Section 8.6 discusses the data structures that were used for memory and cognition. Section 8.7 discusses the data that is recorded and then the chapter concludes with a summary (section 8.8).

8.1 Admissible Cognitive Actions

There are three possible outputs depending on the plan formulated by the management node. One of which has already been discussed in Section 6.3.6 Mobility of Controlled nodes.

The other two actions are the ability to switch routing protocol and the second is to switch operating frequency. First to be discussed is the changing of routing protocol, then reassignment of channel or spectrum frequency.

8.1.1 Switching Protocols

The `SwitchProtocol (p. 520)` method first checks to see which routing protocol the network is operating on. If the network is operating using the AODV routing protocol the network will switch to the DSDV routing protocol. If the network is operating using the DSDV routing protocol it will switch to the AODV routing protocol.

8.1.2 Changing Channel

The `SetChannel` (p. 534) method allows the changing of the current frequency that the node of interest is operating on. First, the frequency for the node is needed to be obtained; this is achieved by invoking the `GetChannel` (p. 534) method. This allows a node the ability to switch to another channel which is directly adjacent to the current channel, because there are several nodes participating in the network; all the channels for these nodes are switched, including the controlled nodes.

8.2 Manager Node Election

The section details how the design from Chapter 5 was implemented. Already discussed is that all nodes in direct communication range of the source node will be candidates, also discussed is that in order to choose the most suitable candidate energy levels are considered to predict how long a node will be alive, velocity to predict how long a node will be in range of the source node and density. The following details the implementation of the theory discussed in Chapter 5.

8.2.1 Distances to other nodes: who is in range

Each source node's identification number (ID) is denoted by k^s; at time t_0 the **`GetAllDistancesFromSource`** (p. 465) method is invoked which populates a vector called nodesInMyWorld, this vector contains the distance from the source node to each and every other node in the simulation: $\{d(k, k^s, t_0)\}_0^N$, where $k \in \{0,1, \dots, N\}$ is not equal to k^s. First the source node position is obtained by invoking the **`GetNodePosition`** (p. 464) method. Next, the node position of a node that is not this source node is obtained by calling the **`GetNodePosition`** (p. 464) method. Using this position, and the position of the source node the distance between the two nodes is obtained by invoking the **`GetDistance`** (p. 465) method. Finally, if the two nodes are in range, the source node identifier, the destination node identifier, the distance between the two nodes and the current simulation time is stored in a vector called nodesInMyWorld. The range is checked by calling the **`InRange`** (p. 468) method, i.e. the nodes of interest are those that are $d(k, k^s, t_0) \quad \leq r_{tr}$, the number of nodes $n(k^s, t_0)$ that are in range of the source node k^s at current time t_0 is calculated.

The **`CreateVectorOfDistancesAllNodes (p. 466)`** method works very similarly to the **`GetAllDistancesFromSource`** (p. 465) method. With the exception that it does not accept a source node identifier. Instead this method populates a data structure called allNodesTheWorld. For each node, every other node identifier, the distance to that node and the current simulation time is recorded.

The **`GetDensityFromNode`** (p. 466) method returns a density reference for a given node identifier. The number of nodes within the range of the node could not simply be returned because different nodes may be able to transmit at different distances. Consider for example a node that transmits at a radius of 100m and there are five nodes within the transmission range. Now consider another source node that is able to transmit at a radius of

200m and there are five nodes within transmission range. Returning the value five for both cases does not give the same density. The node that transmits at 100m with five nodes in its transmission range has a much higher density that that of a node that transmits at 200m with the same number of nodes within transmission range. Therefore to calculate the density of the network in the transmission area of k^s at time t_0:

$$\rho(k^s, t_0) = \frac{\mathrm{n}(k^s, t_0)}{\pi\, \mathrm{r}_{tr}{}^2}$$

Where $\rho(k^s, t_0)$ is the density reference, $\mathrm{n}(k^s, t_0)$ is the total number of nodes within transmission range of the given node and r_{tr} is the transmission radius.

Before obtaining the number of nodes $\mathrm{n}(k^s, t_0)$ in transmission range, the routing protocol that is currently being used is established. If the routing protocol is AODV a vector index is created and set to the start address of the aodvNodes container $A(N)$. If the routing protocol is DSDV a vector index is created and set to the start address of the dsdvNodes container $D(N)$.

All the nodes in the container are then stepped through and the distance obtained from the source node to the current node that is being checked in the container (with the exception of the source node) by calling the **`GetDistance`** (p. 465) method. The **`InRange`** (p. 468) method is called to check if the distance is within the current transmission radius (in range of the source node), if the node is in range the variable totalInRange is incremented by one.

The transmission radius r_{tr} of the node is defined when the class constructor is called. The default value set is a radius up to 100 meters (see chapter 7 for details), although this may change depending on the type of node object.

Constant π is called PI and is set to the value 3.14159265359.

8.2.2 Number of Neighbours

Once a density reference has been obtained an election is performed by calling the **`Election`** (p. 492) method to determine which node will become a management node and have the cognitive attributes that allows the node to observe the network and make decisions.

The **`InRange`** (p. 468) method accepts a parameter that specifies the distance between two nodes. The evaluated distance is compared with the radius and returns a Boolean value of either true or false. True if the distance is within the transmitting range.

$$\text{inRange} = \begin{cases} true, & if \quad d(k, k^s, t_0) \ \leq r_{tr} \\ false, & \qquad\qquad\qquad else \end{cases}$$

The **`GetPredictiedMaximalTimeInRange`** (p. 468) method accepts two integer values into the parameter list and stores these values in local integer variables sourceNodeId and destinationNodeId respectively.

The function creates a vector called sNodePredictedPosition that is to be used to store the predicted position $(\tilde{x}(k,t), \tilde{y}(k,t))$ of a given node k at a future time $t > t_0$ (in this case the source node that is identified by the value that was passed to sourceNodeId variable). This process is repeated for the destination node and stored in a vector called dNodePredictedPosition.

The method for obtaining a future node position is discussed in chapter 5.

Next, a loop begins and one second is added to the value that currently resides in futureTime. The value of the sNodePredictedPosition vector is set to a value returned by the

`PredictPosition` (p. 469) method (the future time value and the source node ID is passed to this method). The same process is completed for the destination node, the same future time value and the destination node ID are passed, and the return value is stored in a vector called dNodePredictedPosition. The loop condition is then evaluated because a “do while” loop is used rather than the more commonly used while loop; this allows the condition to be evaluated at the end of the loop rather than before the loop.

The while conditional statement is evaluated by calling the method **`GetDistance`** (p. 465) and passing the vectors sNodePredictedPosition and dNodePredictedPosition. The value returned from **`GetDistance`** (p. 465) is then used to evaluate if the nodes are in range or not. This is achieved by using the value derived from **`GetDistance`** (p. 465) as the parameter for the **`InRange`** (p. 468) method. The InRange method will return the value true if the nodes are still in range or false otherwise. If the nodes are in range the process is repeated for a further one second in to the future.

The **`WillBeInRange`** (p. 535) method returns true if two nodes will be in predicted range of each other at some time in the future. First the method creates two vectors one called sNodePredictedPosition and the other called mNodePredictedPosition. Next called is the **`PredictPosition`** (p. 469) method for the first node identifier that was passed to the WillBeInRange method with the time that the predicted position is required. The **`PredictPostion`** (p. 469) method returns a vector that contains x, y and z co-ordinates which is copied into sNodePredictedPosition. The same process occurs for the second node identifier that was passed and the return value from **`PredictPostion`** (p. 469) is stored in the vector mNodePredictedPosition.

Once the predicted positions of the two nodes at a given time in the future is obtained, the predicted distance between them can be established by calling the **`GetDistance`** (p. 465) method and passing the predicted positions of the source node and the management node, the return value from **`GetDistance`** (p. 465) is stored in a local variable called distance. Finally, the **`InRange`** (p. 468) method is called and the distance is passed to this method. The **`InRange`** (p. 468) method will return true if the nodes will be in range of each other (based on distance passed) or false otherwise – this return value is then returned to the method that invoked this one.

8.2.3 Will Be Transmitting

The purpose of the **`Election`** (p. 492) method is to select a suitable management node for each source node that is transmitting data across the data communications network. The **`Election`** method begins by obtaining the current time by calling the **`GetTimeNow`** (p. 442) method and storing the returned value in the variable timeNow. Then a vector index is created and set to the value that indicates the beginning of the myTrafficFlows container.

For each node in the myTrafficFlows container a check is performed to establish if the node is transmitting at this current time. If a given node is transmitting the current position of that node is obtained by calling the **`GetNodePosition`** (p. 464) method.

Next, each other node in the network is checked to see if it is in range of the source node. If the node being checked is in range, the predicted time period that the node will be in range of the source node is obtained. Next, the predicted power remaining in the energy source for the node being checked is obtained. In addition, the density for the node being checked is obtained. This is achieved by calling the methods: **`GetPredictedMaximalTimeInRange`** (p.

468), **GetPredictedTimePowerUntil** (p. 530) and **GetDensityFromNode** (p. 466) respectively.

Once the values associated with the predicted maximal time in range, the predicted time the node will have power for and the density for the node have been obtained they are recorded to a data structure with the node ID by calling the **RecordElectionCandidate** (p. 493) method.

The total number of candidates is checked, if this number is greater than 0, a method to select a suitable candidate from the candidate list is called. If there were no suitable candidates the election is scheduled to run again at some time in the future.

8.2.4 Candidates List

The **RecordElectionCandidate** (p. 493) method is used to keep track of nodes that are suitable candidates of the election process. One of these nodes will be chosen as a management node and have the cognition attributes installed. The method accepts a source node identifier (the node that is transmitting), a candidate node identifier, the time that this node became a candidate, how long the candidate node will be in range of the source node, how much time that candidate will have power for and the density for the candidate node. All neighbouring nodes of the source are initially candidates.

In order to store the required data a dynamic data structure called candidatesList is created and the required values pushed into the list. The information that was recorded and how many candidates are currently in the list are output to screen. This is for informational purposes.

The **`RecordElectionCandidatePercents`** (p. 494) method works exactly as the **`RecordElectionCandidate`** (p. 493) method. The exception is that percentages are stored rather than raw values.

Less suitable nodes that are in range of the source node should be eliminated. The process begins by creating variables that store maximum values for the candidates, the maximum values are used later when calculating which node is the most suitable. The variables used are maxTimeInRange, maxTimeInPower and maxDensity – each of these variables are initialised to the value 0.

Next a vector index is created and set to the start of the candidateList vector. This vector is stepped through processing each and every node in the list one by one. For each node in the list is must be ensured that a neighbour of the source node that was passed to this method is being processed, therefore an equality operator is used. If the candidate node that is being checked is in range of the source node, the time that the candidate underwent the candidate election process is obtained. If this value is equal to the value that was passed when calling this function the value for how long this node will be in range of the source node is obtained. If this value is higher than the value stored in maxTimeInRange the value contained in maxTimeInRange is updated by setting it equal to the value obtained from the candidate node. A similar process is undertaken for maxTimeInPower and maxDensity.

```
  while (vectorIndex < candidatesList.end() )
  {
    if (vectorIndex->sourceNodeId == sNode &&
vectorIndex->timeElected == timeNow)
    {
      if (maxTimeInRange < vectorIndex->timeInRangeUntil)
      {
        maxTimeInRange = vectorIndex->timeInRangeUntil;
      }
      if (maxTimeInPower < vectorIndex->timePowerUntil)
      {
        maxTimeInPower = vectorIndex->timePowerUntil;
      }
      if (maxDensity < vectorIndex->density)
      {
        maxDensity = vectorIndex->density;
      }
    }
    vectorIndex++;
  }
```

Now that the maximum values for the correct candidates at the correct time for the correct source have been established, a scoring process to identify the most suitable node to undertake the management duties can begin.

More local variables are set: highestScore, likelyManager, timeInRangeUntilPercents, timeInPowerPercents, densityPercents, timeInRangeUntil, timeInPowerUntil and density. All the variables are initialised to 0 with the exception of likelyManger which is initialize to -1.

The vector index is reset so that it points back to the start of the candidate list and for each node that is in range of the source node and is elected at the correct time is scored based on the following formula.

$$p_1 = \frac{100\ t_1}{m_1}$$

Where p_1 is the timeInRangeUntilPercents, t_1 is the timeInRange until and m_1 is the maxTimeInRange.

A similar formula to work to obtain the timePowerUntilPercents is applied. Where p_2 is the timeInPowerUntilPercents, t_2 is the timeInPowerUntilRange until and m_2 is the maxTimeInPower.

$$p_2 = \frac{100\ t_2}{m_2}$$

A similar formula to work to obtain the densityPercents is applied. Where p_3 is the densityPercents, ρ is density until and m_3 is the maxDensity.

$$p_3 = \frac{100\ \rho}{m_3}$$

Once the values as a percentage out of 100 have been obtained, a weighting system to give each node an overall score is applied.

$$s = (p_1 \times w_1) + (p_2 \times w_2) + (p_3 \times w_3)$$

Where s is the score, w_1 is the weight applied to range, w_2 is the weight applied to power and w_3 is applied to density.

If the current score for the node is higher than the current value in highestScore, highestScore is updated to reflect the new highest score. In addition to this the candidate node ID is copied into the variable likelyManager, timeInRangeUntil into timeInRangeUntil, timePowerUntil into timePowerUntil and density in to density. The values are copied from the vector into local variables.

Once every node has been checked, the source node identifier and the values contained in likelyManager, timeNow, timeInRangeUntil, timePowerUntil and density are passed to the **`RecordElectionWinner`** (p. 497) method to record this candidate as the winner of the election and to assign as a management node.

The **`RecordElectionWinner`** (p. 497) method is used to keep track of the management nodes. The method accepts a source node identifier (the node that is transmitting), a manager node identifier, the time at which point the manager will be in range of the source and the remaining power of the management node. These values are stored in a vector called managersList and is of object type managerNode.

The method also accesses the last entry in the vector and outputs to screen the manager node identifier, what time it was elected, how long it will be in range for, the current density for informational purposes.

The final purpose of the **`RecordElectionWinner`** (p. 497) is to invoke the **`Management`** (p. 497) method and pass the values for the source node and the management node.

Relationship Status

The **`Management`** (p. 497) method is where the cognition is applied to the management node. Various steps are undertaken that first check the viability of the relationship between the management node and the source node. If the relationship is not viable the re-election process is performed so that the source node can be assigned a new management node. This should be completed within a timely fashion, if not all previous knowledge is lost.

A check is performed to establish if the source is still transmitting by calling the **`StillTransmitting`** (p. 476) method and passing the identifier of the source node. If the source node has finished transmitting then the **`Management`** (p. 497) method is not called any longer for this node – unless it begins transmitting at a later date.

If the source node is still transmitting, the current simulation time is obtained by calling the **`GetTimeNow`** (p. 442) method. The time is used to predict if the management node will be in range of the source node at a given time in the future, this is achieved by calling the **`WillBeInRange`** (p. 535) method and passing the source node identifier, the management node identifier, the current time t_0 added to a time value that reflects the time at the end of the next interval that the management method will be called. If it is predicted that the source node will not be in range of the management node the election process is performed again to choose a more suitable management node.

If the source node will be in range of the management node at the time of the next checking interval t_1 the battery power of the energy source(s) for the management node is checked. This is achieved by calling the **`WillBeAlive`** (p. 533) method and passing the management node identifier and the next time t_1 that the **`Management`** (p. 497) method will be called. If the battery level has almost depleted and the management node is unlikely to have enough power to transmit or receive radio waves at t_1, the election process is performed again.

Providing the management node will have enough power to continue its role and providing the management node will be in range of the source node, the source node identifier, the management node identifier, the current range from the source node to the management node, the current energy level of the energy source(s) for the management node and the current time is saved to a dynamic data structure. This information is saved by calling the **`RecordRelationshipState`** (p. 520) method and passing the source node identifier, the management node identifier and the current simulation time. This information allows more accurate predictions on the relationship status of the management node and the source node.

Once the management node has established that the relationship between the source node and the management node is viable the state of the network is checked.

The **`RecordRelationshipState`** (p. 520) method accepts a source node and a management node identifier. The remaining energy of the sNode is obtained by calling the

GetRemainingEnergy (p. 531) method and storing the return value in a local variable called energyLevel. The current time is obtained by calling the **GetTimeNow** (p. 442) method and the value returned is stored in a local variable called timeNow. Next, the node position of the source node is obtained and stored in a vector with x, y co-ordinates in a local variable called locSNode by calling the **GetNodePosition** (p. 464) method. Also, the management node position is obtained by calling the **GetNodePosition** (p. 464) and the return value is stored in a variable called locMNode. The final value obtained is the distance and is achieved by calling the **GetDistance** (p. 465) method and passing the source node and management node location vectors, the returned distance is stored in a variable called range. Those values are then pushed on to a dynamic data structure called **recordRelationshipState** (p. 520).

8.3 State of the Network

In order to make cognitive decisions, managers will have to check the current state of the network and to react appropriately if something changes.

At time now t_0 the number of packets $rp(k^s, t_{-1}, t_0)$ received from the source node k^s for the last period (t_{-1}, t_0) (the period is from when the management method was last called until now) is obtained by calling the method **GetCurrentReceivedPacketsForSource** (p. 487). The packets sent by the source $sp(k^s, t_{-1}, t_0)$ and the packets dropped $dp(k^s, t_{-1}, t_0)$ for the k^s-th source node for this period is obtained by calling the **GetCurrentSentPacketsForSource** (p. 488) method and by calling the **GetCurrentDroppedPacketsForSource** (p. 489) method respectively. The return values from the three methods called are copied into local variables packetsReceived, packetsSent and packetsDropped respectively.

The receiving rate per second is then calculated for the destination node(s)

$$rr(k^s, t_0) = \frac{rp(k^s, t_{-1}, t_0)}{t_0 - t_{-1}}$$

and stored in a variable called recPerSec where $rr(k^s, t_0)$ is the receiving rate of the destination node(s) for this source node k at time t_0. Similar calculations are performed to derive the sending rate per second $sr(k^s, t_0)$ and the drop rate $dr(k^s, t_0)$ per second for this source:

$$dr(k^s, t_0) = \frac{dp(k^s, t_{-1}, t_0)}{t_0 - t_{-1}} \;;\; sr(k^s, t_0) = \frac{sp(k^s, t_{-1}, t_0)}{t_0 - t_{-1}}$$

Then, the previous rates (for $t = t_{-1}$) for the k^s-th source node: $sr(k^s, t_{-1}), dr(k^s, t_{-1}), rr(k^s, t_{-1})$ are obtained, this is achieved by looping though a vector called packetRates. The search starts at the end of the vector and moves towards the beginning, each entry, searching for the appropriate source node (source=k^s). Once the correct source node is found, the received, sent and dropped rates with the time that the information was recorded is copied from the vector. This information is used shortly to calculate the current flow (in percents) rate between the source and the destination(s):

$$fr(k^s, t_0) = \frac{rr(k^s, t_{-1}, t_0)}{sr(k^s, t_{-1}, t_0)} * 100.$$

Just before this calculation the current packet rates $(sr(k^s, t_0), dr(k^s, t_0), rr(k^s, t_0))$ is recorded to the same vector that the previous packet rates was obtained $(sr(k^s, t_{-1}), dr(k^s, t_{-1}), rr(k^s, t_{-1}))$ from, the information recorded is: the current time (t_0), the source node identifier (k^s), the number of packets sent, received and dropped per second.

Before calculating the flow rate, the number of sent packets per second is checked to ensure that it is not equal to zero in order to avoid infinity errors. If the number of packets sent per second is not equal to zero a formula is applied to calculate the current flow rate per second which is the current received rate per second divided by the current send rate per second. The flow rate is stored in a variable called currentFlowRate $fr(k^s, t_0)$.

The same formula is used to acquired values for the previous flow rate $sr(k^s, t_{-1})$, $rr(k^s, t_{-1})$ which are stored in a variable called previousFlowRate ($fr(k^s, t_{-1})$).

At this stage the totals for the packets: received, sent and dropped for this checking period and the same information for the previous period has been obtained. This information is based on the current routing protocol that is in operation or in the case of the previous period the routing protocol that was in operation.

Also of interest is how the network would be behaving if an alternative routing protocol was being used; this is achieved by invoking the **`PrintAllPacketStats`** (p. 542) method and sending the information that relates to this period in time (the start time of the period and the time now). This information is not used in the cognition process because in a realistic scenario this information would not be available –it is however useful to quickly compare if the decisions made by the management node are having a positive or negative impact on the performance of the network.

The next check is to determine if the network is not in a state of fluctuation due to a prior action being initiated, for example, if the routing protocol had just changed, switched channel or moved a controllable node. In each of these cases the network it given some time to adjust to this change and to become stabilized. The amount of time that is allowed to stabilize is based on the action that was performed. The time to stabilize also guards against other competing management nodes putting into action their decisions when a prior decision has just occurred. All management nodes are instructed to back off until the time to stabilize period has passed.

If the network is not in a state of fluctuation the current flow rate is compared against the previous flow rate. Should the currentFlowRate be equal to zero or should the currentFlowRate be lower than the previousFlowRate, i.e. $fr(k^s, t_{-1})$> $fr(k^s, t_0)$, it can be assumed that network performance has decreased (already checked is that the source is still sending packets, that the relationship between source and destination is viable and that the network has stabilised). However, before the management node requests a change that could influence the network state another check is performed to determine if this degradation in performance is a momentarily fluctuation of if a trend is occurring and indeed the performance of the network is degrading. This is achieved by getting the previous previous sending rate ($sr(k^s, t_{-2})$) and the previous previous receiving rate ($rr(k^s, t_{-2})$). The difference between the previous previous sending rate ($sr(k^s, t_{-2}) - sr(k^s, t_{-1})$) and the previous previous receiving rate ($rr(k^s, t_{-2}) - rr(k^s, t_{-1})$) is calculated and stored in a variable called previousPreviousFlowRate. Should the previousPreviousFlowRate be higher than the previousFlowRate, i.e. $fr(k^s, t_{-2})$> $fr(k^s, t_{-1})$, then the rate of flow is decreasing and the reason as a momentarily fluctuation is dismissed. At this point the management node is allowed to consider its options, to better inform the manager least squares approximation is used to obtain a predicted receive rate if the trend of the network continues along this path.

The network will use the least squares approximation algorithm to get the predicted receive rate by using the following six values for the k-th source node

$fr(k^s,t_{-2})$	$fr(k^s,t_{-1})$	$fr(k^s,t_0)$
t_{-2}	t_{-1}	t_0

The equation of the best fit straight line is

$$fr(k^s,t) = a_k + b_k t,$$

$$E_K = (a_k + b_k(t_0) - fr(k^s,t_0))^2 + (a_k + b_k(t_{-1}) - fr(k^s,t_{-1}))^2 + (a_k + b_k(t_{-2}) - fr(k^s,t_{-2}))^2 \qquad \to min$$

The gradient of the best fitted straight line is

$$b_k = \frac{3\sum_{j=-2}^{0} fr(k^s,t_j)t_j - \sum_{j=-2}^{0} fr(k^s,t_j)\sum_{j=-2}^{0} t_j}{3\sum_{j=-2}^{0} {t_j}^2 - (\sum_{j=-2}^{0} t_j)^2}$$

And the shift is

$$a_k = \frac{\sum_{j=-2}^{0} fr(k^s,t_j)\sum_{j=-2}^{0} {t_j}^2 - \sum_{j=-2}^{0} fr(k^s,t_j)t_j\sum_{j=-2}^{0} t_j}{3\sum_{j=-2}^{0} {t_j}^2 - (\sum_{j=-2}^{0} t_j)^2}$$

Thus NCC is being passed a string of predicted values $\{fr(k^s,t_1)\}$, $k^s \in \{1,\ldots,N^S\}$, where N^S is the number of source nodes.

The result from the calculation above is stored in a variable called estimatedRecPerSec and is checked against a lower and higher boundary. If the estimatedRecPerSec value is a negative value, it is changed to the value to 0 (the lower boundary). If the estimatedRecPerSec value is greater than the maximum receiving rate, estimatedRecPerSec is set to the maximum receiving rate (the higher boundary).

Next, the density for the management node is obtained by calling the **`GetDensityFromNode`** (p. 466) method and the channel that the management node is operating on is obtained by calling the **`GetChannel`** (p. 534) method. An instinctive feeling (gutFeeling) for the action that the management node might request based on the current information available is then set (before the management node recalls on its experience and the outcomes of past events when there was a similar network state). The information used to set the instinctive feeling is based on current flow rate, previous flow rate, previous previous flow rate and the number of nodes in direct communication range of the manager node. The number of neighbours is obtained by calling the **`GetNumberOfNeighbours`** (p. 471) method. The instinctive feeling is set to C, M or P where C is the feeling to request to change frequency channel, M is the feeling to request control of a node and P is the feeling to request a change in routing protocol.

The management node is only allowed the opportunity to act upon their intuition if after consideration there is no other alternative. The management node considers alternative actions by calling the **`GetAction`** (p. 512) method.

Providing the management node has not been told to back off for all available actions a request will be made by the management node. The requested action is recorded to a data structure by invoking the **`RecordActionRequest`** (p. 507) method and passing all the available data (current time, the source node that the request is on behalf of, the management node making the request, the drop rate per second, the receive rate per second, the send rate per second, the density, the current operating channel, the current protocol and the requested action). The action is then requested by calling the **`RequestAction`** (p. 507) method.

What remains is the management node to instruct the Network Command Centre to consider the requested action. If the network has had sufficient time to stabilise since a previous request and if the Network Command Centre is not already schedule (by this manager or another manager) the management node will request the actions to be considered by the Network Control Centre immediately. However, if the Network Command Centre has a previous request pending the management node will schedule a request at some point in the future. The time in the future is four seconds after the time to stabilise period has passed. In either case (immediately or at a scheduled future time) the **`NetworkCommandCentre`** (p. 508) method is called to consider the requested action.

Should this be the case the management node associated with source node k can instruct a controlled node to move to a new location in order to re-establish or strengthen the link. The location that the controlled node will move to is based on the current location of the destination, say coordinates $(x^d(t_0), y^d(t_0))$, the current location of the source $(x^k(t_0), y^k(t_0))$, and the location $(x^b(t_0), y^b(t_0))$ of the node that has the most dropped packets for the current source node. The predict position method is used by calling **`PredictPosition`** (p. 469) to calculate the predicted location of the source and destination nodes at time $t = t_1$, $(x^k_{predicted}(t_1), y^k_{predicted}(t_1))$, $(x^d_{predicted}(t_1), y^d_{predicted}(t_1))$, and send the controlled node to the new goal:

$$(x^c_{goal}(t_1), y^c_{goal}(t_1)) = (\frac{x^k_{predicted}(t_1)+x^d_{predicted}(t_1)}{2}, \frac{y^k_{predicted}(t_1)+y^d_{predicted}(t_1)}{2}),$$

i.e. to the mid-point between the two predicted locations. Repeating this procedure at each checking interval provides better and better approximations of the predicted locations. The procedure completes when the connection is re-established.

Once it has been established that the network performance for source m is degrading the gradient for the two intervals is calculated $(\mathrm{t}_{-2}, \mathrm{t}_{-1}), (\mathrm{t}_{-1}, \mathrm{t}_0)$, i.e. $\mathrm{gfr}(k^s, \mathrm{t}_0) < 0$, the network would base its action on the previous gradient $\mathrm{gfr}(k^s, \mathrm{t}_{-1})$. If the previous gradient $\mathrm{gfr}(k^s, \mathrm{t}_{-1}) > 0$, was positive, in this case the manager will not perform the action this time but will retain this information and increase the checking frequency so that the next checking time, i.e. t_1, is halved not to miss this trend in performance at the next checking interval time.

However, if $-1 < gfr_k(k^s, t_{-1}) < 0$, the manager will look through the history of flow rates (searching through $\{t_i\}$, $i < -2, t_i \rightarrow 0$ as i decays) to find a similar state of the network and then will look at the history of the actions to make a decision. So the next stage in decision making procedure will consist of two major steps: looking for the systemState structure (full history of the flow rates) and looking through the archive of performed actions with later reflections on how clever they were.

The management node will look through the actions taken at time t_i, and since this historic time is similar to present network conditions (state of the network) the network will apply an action applied at time t_i, provided it was successful. Each node k's manager then produces either 0 or 1 as a request for switching protocols. So NCC is being passed the string of zeroes and ones: $\{j_k\}, j \in \{0,1\}, k \in \{1, \dots, N^S\}$.

At time t_0 NCC collects requests from all managers for switching the protocol. The set of collected data will look like: {0,1,1,0,1,…,1,1,1,0}. The size of this set is n (the number of source nodes), together with it, NCC gets the predicted values $\{fr(k^s, t_1)\}$, $k^s \in \{1, \dots, N^s\}$.

$FR(t) = fr(0,t) + fr(1,t) + fr(2,t) + fr(3,t) + \dots + fr(N^S, t)$, where $fr(k^s, t)$ is the current flow rate calculated for each node k^s at time t. NCC also calculates the predicted performance based on predicted by managers rates:

$$\widetilde{FR}(t_{k+1}) = fr(0, t_{k+1}) + fr(1, t_{k+1}) + fr(2, t_{k+1}) + \dots + fr(N^S, t_{k+1}).$$

Compare the two values: $\widetilde{FR}(t_{k+1})$ and $FR(t)$ to check if the increase is expected.

Then it calculates $FR(t_{-1})$ and $FR(t_{-2})$ and repeats the procedure of finding the most similar situation in the history (Step 1 and Step 2 above) and makes a decision based on previous actions.

By moving through time points (on the current protocol) towards past (t_k,t_{k-1},t_{k-2}), k<t_{-2}, each time the network evaluates the following error function

$$E^i = (FR(t_i) - FR(t_0))^2 + (FR(t_{i-1}) - FR(t_{-1}))^2 + (FR(t_{i-2}) - FR(t_{-2}))^2$$
$$= \sum_{l=0}^{2} (FR(t_{i-l}) - FR(t_{-l}))^2 \ \rightarrow min$$

The set of values E^i is then examined on the minimal value. The set of times (t_i,t_{i-1},t_{i-2}) that bring minimum to the error function are treated as similar to present state and the network is ready to move to step 2. The NCC will look through the actions taken at time t_i, and since this historic time is similar to present state of the network the network will apply an action applied at time t_i provided it was successful.

8.4 Management Role

8.4.1 Requesting Actions

The **`RequestAction`** (p. 507) method adds the requested action to a data structure called newRequests. The information added is: the time of the request, the management node making the request, the requested action and the predicted receive rate should no action be taken.

The network command will also acknowledge the request and the management nodes memory will be updated to reflect that the request has been acknowledged. This is achieved by setting the requestProgress value of the manageNodeMemory vector to 'A'.

The **`RecordActionRequest`** (p. 507) method commits to memory the current time, the source node being managed at this time, the identifier of the management node, the drop rate per second, the received rate per second, the send rate per second, the current density for the management node, the current operating channel, the current protocol and the requested action.

8.4.2 Recording Action Outcomes

The **`RecordActionOutcome`** (p. 517) method will record the outcome of any actions authorized by the NCC. A relatively simple process is used in establishing if the outcome improved the performance for the source node that the management node requested the action for.

First, the current simulation time is obtained by calling the **`GetTimeNow`** (p. 442) method, this is stored in a local variable called timeNow. A variable called outcome is also created and initialized to 0.

The management node memory is checked to establish which source node the action was requested for by searching the memory for the time of the request (atTime) and the action that was requested (action) for the appropriate management node (mNode).

Once the correct entry has been found, the source node is recorded by storing the node identifier in a variable called sNode. Then for this source node, the number of packets received for a specific duration is obtained by calling the **`GetCurrentReceivedPacketsForSource`** (p. 487) method. The duration is from the time of the request until the current time. The return value from the method is copied in to a local variable called packetsReceived.

A similar action is performed in order to establish how many packets were sent for this source node for the same period by calling the **`GetCurrentSentPacketsForSource`** (p. 488) method, the return value is copied in to a local variable called packetsSent. The same action is performed for the number of packets dropped for this source using the same period by calling the **`GetCurrentDroppedPacketsForSource`** (p. 489) and the return value is copied in to a variable called packetsDropped.

The number of received, sent and dropped packet values are used to calculate the number of packets received per second, sent per second and dropped per second by applying the following formulae:

$$rr(k^s, t_0) = \frac{rp(k^s, t_{-1}, t_0)}{t_0 - t_{-1}}\ , sr(k^s, t_0) = \frac{sp(k^s, t_{-1}, t_0)}{t_0 - t_{-1}}\ , dr(k^s, t_0) = \frac{dp(k^s, t_{-1}, t_0)}{t_0 - t_{-1}}$$

Now that the received, sent and dropped rates per second have been obtained they can be compared. The current rates are compared against the previous rates (the rates recorded at the time of the request). For example, the current received rate per second is compared against the recorded received rate per second when the management node made the request. If the current received packet rate has increased then network performance has increased and one is added to outcome, if the current received packet rate is less than the previous received packet rate one is subtracted from outcome to indicate that network performance has decreased. Should the current receive rate and the previous receive rate be identical the value of outcome is not changed.

Similar actions are performed for dropped packets. One is added to outcome if the number of dropped packets for this period has decreased, one is deducted from outcome if the number of dropped packets has increased. In relation to the send rate, one is added to outcome if the rate of sending packets has increased or deducted if the rate of sending packets has decreased.

This gives values of $-3, -2, -1, 0, 1, 2$ or 3. Zero indicates that there has not been any change in regards to performance for the source node, -1 indicates that performance has decreased slightly, -2 indicates that performance has decreased, and -3 indicates that performance has dropped further. The positive values indicated performance has increased. One indicates a slight increase in performance, two - an increase in performance and 3 indicates that performance has increased further.

The outcome is recorded for this manager node and used when evaluating future recommendations by this manager node.

8.4.3 Recalling Previous Actions

The **`PrintVectorOfMemoryEntries`** (p. 544) method prints to screen all the current memory entries for each and every management node. The information printed is the time of the request, which source node the request was on behalf of, the management node identifier making the request, the drop rate at the time the request was made, the receive rate at the time the request was made, the send rate at the time the request was made, current density at the time the request was made, the routing protocol in use at the time of the request, the request that was made, the progress of the request (i.e. waiting acknowledgement, acknowledge, permitted or denied), the back off time for making the same request and a performance metric indicating if performance has been checked since the action and if it has the normalized value indicating the performance based on the action. This method is used for information to ensure the management nodes memory is working correctly.

The **`GetAction`** method is the method where the management node considers its memory of previous actions and the consequences of those actions in order to make a decision that will best serve the network in regards to performance by using the current network state.

A vector called managementNodeMemory has been implemented to store the memory of past events. The management node loops through its memory looking at past events. It is assumed that historic events may be forgotten so a past Count is included, this allows the control of how far back in time the manager can recall memories of previous requests or actions. The management nodes begins to recall memories starting with the most recent and working backwards to the lesser recent memories.

When the management node puts a request to the Network Command Centre to change the operation of the network the Network Control Centre will consider the request and allow or deny the request. In either case the Network Command Centre will send a backoff time to the management node to prevent the management node making the same request until the backoff time period has elapsed.

The management node is able to request three different actions. There could arise a situation where the management node has unspent backoff times for each of the three requests, therefore to prevent the management node making any further requests a strikeCount has been included in the procedure. If the strike count becomes equal to 3 the management node is not allowed to make any further requests until the minimum backoff time has been spent (elapsed).

The management node, as it recalls previous actions from memory is only interested in the actions that resulted in a positive outcome (i.e. performance of the network increased); therefore, the manager will only consider events that have resulted in a positive outcome. If the management node recalls a previous memory that did result in a positive outcome the management node will recall the drop, receive and send rates for this period, in addition to the density, the operating channel and the routing protocol that was in use. Using the information obtained the management node will attempt to calculate the how similar the conditions were then from what they are now, this is achieved by performing the following calculation and give a distance value. The value is stored in a variable called distance.

For each memory line, that corresponds to the moment $t = t_m$, dropped, received and sent errors are calculated by comparing to the current (at $t = t_0$) rates:

$$DR(k^s, t_m) = (dr(k^s, t_0) - mdr(k^s, t_m))^2 * w_d$$

$$RR(k^s, t_m) = (rr(k^s, t_0) - mrr(k^s, t_m))^2 * w_r$$

$$SR(k^s, t_m) = (sr(k^s, t_0) - msr(k^s, t_m))^2 * w_s$$

Here above $w_d = 0.1, w_r = 0.5, w_s = 0.4$ are the weights used to stress the importance of receiving rates. Also, $mdr(k^s, t_m), mrr(k^s, t_m)$, $msr(k^s, t_m)$ are correspondingly the dropped, received and sent rates taken from the memory of the manager node. Hence, the distance is calculated from current state of rates to the one that happened at $t = t_m$

$$distance_m^k = DR(k^s, t_m) + RR(k^s, t_m) + SR(k^s, t_m),$$

and find the state at the minimal distance from $t = t_0$.

The next check is to establish if the distance from this memory is less that the already established closest distance. The closest distance is the action that has the most similarities in network state. For the closest distance the action that was performed previously is obtained, this action is recorded so that the same action can be requested.

There may be a situation in which the current memory being recalled does not have backoff time attached and shows that the performance of this action hindered network performance. If this is the case a counter for that action is updated, for example if the action was to change channel one is added to the performance value of CCount. This is done so that the least hinderer's action can be determined.

Once the memories of past actions have been recalled there will be one of four situations.

1. the management node has three strikes
2. an action that resulted in a positive outcome
3. three counters with values derived from past experiences (negative outcome)
4. could not determine if positive or negative outcome

If the management node has three strikes that action sent back to the method that invoked this function is the minimum back off time.

Otherwise, if at this stage an action has not been determined, i.e. in the case of not being able to establish a positive outcome from memory (i.e. if the management node is new to the role) the manager is allowed to go with their intuition (gut feeling). However, this is only the case if their intuition does not conflict with the manager being told to back off for this action. For example, consider a manager making a request for the use of a controlled node, the Network Control Centre authorizes this requests and asks the manager to back off from making the same request again until a certain time has elapsed (normally until the node has reached a given location). The decision based on intuition is changed if there is evidence that the gut feeling has some history of hindering network performance rather than improving. If this is the case the action is set to that of a better informed decision using the performance counters.

If the managers gut feeling was to make a request that has a back off time attached or if it cannot reach a decision based on performance counters then an action is chosen at random.

When choosing a random action steps are taken to ensure that that random action is not one that has a back off time attached.

8.5 Network Command Centre

8.5.1. Main functionality

The `NetworkCommandCentre` (p. 508) method is called by a management node(s) to consider their requests – the request could be authorized or denied. It could transpire that the network is in a state of fluctuation, if this is the case the Network Command Centre (NCC) will be scheduled to be called at some point in the future. When the Network Command Centre is called the NCCschedule variable will be set to false to show that the Network Command Centre is not currently scheduled and therefore can be scheduled by a management node.

Each time the Network Command Centre is called, a time to stabilise variable is set, which is 2 seconds by default.

The NCC is to consider the current requests of the manager nodes. There may be one or many requests depending on the number of manager nodes and the time elapsed since the NCC was last called. The manager nodes should have the ability to make requests even if the NCC is busy dealing with current requests. Therefore, the current requests are copied into a data structure called newRequests and the requests data structure is cleared (emptied). This allows the manager to deal with the current requests but also allows managers to make requests in the normal way. Any requests made whilst the NCC is busy will be dealt with the next time that the NCC is called.

The NCC will only authorize one request each time the NCC is called upon, this is to prevent manager nodes cancelling out each other’s requests. For example, consider manager node 2 makes a request to switch to another routing protocol and manager 4 requests the same action. If the NCC authorized both requests there may be a situation of switching twice.

Therefore the NCC keeps counters: M (control of mobile node), C (change frequency channel) and P (switch protocol). The NCC will attempt to authorize the action with the most requests, however, it could transpire that the most requested action is not suitable, in this case the NCC will then consider the next most popular request.

If the most popular request was to move a controlled node the NCC has to consider carefully which manager should be authorized to instruct the controlled node, this is because the number of controlled nodes in finite. In the simulations that had full cognition switched on the number of controlled nodes is typically set to 2, however, this is at the discretion of the application of the network and can be increased or reduced depending on circumstance.

There should not be a situation where the NCC allows the use of a controlled node shortly after already authorizing a previous manager the use of the same controlled node. Therefore a backoff time is used that allows the previous action(s) of the controlled nodes to be completed. In order to get the minimum backoff time (a time of zero indicates that a controlled node is not currently busy performing a previous request) the **`TimeAllowedToMoveControlledNode`** (p. 473) method is called.

Should there be a situation where all controlled nodes are busy performing a previous request the NCC will set the M counter to 0 so that other actions may be considered. In addition to this it will deny all requests from any manager that has requested the use of a controlled node and send them a backoff time so that the manager cannot make the same request again until at least one controllable node becomes available. This is achieved by the NCC calling the **`DenyAllRequestForControlledNode`** (p. 516) method.

Providing that there is at least one controllable node available the NCC should carefully consider which manager node(s) to allow the use of this resource. It was stated earlier that the NCC will only authorize one request per NCC call and that there may be several requests to consider. In the case of controllable nodes there could be a situation where there are multiple manager nodes requesting the use of a controllable node, if the controllable nodes are available the NCC will assign the available controlled nodes. Consider for example a case where there are three manager nodes and each has requested the use of a controllable node and that there are only two controllable nodes. The NCC will make an effort to accommodate the request for two of the manager nodes and deny the request for the third. A first come first serve scheduling algorithm was considered to accommodate this, however, the NCC should be more interested in improving overall network performance. Therefore a scheduling algorithm was implemented which uses current receiving rates and predicted receiving rates to determine the most appropriate manager to release a controlled node to. This is explained in more detail when the **`GetManagerToMoveNodeFor`** (p. 516) is discussed.

Once it has been established which manager node to move the controllable node for a prediction is performed to determine how long the manager should make use of the controllable node. This is so that the resource can be allocated again later. Or as the case may be if the NCC should deny this resource at some point in the future. This value is also used by the NCC in its final consideration to allow or deny the request of the resource.

The predicted time that the controllable node will be in use is obtained by calling the **`MoveControlledNode`** (p. 522) method and passing the appropriate manager identifier. There may be a situation where a manager node is looking after multiple source nodes, in this situation the manager will select the most appropriate source node and this is explained when the **`MoveControlledNode`** (p. 522) method is discussed.

The NCC makes one final check to see if the use of the controllable node is suitable for this situation. It does this by checking how long the controllable node will be used for, if the value is negative (which is the indication that the source node will already be in range of the destination node before the controllable node reaches its goal destination) the NCC will deny this request. It does this by calling the **`DenyARequestForControlledNode`** (p. 516) and passes the appropriate manager identifier and a back off time so that the same manager will not make the same request for this given situation.

If the NCC is satisfied that the use of the controllable node will be of some use for the manager node that requested this action, the NCC will authorize the request by calling the **`GrantRequestForControlledNode`** (p. 519) and pass the appropriate manager identifier and a backoff time so that this manager cannot make another request to move a controlled node until the current request has been completed.

If the most popular request was to change channel, of if the second most popular request was to change channel and the most popular request was to move a controllable node but the NCC denied all requests for that action the NCC will switch channel. In order to switch channel the NCC first checks which channel the data communications network is operating on by calling the **`GetChannel`** (p. 534) method and passing an arbitrary node number (all nodes transmit on the same channel). Once the current operating channel has been established, the NCC will instruct all nodes, including the controllable nodes to switch to the alternative channel by calling the **`SetChannel`** (p. 534) method and the **`SetChannelControlledNodes`** (p. 535) method. Once the action has been completed the

NCC gets a backoff time for this action by calling the **GetPeriodToStablise** (p. 517) method. The NCC will also send information to the management nodes that requested the change channel action with the back off time by calling the **GrantRequestForChangeChannel** (p. 521) method. The final action if change channel occurs is to print the channel each node is operating on to screen. This does not alter the state of the network and is for information purposes that is used to ensure that channels have changed correctly for each and every node, this is achieved by calling the **PrintAllChannels** (p. 544) method.

If the most popular request was to switch routing protocol, of if the second most popular request was to switch routing protocol and the most popular request was to move a controllable node but the NCC denied all requests for that action the NCC will switch routing protocol. First, a back off time for this action is obtained by calling the **GetPeriodToStablise** (p. 517) method. Next the NCC will inform all the management nodes that requested to switch routing protocol that this action has been authorized, it does this by invoking the **GrantRequestForSwitchProtocol** (p. 517) method. The NCC then instructs all nodes to switch to an alternative routing protocol by calling the **SwitchProtocol** (p. 520) method.

The **GetManagerToMoveNodeFor** (p. 516) method will establish which manager is most in need of the use of a controllable node.

Next the method will check each manager node that has made a request to move a controllable node. Network Command Centre checks the predicted receive rate for each manager to determine which manager is most in need of a controllable node. Should the node that is being checked have a predicted receive rate that is less than the minimum receive rate the minimum receive rate is updated to reflect the predicted receive rate for that node. Also updated is the mNode variable so that the manager node can be kept track of. Once all the manager nodes that have requested the use of a controllable node have been checked a management node identifier is returned to the invoking function.

In the unlikely event there is the same number of requests for all the actions a message is generated informing of a three way tie.

8.5.2 Stabilization

The **`GetPeriodToStablise (p. 517)`** returns a value that indicates how long the Network Command Centre should wait before considering new action based on the current action just performed.

8.5.3 Denied Actions

The **`DenyAllRequestForControlledNode`** (p. 516) method will update all manager nodes that have requested the use of a controlled node and instruct that they back off from asking for the same request again until the back off time has elapsed. Each manager's nodes memory will be updated to reflect that the request was denied and when it is appropriate to make the same request again.

The **`DenyARequestForControlledNode`** (p. 516) method will update a specific manager node that has requested the use of a controlled node and instruct that they back off from asking for the same request again until the back off time has elapsed. This manager nodes memory will be updated to reflect that the request was denied and when it is appropriate to make the same request again.

8.5.4 Finding suitable controlled node

The **`GetDestinationNodeForSourceNode`** (p. 525) method is used to establish which of the destination nodes that this source node is receiving the least number of packets. First a minimum received number of packets is set to 9999999 and stored in a variable called minPR.

Next the current simulation time t_0 is obtained and stored in a variable called timeNow by calling the **`GetTimeNow`** (p. 442) method. This is used later when defining a time interval.

Not every source → destination pair needs to be checked, only the pairs that are currently transmitting, therefore only the source → destination pairs that are currently transmitting or that are scheduled to be transmitting at this time are checked.

For each of the destinations that are paired with the source node, the number of received packets for each destination for a given period is obtained. In this case the period is the last five seconds, this is achieved by calling the **`GetCurrentReceivedPacketsForDestinationFromSource`** (p. 486) method and passing the source node identifier k^s, the destination node identifier k^d, the from time and the until time. The from time is five seconds before timeNow and the until time is timeNow. The value returned from this method reflects how many packets the destination has received from the source node for the last five seconds, this value is copied to the numPR variable.

Should the value received be less than the minPR value the minPR value is updated to the value in numPR, the destination node identifier is also recorded.

Once each destination paired with this source node for all the active (or should be active) transmission paths have been checked, a source node identifier is returned. The source node identifier is the source node that has received the fewest packets within the given time period.

The `GetSourceNodeForControlledNode (p. 525)` method is used to establish which of the source nodes that this manager is managing is performing the worst. First a minimum receiving rate is set to 9999999 and stored in a variable called minRR. Then for each source node the receive rate is checked to determine if it is less than the value in minRR, if the value is less minRR is updated to reflect the receive rate for that source. Also recorded is the source node identifier. When each source node that this manager is responsible for has been checked the source node identifier with the poorest receiving rate is returned.

The `GetClosestControlledNode (p. 526)` method gets the closest available controlled node between the source node and the destination node. The method begins by declaring a variable called chosenCNode and assigning the value -1 and minDist and assigning the value 999999999.

For each controllable node, a check is performed to determine if the node is available (not currently moving) by calling the `IsControlledNodeMoving` (p. 463) method. If the node is available then the current controlled node position is obtained by calling the `GetControlledNodePosition` (p. 527). The destination node position and the source node position are also obtained by calling the `GetNodePosition` (p. 464) for each node and passing the appropriate node identifier.

Now that the source node position (x_s, y_s) and destination node position (x_d, y_d) have been obtained a calculation is performed in order to get the midpoint (x_m, y_m) between the source node and the destination node.

$$(x_m, y_m) = (\frac{x_s + x_d}{2}, \frac{y_s + y_d}{2})$$

Next, the distance from the controlled node to the midpoint derived by the formula above is obtained by calling the `GetDistance` (p. 465) method and passing the two position vectors. If the distance is less than the value in minDist then minDist is updated to reflect the new minimum distance, also updated is chosenCNode to the node identifier.

Once all controllable nodes have been checked the node identifier that signifies the closest node to the midpoint between the source node and the destination node is returned. If there are no controllable nodes available the value -1 is returned.

The `GetControlNodeMidPointXY` method will predict if the controlled node is able to reach the midpoint between a source node and a destination node within a given time constraint. If the controlled node can reach the midpoint within the time constraint the new coordinates are returned that reflect the position of the midpoint for the time that the midpoint was reached.

First, the current simulation time is obtained by calling the `GetTimeNow` (p. 442) method. A futureTime variable is declared and initialized to the current simulation time. A constraint time (the time by which the controlled node must have reached the midpoint of the source and the destination node) is created by declaring variable called journeyTime and setting this to the remaining simulation time.

While the time at which the midpoint will be reached has not been established, the predicted position of the source node is calculated by calling the `PredictPostion` (p. 469) method. The same method is called to get the predicted position of the destination node.

The time of which it would take the controllable node to get to the midpoint derived from the above formula is obtained by calling the `GetTimeOfJourneyControlledNode` (p. 464) method and passing the controlled nodes current location (the position of the controllable node is passed into a Vector called cNode), the midpoint position and the controlled node speed. Time is stored in a variable called journeyTime.

If the journeyTime plus the current time is less than the future time then the controlled node can get to the midpoint, in this case the controlled nodes position is updated to reflect the position of where it would be when it reached the midpoint.

If the journeyTime plus the current time is greater than the future time, the future time is incremented by one and repeats the process. The process is repeated until the midpoint can be reached, or until it has been established that the controlled node cannot reach the midpoint within the given time constraint.

The controlled nodes vector is returned to the invoking function. If the controlled nodes coordinates have been updated it means the controllable node will reach the midpoint. Otherwise the controlled nodes coordinates will not be updated.

8.5.5 Granted Permissions

The **`GrantRequestForSwitchProtocol (p. 517)`** method grants the request for each manager node that requested to switch routing protocol. In addition to granting the request a back off time is sent so that the manager cannot make the same request again until the back off period has elapsed. Each manager that was granted this request will call the **`RecordActionOutcome`** (p. 517) in order to record if performance has increased or decreased for the node that they are managing at the time specified in the back off time. The **`PrintVectorOfMemoryEntries`** (p. 544) method is called once all the requests have been granted for this period. This is for informational purposes only and does not alter the state of the network.

The **`GrantRequestForControlledNode (p. 519)`** method will grant authorization for a manager node the ability to move a controlled node. The method begins by getting the current simulation time by calling the **`GetTimeNow`** (p. 442) method. Next the appropriate manager will update its memory to reflect that authorization has been given that shows it has been permitted use of a controlled node. The manager also updates its back off time so that it cannot make the same request again until this request has completed. In addition to this the manager will schedule and event based on the back off time in order to record the outcome of this action by calling the **`RecordActionOutcome`** (p. 517) method. The ability to schedule when to check the outcome of the action allows the management node accurate times to check if performance has increased or decrease for the given action. This method also calls the **`PrintVectorOfMemoryEntries`** (p. 544) to output the memory management process to screen in order to ensure that the memory entries are being accessed and updated correctly.

The **`GrantRequestForChangeChannel`** (p. 521) method grants the request for each manager node that requested change channel. In addition to granting the request a back off time is sent so that the manager cannot make the same request again until the back off period has elapsed. Each manager that was granted this request will call the **`RecordActionOutcome`** (p. 517) in order to record if performance has increased or decreased for the node that they are managing at the time specified in backOffTime. Also called is the **`PrintVectorOfMemoryEntries`** (p. 544) once all the requests have been granted for this period. This is for informational purposes only and does not alter the state of the network.

8.6 Memory

The data structures that are used for storing information consist of many variables, however, variables are limited in what they can store - normally each variable will only hold one piece of information. Initially, arrays were used to group items of related data. An array is a more complex data structure that allows a fixed number of the same data type to be stored and accessed using an index. The limitation of arrays became obvious quickly, the data that is being stored is of an undetermined size, the fixed size data structure either meant the array was too small to store all the information that was required or too large and consisted of many gaps and was a waste of the computers finite resources. In addition to this the limitation the data had to be the same data type (i.e. all integers) further restricted how related data could be grouped; however, this was overcome by introducing structs. A struct is a complex data type declaration that allows the grouping of variables of different types under one name. Array of structs were then created. To overcome the limitation of the fixed size the array was dynamically created. A new array of a given size was created should the original array become full, then transfer the contents of the original array to the newly created array and destroy the original array. This method is slow and was repeated often.

Therefore, the memory used by the management node consisted of a more complex data structure known as a vector in the C++ programming language. A vector can be thought of as a combination of an array of structures that is able to grow and shrink in size. Vectors are sequence containers and allow for contiguous storage locations and are accessed using memory pointers, the vectors automatically resize without the need to create new vectors. Thus, new vectors did not have to be created, transfer of the contents from the original vector did not have to occur and nor did original vector have to be destroyed, like in the initially case when arrays were used.

The original installation of NS3 uses the standard C++ compiler. In order to take advantage of the vector data structure changes had to be made to the Waf (a tool used to assist in compilation of computer programs) configuration so that it would use the C11++ compiler rather than the original compiler.

8.7 Attention

NS3 has the capability of allowing the storing or obtention of information regarding almost all aspects of the simulation model thus generating vast amounts of data. The NS3 logging feature allows for the turning on or turning off certain logging of components. However, although this feature was very useful it did not give the granularity of control that was required by the simulations. Thus, a logging system was created that allowed attentiveness to the current situation that was being tested. This logging system allowed the logging (or not as the case may be) when a function was called, when a function completed, dropped packet information, packet sent information, packet received information, packet route information, the distance between two given nodes, device information, address information, traffic flow information, election process information, destination information, position information, remaining energy information, distances for all nodes for the source node(s), neighbour distances from the source(s) node, routing information and specific node information. As part of the logging a debug step count was also implemented. The debug step count allowed the presentation of the log information in stages and prompted the user to continue after each stage. This was particularly useful when things did not work as intended and it was needed to ascertain what part of the simulation model was incorrectly configured.

As well as logging, also accounted for was attention by only storing information that was required. Other recorded information relates to the number of nodes in the simulation, the current protocol being used, the transmission power, the name of the prefix for the log files, total number of bytes sent and received, total packets received, the port number, the size of the world, the default node speed, the transmission radius, the energy detection threshold, the transmission gain, the receiver gain, the transmission power level, the receiver noise figure being used, the initial energy level of the energy sources, the time for the network to stabilise before transmitting data, the interval at which network state would be checked, the total number of AODV packets dropped, the total number of AODV packets received, the total number of AODV packets sent (and for DSDV), total number of packets received as a result of switching routing protocols and others. These values are not part of the cognition but define the parameters of the network simulation model.

Cognitive memory has been mentioned in the previous section. The more complex data structures consisted of information that related to: neighbours from a source node, all the nodes in the world, all the nodes in the vicinity of a given node, nodes that would be in range of the sources node for an estimated time, the candidate suitable for an election process, flow information, current positions, manager node information, source and destination IP addresses for packets, sent, received and dropped, the controllable node information, when a node was moving, the current system state, traffic information and scheduled traffic flow information. The only information stored was that used during the simulation runs.

8.8 Chapter Summary

This chapter presented the implementation of the cognitive application. The chapter began by discussing the admissible cognitive actions (change routing protocol, change frequency and instruct a controllable to not a new destination). An election process was discussed that detailed how a neighbour of a source node could become a candidate to become elected a manager node based on the predicted time in range of the source, the predicted time that the candidate would have a suitable energy level and the density of the candidate. In addition to this it was discussed that if a candidate node was also transmitting it could still be elected manager node but it was less desirable. This section also discussed the relationship state of the elected manager and source node, and how that dependent on the relationship state a new election may be performed (i.e. if the manager node is predicted to be out of range of the source node within a given time period).

Section 8.3 State of the Network, discussed how received rates, dropped rates and sent rates are used by the manager node in order to predict or detect if performance for the source node that is being managed has dropped. Part of this process used least squares approximation and it was explained that 6 points were used to predict future network states.

Section 8.4 Management Role detailed how the manager node could request a change in the network by requesting the use of additional resources, changing channel, or changing frequency. Also detailed was the procedure undertaken in order to make this request and the actions required if this request was denied or granted.

In order to request a change in network operation the manager node undertakes some cognitive process as explained in section 8.4.3 Recalling Previous Actions.

Section 8.5 Network Command Centre introduced an overarching architecture that allows for multiple managers making multiple requests. It was explained that the NCC authorises or declines each of the managers' requests after considering the predicted states of each of the managers.

This meets Objectives 2, 3, and 4:

Objective 2. To design a suitable testbed in order to test the implemented elements described in Objective 1, whilst ensuring that the solution is rigorous, transparent, and replicable for the testing of the scientific theories. It will be assumed that this objective is achieved when the MANET testbed has been configured using current wireless communication standards found in MANET technology.

Objective 3. To enhance the design of the MANET to incorporate cognitive attributes to the designed MANET in Objective 2. Discussion of the essential necessary features that allow one to call a network cognitive is presented. Once these features are listed with brief explanations, each attribute is discussed separately. It is assumed that this objective is achieved when all the cognitive attributes are ready to be implemented into the testbed.

Objective 4. To apply cognitive attributes to a MANET as discussed in Objective 3 into the testbed implemented in Objective 3 by enhancing the testbed and by making the MANET cognitive. It is assumed that this objective is achieved when simulations of a MANET with cognitive attributes applied are ready to be run and when the configuration settings used in the code run correctly.

The design, and implementation of a cognitive MANET testbed. All the proposed functionality is now implemented and the testbed was tested for being correctly reflecting actual MANET. Next the model simulations are discussed and the performance compared under various settings.

Chapter 9 Results

This chapter presents the results from a total of 52 simulations that were ran. The chapter begins by discussing the settings applied to control the configuration of the model (Section 9.1). The results from the 7 models that were ran are presented and the results are discussed next. Each model had to be run several times so that the results could be compared. Section 9.2 discusses the simplest model which consisted of 10 nodes, 1 traffic flow and 1 destination. Section 9.3 builds on Section 9.2 and presents a model of 10 nodes, 4 traffic flows with 4 destinations. In Section 9.4 a model with one traffic flow to one destination is created, but, the number of nodes is altered and simulations are run for 20, 40 and 80 nodes. In section 9.6 a model was run with 40 nodes and one traffic flow, however, the simulation time was increased significantly from the default setting of 600 seconds to 1200 seconds. The increase in simulation time was to determine if the network makes better decisions after 600 seconds as a result of experience.

All the models mentioned so far use the bespoke mobility models. The Random Waypoint mobility model is also used and section 9.7 presents the results with 20 nodes and 40 nodes with one traffic flow using the random waypoint model. This is extended, and in Section 9.8 results are presented with 20 nodes and 40 nodes with 4 traffic flows.

All of the models require several simulation runs (i.e. for each model: AODV, DSDV, with partial cognition, with full cognition) and each of the four runs are compared for that model. Table 5 - Models' Basic Settings summarises each of the models.

Model	Number of Nodes	Mobility	Time	Number of Traffic Flows
Model 1	10	Lakes	600s	1
Model 2	10	Lakes	600s	4
Model 3	20	Lakes	600s	1
	40	Lakes	600s	1
	80	Lakes	600s	1
Model 4	20	Lakes	600s	4
	40	Lakes	600s	4
	80	Lakes	600s	4
Model 5	40	Lakes	1200s	1
Model 6	20	RWP	600s	1
	40	RWP	600s	1
Model 7	20	RWP	600s	4
	40	RWP	600s	4

Table 4 - Models' Basic Settings

9.1 Simulations Settings

In order to compare and contrast data many variations of each network model have been run. Mainly, results are compared and contrasted across simulated networks with:

1. no cognition using the AODV routing protocol.
2. no cognition using the DSDV routing protocol.
3. partial Cognitive attributes that entails an election of a manager node(s) and cognition applied to the manager node(s) to allows the manager(s) the ability to request a change in routing protocol and/or frequency channel assignment.
4. full Cognitive attributes applied; this allows management nodes the ability to request the use of a control of node in addition to partial cognitive attributes.

The simulation models begin by modelling relatively simple scenarios but each subsequent model includes more complexity, for example, the first scenario models an environment with a small number of nodes with communication via a single source and destination, later models consist of multiple sources with each source communicating with multiple destinations.

In order to test the theories that cognition applied to a MANET will enhance the overall performance of the network the same models with the cognition switched off were run, thus there is no election process, no management nodes and no memory of past events, but the parameters for each network run, i.e. number of nodes, placement strategy, mobility model and traffic flows are consistently applied for each model.

The results presented give an overview of how the network has performed. In particular, plans that were proposed are reported, if the plan that was requested was initiated, and if this had any impact on the network. Overall statistics for each model are also reported including the number of packets sent, received and dropped. Also reported is the rate of flow per second at times of significance.

9.1.1Topology of the "Lakes Scenario"

In the first simulation runs, a realistic scenario model is created which is based on a search and rescue operation. The scenario used consists of soldiers engaging in a search operation and navigating a geographical area consisting of lakes. Similar scenarios could be search and rescue operations carried out by the Lake District Search and Mountain Rescue Association which reported 474 incidents in 2014 (Warren, 2015. pp. 3).

This can be seen as a test case model, using this model allowed certain checks to be conducted to determine if the simulation design accurately reflected a physical implementation of a network with a similar configuration, or that the events in the network where not unexpected.

For the benefit of the reader the time has been taken to explain all aspects of the design here, later model problems are based on this design thus, later models are not explained in the detail that is included here, but focus on the results that the simulations generated. In this model the time is also taken to explain how data is generated from the model, the data that is used, the data that is not used and the justification for the chosen data.

This is the simplest model that was ran in terms of complexity. The simulation runs consists of 10, 20, 40 and 80 mobile nodes which can be thought of as the soldiers.

The first scenario has been modelled in order to test and adjust the ideas used in the thesis. In order to keep track of the changes, the first model begins with only 10 nodes and a specially created model mobility scheme. The created scheme allowed the control of changes in network topology. Hence, for example, the mobility model used managed to prolongate intervals when AODV performed better than DSDV and vice versa. The continual testing and simulation runs determined how long a simulation would need to run in order to begin to show signs that cognitive attributes were having a positive impact on the performance of the networks. If the simulation time was too short the performance would likely be worse because enough experience would not have been gained and mistakes may have occurred when making decisions. On the other hand, if the simulation time was too great it impacted on the resources that were used to run the simulations, for example, a simulation run could potentially take days with a simulation time of 1 hour.

This allowed a simulation time that was not too short or too long, but still allowed sufficient time for the network to demonstrate cognitive attributes, monitor network state, monitor resources and make decisions accordingly.

The simulation time that was agreed on is 600 seconds (10 minutes), although the actual time that the simulations run is much greater because of the amount of data being output for analysis, for example, one of the more complex simulations took over 6 hours to complete and generated data files in excess of 30GB. Model 5 is the only scenario run for the extended time, which is 1200 seconds (20 minutes) in this case.

The mobility scheme also created link breakages, creating a situation where there is no viable route from the source node to the destination node. This allowed the testing of the responsiveness of the manager node(s). The manager node should request the use of one or more of the controlled nodes in an attempt to re-establish the communications link for the source node.

The model also introduces periods where density is higher, thus congestion potentially being greater, which also allows the network the opportunity to choose the most suitable routing protocol or switch transmission to an alternative frequency.

The "Lake Scenario" is tested in Models 1, 2, 3, 4 and 5. Models 6 and 7 are based on Random Way Point mobility scheme, thus are tested on the ground with no obstacles.

9.1.2 Placement Strategy

The nodes are placed on the grid using the **`PlaceNodesTwoGroups`** (p. 451) method and are split into two groups. This is depicted in Figure 32 - Placement Strategy for Nodes, which shows a screen capture from Network Animator, the figure also shows that the first group consists of roughly one third of the nodes, the second group consists of the remaining nodes, roughly two thirds. The first group are called Eagles (green nodes) and the second group are called Sharks (black nodes).

Initial positions are randomly chosen, the Eagles are placed 20 meters from (0, 0) and the Shark's positions are within 20 meters from (130, 0)). Moreover, the consequent motion of all the soldiers is also affected by some slight randomness and is discussed next.

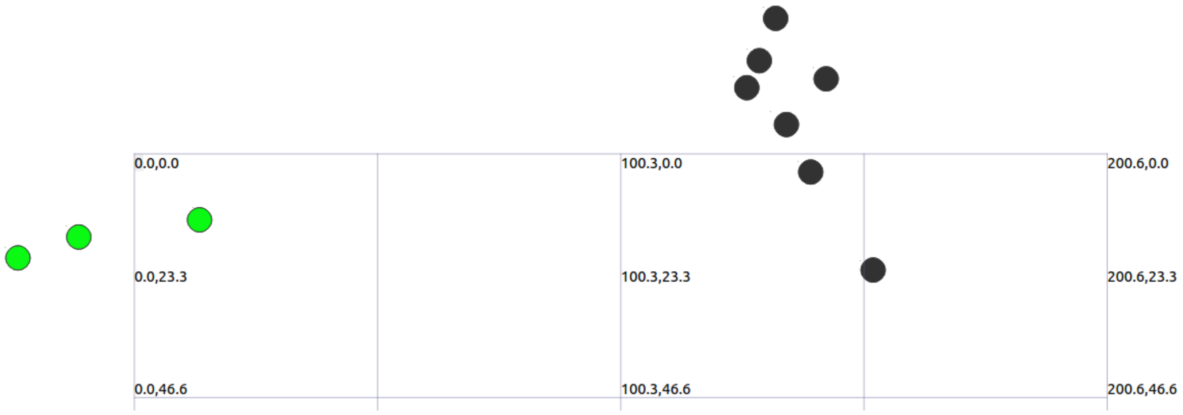


Figure 32 - Placement Strategy for Nodes at 0 seconds

The number of nodes in the network differs for each model. Models 1 and 2 include 10 nodes and compare the performance of the network with different number of traffic flows. Models 3, 4 and 7 are each run with 20, 40, and 80 nodes, so each model includes three scenarios. Model 5 is based on 40 nodes with one traffic flow with extended time. Model 6 consists of two simulation runs for 20 and 40 nodes consequently.

9.1.3 Mobility Models

The first mobility model is implemented by calling the `LakeScenario` (p. 461) method. This method consists of nodes navigating around lakes and is depicted in Figure 34 - Lake Mobility Scenario. Both groups start motion roughly on the same line of trajectory (that corresponds to y=0) at time t=0 and move forward (y-coordinate is increasing as time goes) with x-coordinate increasing for the Eagles and decreasing for the Sharks.

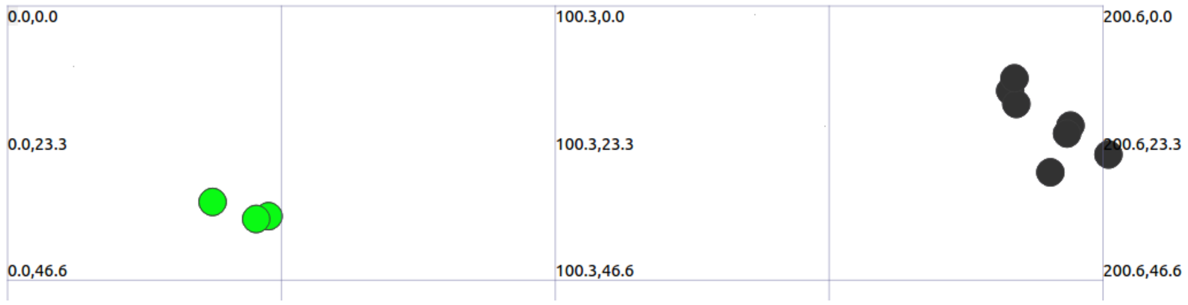


Figure 33 - Placement Strategy for Nodes at 36 seconds

As time passes they begin to move from their initial position to the position of where their search will begin. The search is not started immediately to give the network chance to stabilise. This in effect moves the groups closer to each other which are depicted in Figure 33 - Placement Strategy for Nodes at 36 seconds.

As the nodes continue to move closer the density of the network increases significantly. Once the two groups have met for the first time the search operation begins, this meeting position is depicted on Figure 34 - Lake Mobility Scenario by a red circle directly South of Lake 1 (approximate coordinates are x ≈100, y ≈60).

Structuring the nodes in this way allowed the moving of the groups toward each other which increases density and also potentially creates multiple data communication paths from the source node(s) to the destination node(s).

Moving the groups away from each other has the opposite effect and reduces density and creates link breakages, this occurs as each group checks opposite boundaries of each lake, one group heads West as the other group heads East. One can think of this as two groups of soldiers passing by the lakes using different shores to check the surrounding area.

Each group has a goal position to be reached, but each soldier in the group moves at their own speed and at some small randomly chosen distance from the others. So, both groups move simultaneously along the lakes taking different paths coming closer and moving apart from time to time (see Figure 34 - Lake Mobility Scenario, Eagles take the yellow path, Sharks take the red path, stopping momentarily at each intersection). To emphasize again, this type of motion allowed the testing of the new algorithms on the dynamically changing topology before being applied to the well-known and widely used ones, such as the Random Waypoint mobility model. Thus, when the ideas are tested later for the second mobility model, Random Way Point, it is assumed that there are no Lakes, or any other obstacles in the Network World. All the nodes are positioned randomly and would move randomly.

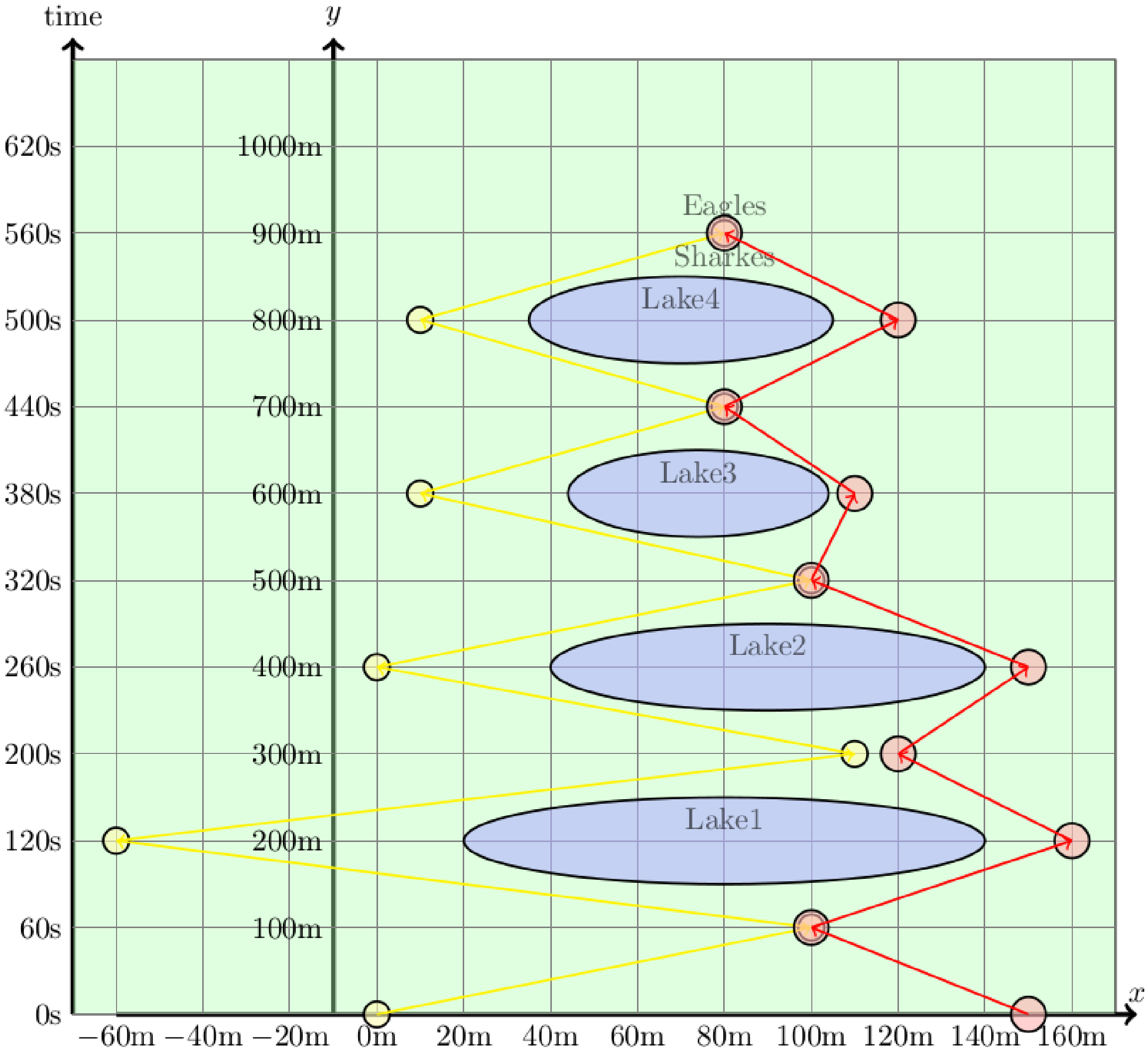


Figure 34 - Lake Mobility Scenario

9.1.4 Traffic Flows

In the first scenario one source node is communicating with one destination node, namely, node 0 is the source node and node 9 is the destination (when there are 10 nodes in total). The traffic flow is scheduled by calling the **`CreateTrafficFlows`** (p. 474) method. The traffic flows are duplicated for both the AODV and the DSDV protocol and are both set to transmit at 0.01 seconds until 10 seconds before the end of the simulation (590 seconds). The traffic flows are created by calling the appropriate methods **`CreateAodvTrafficFlow`**

(p. 474) and `CreateDsdvTrafficFlow` (p. 475). The antennas are set such that the transmission radius is 100 meters which is in line with the 802.11b Wi-Fi protocol transmission properties. The data rate is 11Mbps. It is assumed that the transmission antennas are omnidirectional, so each source node has a transmission area shaped as a circle. The source node is at the centre of the circle and transmits in all directions; the signal propagates up to 100 meters, after 100m the signal is too weak to be received correctly which results in corrupt data and dropped packets.

It should be noted that introducing only one traffic flow initially in the scenario simplifies significantly the job for the network in making decisions. Since there is one source only, the aim of the management node coincides with the aims of the network command centre. Thus there is no antagonism involved. Both management node and network solve the same problem and receive the same benefits if successful. The situation will change drastically once there are at least two source nodes and their aims might contradict each other. This will be shown later in this chapter.

However, the procedure used does not give any benefits for the single source node in order to preserve the purity of the test. An adapted procedure for the case of one management node making decisions could have been implemented into this model. In this monopolistic case the performance of the network is significantly increased because there is only one manager, therefore, no contending for use of resources.

The aim is to implement a model without the need to apply further adaptations for particular cases, i.e. the monopolistic case.

9.1.5 Controlled Nodes

In the first simulation one node was introduced as a controlled node. A controlled node is not designed to be a source node, destination node or manager but is intended to participate in traffic flow as required. This node is never considered as a candidate to become a manager node because it is used to strength links, therefore, will be likely to move from its current path or vicinity to a new vicinity. This may cause the controlled node to move out of range of the source node much sooner than that of a normal neighbouring node.

Initially the controlled nodes act autonomously and move under their own volition; this is achieved by invoking the AutonomousControlledNode (p. 521) method. Should a manager node suspect a link breakage or should a link breakage occur the manager node will request the use of one or more of the controlled nodes.

As detailed in chapter 5, a request is made to the Network Command Centre by a management node. The Network Command Centre will undertake some initial checks before requesting or denying the request. Firstly the Network Command Centre will check to see if a controllable node is available, i.e. not performing an action for another node. If there is a controllable node available the Network Control Centre will inform the management node that a node is available.

The emphasis is then passed back to the manager to undergo some checks of its own. The manager firstly selects the closest available node, it also checks which of the source nodes that this manager is managing requires the controllable node most (the source node with the lowest rate of received packets at its destination(s)). The manager passes this information to the Network Control Centre in order for some more checks to be made.

The Network Control Centre communicates with the controllable node and asks for it to predict when it will be at the midpoint of the source and destination that has been selected by the manager node. This information is returned to the Network Command Centre. The Network Command Centre takes the time passed back from the controllable node and predicts the positions of both the source node and destination node. Using the positions the management node is able to check if the source node will be in range of the destination node at this point in the future. If the source node will not be in range of the destination node at this time the Network Command Centre will authorise the request and instruct the controllable node to move to the predicted midpoint. If at the predicted time the source and destination are predicted to be in range of each other the Network Command Centre will deny the request and tell the management node to back off from making the same request again and use the predicted time as the time for the management node to back off.

The existence of the controlled node changes the topology of the network by increasing the number of nodes. Nodes will often move into and out of range of other nodes. If the controlled node is used successfully it can be placed in a location that forms a communications like between the source and destination where previously there may not have been one, allowing longer transmission intervals.

The first problem that arises by using the controlled nodes is that since the topology of the network is changing dynamically; the command centre has to predict future topological state. Thus the decision where to send the controlled node is made upon prediction makes it difficult to make long-time predictions. The other problem caused by manipulating controlled nodes is that it takes time to get the outcome of the network action. The Network has to wait until the node gets to the required position. Some delays in the improvement of network performance under cognition with the presence of controlled nodes are expected.

9.1.6 Model Comparison

All the simulation models use the parameters discussed in chapter 6, 7 and 8. Any parameters that have been changed are discussed with the model problem, for example the number of nodes, the number of sources, the number of destinations for each source and the mobility model applied.

Seven model simulations were run with the basic settings shown in Table 4 - Models' Basic Settings.

The settings for the model are chosen in order to highlight network performance changes when;

- The number of nodes is increasing while all the other settings are the same. Namely, outcomes of Model 1 are compared with outcomes of Model 3; Model 2 with Model 4; outcomes within Models 6 and 7.
- The number of traffic flows is increasing while all the other settings are the same. Namely, Model 1 is compared with Model 2; Model 3 and Model 4; Model 6 and Model 7.
- The time of simulation is increased, thus Model 3 was compared with Model 5.
- A new mobility scheme was introduced, thus Model 3 and Model 6 are compared; Model 4 and Model 7 are compared.

In order to highlight the changes brought by the new techniques applied, four different simulations for each Model were run which are:

1. Network with the AODV routing protocol
2. Network with the DSDV routing protocol
3. Network with Cognition applied but no controlled nodes
4. Network with Cognition applied with controlled nodes.

The above simulations for each variation were ran, thus when discussing the results for models 3, 4 and 7, 12 simulations in each. Thus, 13 × 4=52 simulations in total have been run.

The other parameters are consistent across all scenarios, namely the node speed (1.8mps), the controlled node speed (1.8mps), the transmission radius (100m) and the data rate (11Mbps). This was to keep in line with the 802.11b wireless communications protocol specification.

In order to compare the results across the simulations the only parameters that are changed are the number of nodes and the level of cognition (none, partial or full). No cognition is applied when testing the performance of AODV and DSDV without any change. Partial cognition allows the manager node to request a change in routing protocol or a change in operating frequency. Full cognition is Partial Cognition with the introduction of controllable nodes.

Therefore the values that do not change across the simulations runs are:

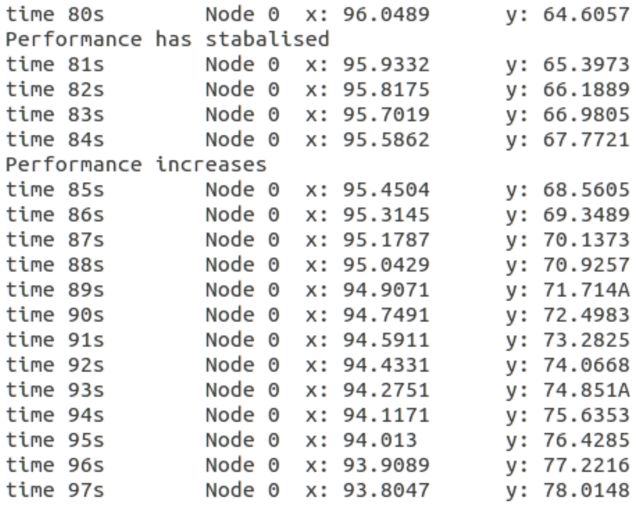
The data rate
The Simulation Time (except in Model 5)
The Transmission Radius
The Mobility Model (except in Models 6 and 7)
The Controlled Node Speed
The Node Speed (not controlled node)
The interval at which positional data is recorded

The parameters that are changed for each simulation run are:

Number of Nodes
Number of Traffic Flows
The Routing Protocol
The application of Cognitive attributes
The introduction of Controlled Nodes

9.1.7 Representation of Results

Various methods were used to check the simulation runs are accurate, one of which is mentioned in the section 9.1.2 when a visual representation of the placement of the nodes using Network Animator was depicted. Network Animator also allows the checking of the accuracy of the mobility models by showing the mobility of the nodes during the simulation. Network Animator does have some issues with correctly updating the positions of the nodes when using the random waypoint mobility model, to circumvent this, text output to report the positions of the nodes was also used to double check that they are moving as intended, a sample for Node 0 is depicted in Figure *35* - Position Output for Node 0.



```
time 80s        Node 0  x: 96.0489      y: 64.6057
Performance has stabalised
time 81s        Node 0  x: 95.9332      y: 65.3973
time 82s        Node 0  x: 95.8175      y: 66.1889
time 83s        Node 0  x: 95.7019      y: 66.9805
time 84s        Node 0  x: 95.5862      y: 67.7721
Performance increases
time 85s        Node 0  x: 95.4504      y: 68.5605
time 86s        Node 0  x: 95.3145      y: 69.3489
time 87s        Node 0  x: 95.1787      y: 70.1373
time 88s        Node 0  x: 95.0429      y: 70.9257
time 89s        Node 0  x: 94.9071      y: 71.714A
time 90s        Node 0  x: 94.7491      y: 72.4983
time 91s        Node 0  x: 94.5911      y: 73.2825
time 92s        Node 0  x: 94.4331      y: 74.0668
time 93s        Node 0  x: 94.2751      y: 74.851A
time 94s        Node 0  x: 94.1171      y: 75.6353
time 95s        Node 0  x: 94.013       y: 76.4285
time 96s        Node 0  x: 93.9089      y: 77.2216
time 97s        Node 0  x: 93.8047      y: 78.0148
```

Figure 35 - Position Output for Node 0

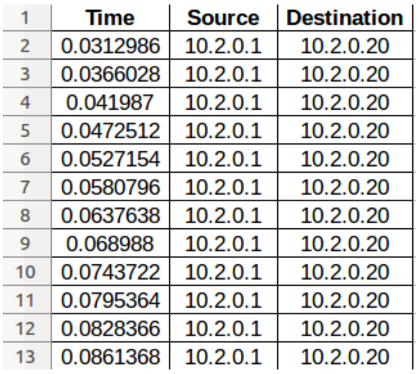
Various log files have also been created which have been explained in section 7.5. In addition to this, other outputs have been incorporated that show key events during the simulation run. In particular requests that management node(s) have issued and responses from the Network Command Centre, for example:

```
Simulation Time: 600
Number of participating nodes: 20 at 1.8mps
Number of Traffic Flows: 1 at 11Mbps
  0  to  19 starting at 0.01 finishing at 590
Cognition is ON
Number of controllable nodes: 2 at 1.8mps
Transmission Radius is: 100m
Using Random mobility scenario
```

```
Recording node positions every 5seconds.
0s Simulation Started
0s Node 0 scheduled to transmit at 0.01s to node 19
5s Node 1 identified as candidate management node for node 0
...
145s Manager 11 talks - Request: change channel
145s NCC talks - Request Acknowledged from node 11
145s NCC talks - The number of requests is: 1
145s NCC talks - Wait! Considering requests
145s  Manager 11 requests C if no action predicted receive rate is 760.833
145s  Total requests to move node      :0
145s  Total requests to change channel :1
145s  Total requests to switch protocol:0
145s NCC talks - Change channel is most popular - we are currently on channel: 2
145s NCC talks - Request Permitted to Change Channel
145s NCC talks - We have switched channel to: 1
 Number of devices is: 20
AODV NODE: 0  frequency is: 2412Hz  The channel number is: 1
AODV NODE: 1  frequency is: 2412Hz  The channel number is: 1
AODV NODE: 2  frequency is: 2412Hz  The channel number is: 1
```

The file continues and in this simulated model over 2000 lines of output are generated.

Statistics are also generated that shows all packet information; all packets that are sent, forwarded, dropped and received are recorded. In addition to this, the system is able to differentiate between the types of packets that are sent (i.e. control packets such as those generated by routing protocol requests). The amount of packets sent is not a good indicator to use for measuring network performance because a packet sent does not mean it will be received. A data file produced for received packets is depicted in Figure *36* - Sample of Received Packets. Similar data files are created for the all packets.



1	Time	Source	Destination
2	0.0312986	10.2.0.1	10.2.0.20
3	0.0366028	10.2.0.1	10.2.0.20
4	0.041987	10.2.0.1	10.2.0.20
5	0.0472512	10.2.0.1	10.2.0.20
6	0.0527154	10.2.0.1	10.2.0.20
7	0.0580796	10.2.0.1	10.2.0.20
8	0.0637638	10.2.0.1	10.2.0.20
9	0.068988	10.2.0.1	10.2.0.20
10	0.0743722	10.2.0.1	10.2.0.20
11	0.0795364	10.2.0.1	10.2.0.20
12	0.0828366	10.2.0.1	10.2.0.20
13	0.0861368	10.2.0.1	10.2.0.20

Figure 36 - Sample of Received Packets

Dropped packets are also not a good indicator of network performance because the amount of packets dropped by the different protocols that are used does not indicate the number of received packets or the receive rates of the packets. Sent packet information with received packet information to measure end-to-end delay is not used, which is one of the indicators used to measure network performance.

The number of packets received during the simulation run is what is of most interest in regards to measuring network performance. A received packet in this context is a packet with a payload (a data packet) that has been received at the destination (not by an intermediate node for forwarding). Received data packets are used as a key metric for measuring the performance of the network. Figure 37 - Received Rate per 5 seconds shows the receive rate for four simulation runs. One using the AODV routing protocol, one using the DSDV routing protocol and one with cognitive attributes applied that allowed for dynamic routing and dynamic channel assignment and full cognition. What the initial simulations show is that the network could successfully dynamically select the most appropriate routing protocol - most times.

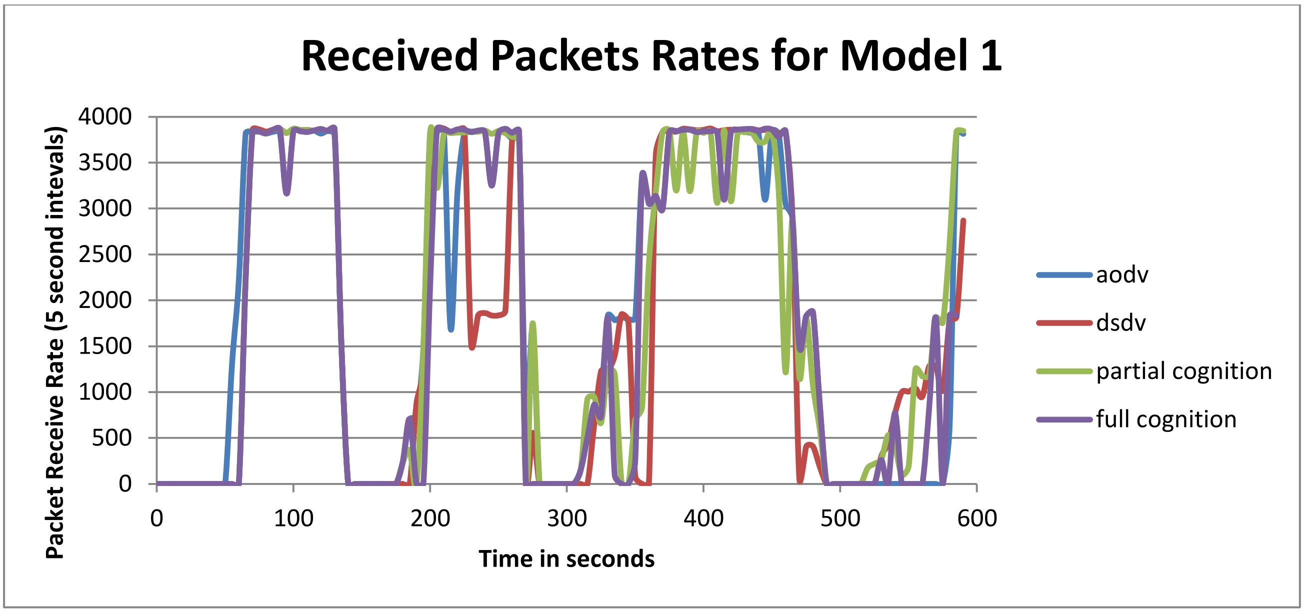


Figure 37 - Received Rate per 5 seconds | Model 1 | 10 Nodes

One can clearly see decreased transmission zones – these correspond to the times when Eagles are outside of Shark's transmission range – on the other side of the lake. Depending on the distance between the two groups, the signal reception might be completely lost as Lakes 1 and 2 are quite wide, see Figure 34 - Lake Mobility Scenario, or partially lost, since Lakes 3 and 4 are narrower.

When zooming into Figure 37 - Received Rate per 5 seconds one can see how the network with cognitive features works in a more stable way than networks without cognition, in order to illustrate this, see Figure 38 - Received Rate per 5 seconds for 200 – 265 seconds. The chosen interval is when transmission is constant for all for network simulations - both Sharks and Eagles are in the transmission range.

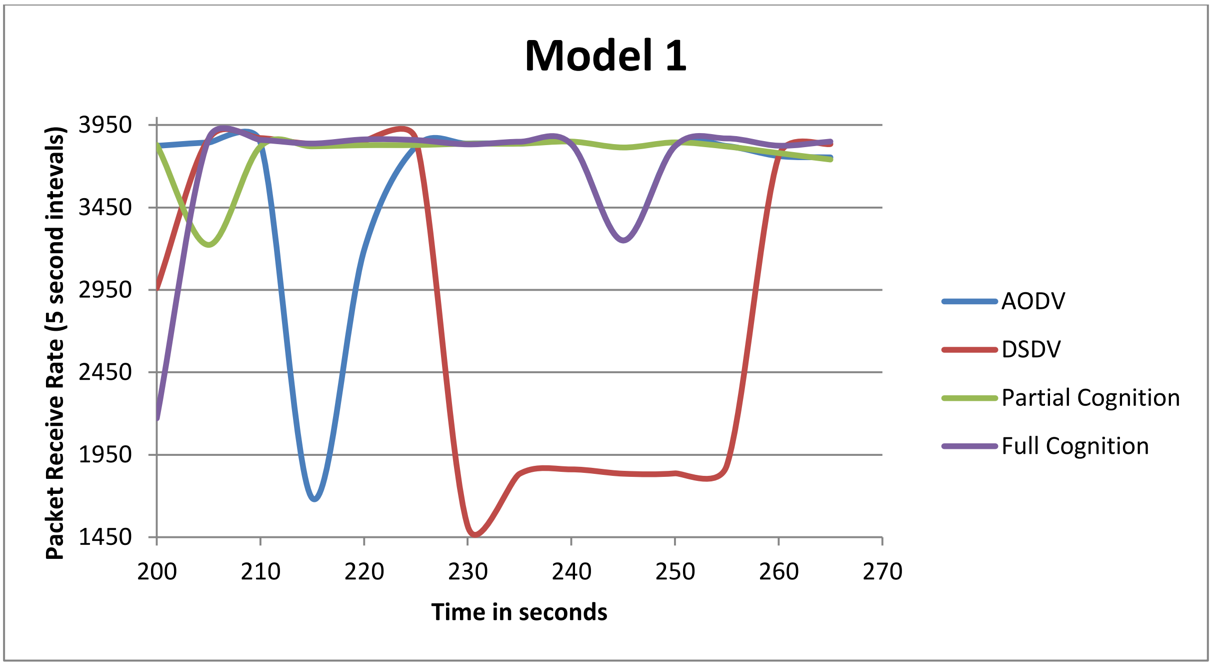


Figure 38 - Received Rate per 5 seconds for 200 – 265 seconds

It is clear from Figure 38 - Received Rate per 5 seconds for 200 – 265 seconds that DSDV (red line) was not coping well for the second half of the chosen interval, this is most likely because DSDV is a table driven proactive routing protocol, and thus, the path to a destination is already known. In this case nodes have regrouped and density has increased. Some routes are now invalid and will be until the DSDV routing tables have been updated. This corresponds to the time when both groups have already met and were waiting for the next goal. They have spent roughly half a minute waiting for the slower nodes to join. Thus the density of the network was increased during that time. The three other simulation runs: AODV, partial cognition and full cognition (with controlled nodes) were not too badly affected by higher localised density.

9.2 Model 1 (10 nodes, 1 traffic flow to 1 destination)

9.2.1 Results

The aim of this section is to present and discuss the results of the four simulation runs for Model 1 with 10 nodes, one source node transmitting to one destination node based on

Lake Scenario mobility model. The major settings are discussed in Section 6.1. This model is the first one and is the most basic one. The discussion begins with the outcomes and then the analysis in order to highlight the new phenomena obtained in the more advanced simulations later.

Received Packets

The graphs of the receiving packets rates were already presented in Figure 37 - Received Rate per 5 seconds and zoomed part of it in Figure 38 - Received Rate per 5 seconds for 200 – 265 seconds. The outputs reflect the nature of the Lake Scenario, it clearly shows areas where Eagles are outside of the range of the Sharks (where separated by the lakes). Also, at some places the transmission is still present (Lakes 3 and 4), though it highly affected by the distance between the source and receiver.

Below is the table which summarises the total number of received packets for all four simulation runs.

Number of Nodes	No Cognition		Cognition (no controlled nodes)	Cognition with controlled nodes
	AODV	DSDV		
10	221888	201710	216314	212068

Table 5 Total Number of Received Packets for Model 1

In order to understand Table 5 Total Number of Received Packets for Model 1, some explanation may be useful. As stated earlier each model is run several times in order to compare and contrast the results. In this table the first column shows how many nodes were participating in the network (this number excludes any nodes introduced as controlled nodes). The second column "No Cognition AODV" shows the simulation run using the routing protocol AODV with no cognition applied. The third "No Cognition DSDV" column shows a simulation run using the routing protocol DSDV with no cognition applied. The fourth column shows a simulation run with cognition applied, thus the network is able to dynamically select which protocol it will use and which frequency to transmit and receive data. The final column is a simulation run with full cognition applied with the additional ability that enables the network to instruct controllable nodes to move to locations determined by the manager node in collaboration with the Network Command Centre. A table is not shown for sent packets because all simulation runs are instructed to send the same amount of packets at the same times. The network clearly performs better the more data packets it receives; however, if the reader is interested in sent packets this information is included in the appendix along with dropped packets.

Analysis of AODV vs. DSDV

The first and second simulation runs (AODV and DSDV) give a good indication which protocol performs better for the duration of run. It can be quickly seen that in this run (see Table 5 Total Number of Received Packets for Model 1) that AODV (221888 packets received) performed better than the DSDV (201710 packets received), or AODV worked roughly 10% better overall.

This analysis also looks at performance in regards to the AODV routing protocol and the DSDV routing protocol; see Figure 39 - AODV vs. DSDV for Model 1. The aim of creating the Lake Scenario was also to highlight areas when AODV routing worked significantly better than DSDV and vice versa.

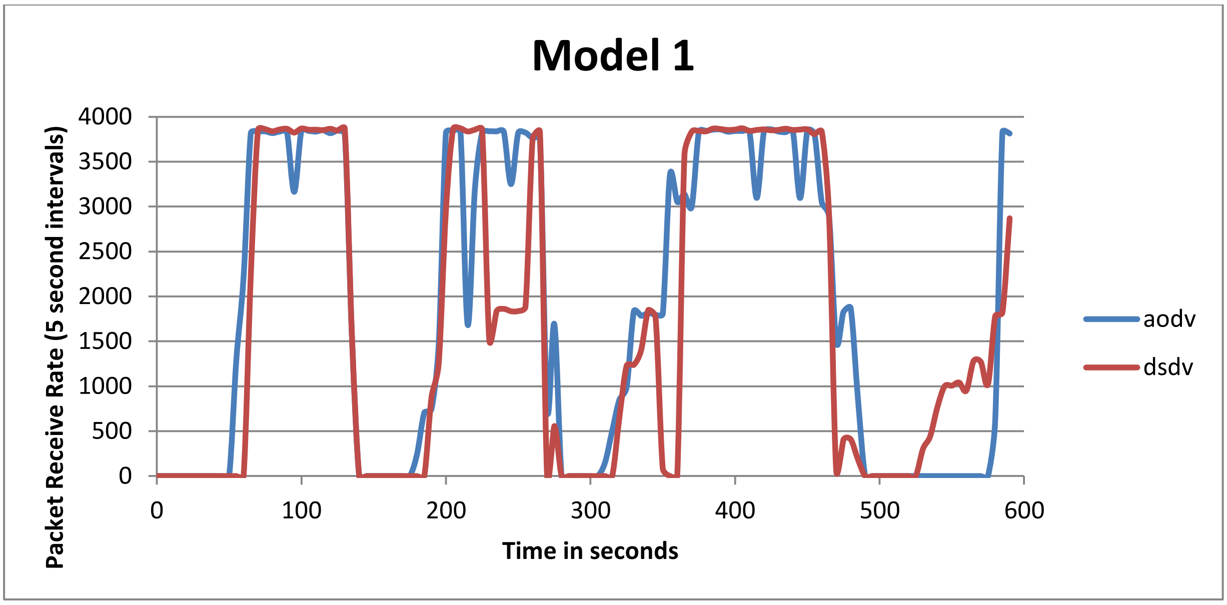


Figure 39 - AODV vs. DSDV for Model 1

This information also indicates that AODV was the first protocol that was able to establish a communications path and that the destination using the AODV protocol was able to receive packets for approximately 10 seconds before that of DSDV. It can also be seen that both destinations (the one using AODV and the one using DSDV) both were unable to receive packets at approximately 50 seconds into the simulation and performance degraded for both protocols, most likely because the source and destination became out of communications range and there were no intermediate nodes available to form the route.

As well as establishing which protocol performed better generally, it also allows the setting of a base target that allows the measuring of the performance of the same simulation run with cognition applied. Thus in this model, hopefully, when cognition is applied at least 201710 packets are received, with the intent that the figure is closer the better performing protocol which in this case is AODV and reach closer to 221888 packets.

The special mobility model (Lake Scenario, see subsection 9.1.3) was applied, such that, snapshots in time can be taken in order to establish whether AODV or DSDV performed better or worse at given intervals in time. This can be achieved because all packet information including when precisely a packet was received is recorded.

The same data file is used to map out percentages that show the performance in relation to AODV, DSDV and dynamically chosen routing. This analysis shows that AODV performed better than DSDV 20% of the simulation time. DSDV performed better than AODV roughly 52% of the simulation time. Therefore for roughly for 28% of the time both protocols performed equally well.

Analysis of Cognitive Networks' Performances

The second and the forth simulations (with partial cognition applied and with full cognition applied) has already been shown in Table 5 Total Number of Received Packets for Model 1. The third simulation run resulted in 216314 packets being received (green line in Figure 37 - Received Rate per 5 seconds). It should be noted that with cognition applied each source is assigned a suitable manager, the manager can make three requests on behalf of the node: change channel, switch routing protocol and instruct a controlled node to move to a new location. The point to note is that the manager (nor the rest of the network) can compare routing protocol performance dynamically. That is the network has no access to the information on how the network would be performing at that point in time if it were using an alternative routing protocol. Therefore the network knows if performance is stable on the current protocol, or performance is increasing or dropping. With hindsight the network could of course select the most optimum protocol every time, however, this is not realistic because one cannot travel back in time.

In this case, with cognition applied the simulation run with cognitive attributes applied was able to successfully perform better than DSDV (by 7%, see Table 5 Total Number of Received Packets for Model 1). However, the network would have performed better overall if no cognitive attributes were applied and the AODV protocol was used. This said, the benchmark figure was surpassed. What the table does not illustrate is the performance of the network during the simulation run; that is the table only shows the performance of the network for the duration.

Thus after the run the amount of time the third simulation (network run with cognitive features applied) was successful could be mapped, basing its decisions on the memory of previous actions. However, since the network does make some decisions before enough time has elapsed and enough memory of previous actions has been considered, performance is not expected to always be better than the best performing protocol. Thus, the management node is initially ill-informed as will 'guess' which action would be the most appropriate. As the management node gains in experience the choices made by the management node are better-informed. Therefore, performance does improve with time.

The reason that more packets have not been received is that in order to compare the simulations carefully it was decided to run them all for 600 seconds with the same management settings. By the latter, this means that in order for the network to get enough actions to make informed, cognitive decisions, some stabilisation time should be allowed. This is essential for the more complicated simulations. Results could be improved for this particular scenario, Model 1, i.e. by setting less stabilizing (thinking) time more packets would have been received, than the network simply working using the AODV protocol.

The forth simulation for Model 1 was the one with full cognition, i.e. the availability to use controlled nodes (blue line on Figure 37 - Received Rate per 5 seconds). The problem with the use of controlled nodes originates in the speed of the node's motion. The network has to allow sufficient time for the controlled node to get to the point of the link breakage. Moreover, the network has to learn to predict its future state correctly, re-adjust its predictions every now and then. Also, controlled nodes might be quite far away from their goal position, so the time to get to the destination might be very large. Finally, there might be a few link breakages and the number of controlled nodes might be less than the number of breakages. Thus, the Network Command Centre should first decide the priority of the breakage to be fixed; then wait until the controlled node reaches its destination (and predicting the future state), so the node might get there too late, or arrive at the wrong place. Thus, it is assumed that in order to get good results with the use of controlled nodes, there should be sufficient stabilization time. From the pre-set simulation time (600 seconds) it has been established that it might not be enough for the network to learn from its mistakes. Hence, it was decided to test the hypothesis and run the simulation for longer, 1200 seconds and the successful promising results are reported in Section 6.6. Unfortunately, due to the limited resources (time and computer memory mostly) not all the simulations could be rerun for longer than 20 minutes intervals of time.

9.3 Model 2 (10 nodes, 4 traffic flows to 4 destination)

In the second model problem is built upon the model presented in 6.1. However, in this scenario further complexity is added by introducing multiple source nodes (a total of 4). Each source communicates with its own destination. This also further complicated the model because each source would require its own managing, thus the scenario would introduce up to an additional four managers for the source nodes at any one time.

However, depending on the election process a management node may manage more than one source node. The election of management nodes is discussed in chapter 5. The plan is to analyse the performance of the network in Model 2 first, and then compare the results with the results generated by Model 1.

Another additional complexity appears with possible antagonism of the sources' aims. Each source manager is helping to increase the client source's performance by requesting some action from the network, while the requested actions might decrease the overall network performance.

In some sense they are working in collaborative mode when asking to fix the link breakage (the network will benefit from that), but if there are not enough controlled nodes, say less than link breakages, the aims become antagonistic. Game theory in this thesis has not been considered in this thesis, but, the plan to overcome this difficulty has been discussed in chapter 10 (10.9.4 Game Theory). This section presents and discusses the achieved results.

The traffic flows are initialised by calling the **`CreateTrafficFlows`** (p. 474) method. In this model a traffic flow between node 0 → 9, 1 → 8, 2 → 7 and 3 → 6 is created. The first traffic flow is scheduled to start at 0.01 seconds and continue until 10 second before the end of the simulation – 10 seconds (590 seconds). The remaining three traffic flows are scheduled to start at 3 seconds and are also scheduled to stop 10 second before the end of the simulation (590 seconds). The traffic flows were created by calling the appropriate methods: **`CreateAodvTrafficFlow`** (p. 136) and **`CreateDsdvTrafficFlow`** (p. 176).

9.3.1 Results

The discussion of results here is similar to the ones in section 9.2 for consistency when comparing results. Four simulations were run for Model 2, namely:

Normal Network with AODV
Normal Network with DSDV
Cognitive Network No controlled nodes
Cognitive Network with two controlled nodes

However, there are some significant differences. The final numbers of received packets for 600 seconds are presented in Table 6 - Total Number of Received Packets for Model 2 and will be compared to Table 5 Total Number of Received Packets for Model 1.

Number of Nodes	No Cognition		Cognition (no controlled nodes)	Cognition with controlled nodes
	AODV	DSDV		
10	431020	426137	433536	420179

Table 6 - Total Number of Received Packets for Model 2

The first point to mention is the presence of multiple traffic flows. The latter increased the total amount of packets sent by the network by 94% on AODV and 111% on DSDV (compare with section 6.2.: AODV 431020 vs. 221888, DSDV 426137 vs. 201710). The best performing protocol is still AODV (by 1.1%, total number is 431020) when no cognition is applied.

With cognition applied to the network, winning result were achieved, and resulted in 433536 received packets, 0.5% more than AODV and 1.7% more than DSDV. AODV has worked better than DSDV for 49% of the whole of the simulation time, while DSDV 47% (again, 4% of the time they were equally good/bad). This is exactly what was wanted from the mobility scheme based on the Lake Scenario. No need to say, the achieved results are extremely promising. Running the simulation for longer would have brought further increased performance. The dynamically changed protocol and channel (partial cognition) have increased the performance by almost 2%. The same number of nodes did not change the overall topology greatly, so AODV was still working better (that is not going to be the case on all further scenarios).

During the simulation, AODV was performing better 49% of the time, while DSDV was better for 47% of the time; thus they performed equally for 4% of the time.

The received packets rates are graphed in Figure 40 - Received Packets Rates for Model 2. Similarly to Model 1, all four simulations' receiving rates have been plotted on the same graph for the full duration of the simulations run (600 seconds).

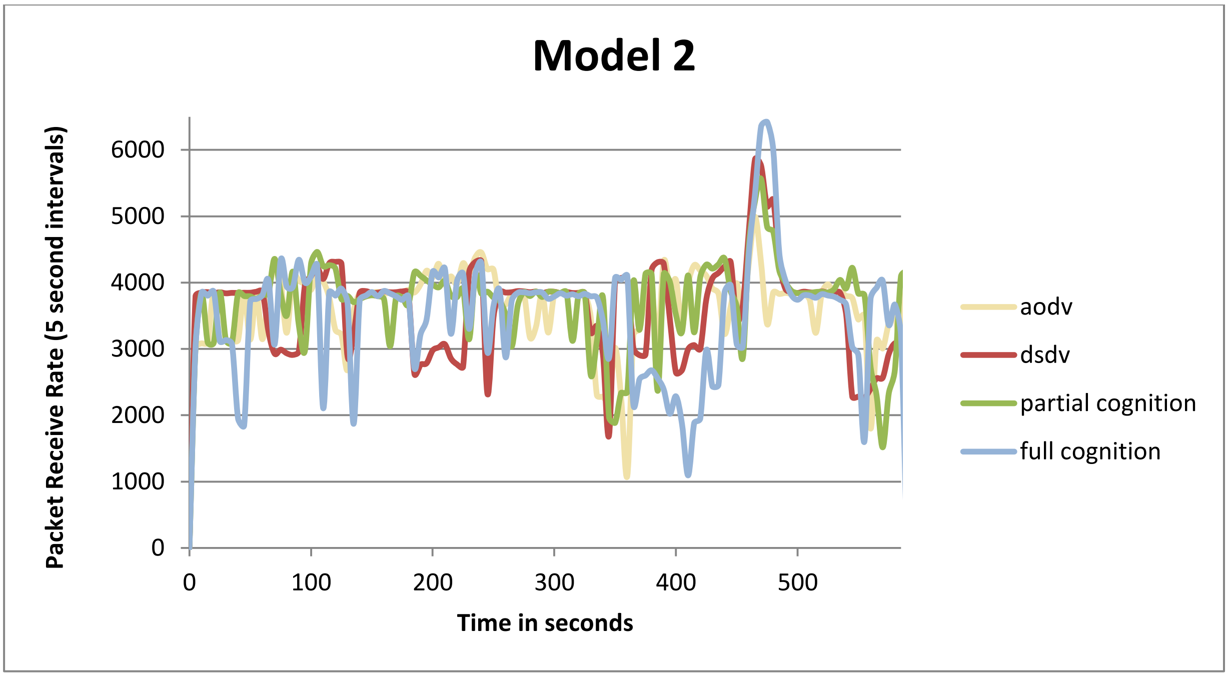


Figure 40 - Received Packets Rates for Model 2

The first difference comparing to Figure 37 - Received Rate per 5 seconds is that even though the same Lake Scenario is being used, there are now no clear areas where there was no transmission. The reason for this is quite clear. In Model 2 there are four source nodes transmitting to four destinations, namely nodes 0 → 9, 1 → 8, 2 → 7 and 3 → 6. However, recall how the nodes have been split into two groups – Eagles and Sharks: the first group contain roughly one third (the floor of the total number of nodes in the network divided by three), which is three in the case of ten nodes (floor($\frac{10}{3}$)=3), thus the first group contains nodes 0,1 and 2, while the second group contains all the rest, namely nodes 3 – 9. Thus source node 3 is in the same group with node 6, its own destination. The latter explains absence of zones of no transmission.

In order to see the results better, the final stage of the simulation is zoomed in, see Figure 41- Received Rates for Model 2 (Final Stage).

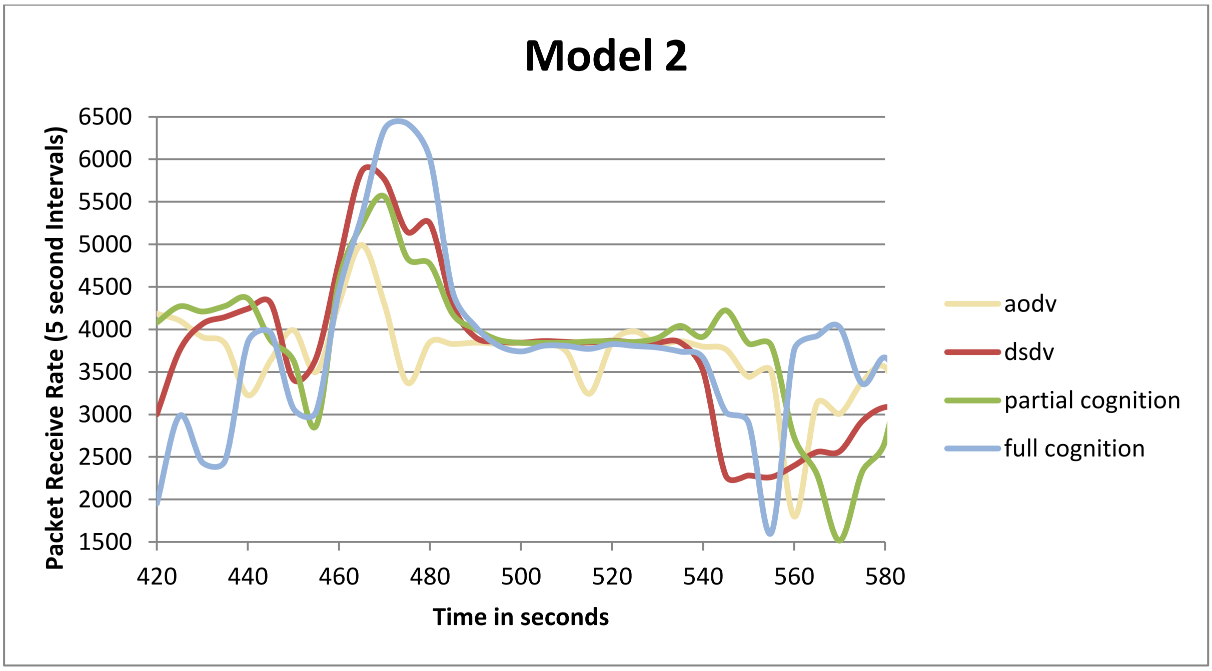


Figure 41- Received Rates for Model 2 (Final Stage)

The antagonism and multiple requests for the controlled node allowed network to make better decisions on using the controlled node. Here, the disadvantage of individual goals is clear, once there is only one manager, the network acknowledges the requests and does not get a chance to choose a better decision, while with many requests, it manages to choose the better decision. It is shown in the Figure 41- Received Rates for Model 2 (Final Stage) that the presence of cognition and availability of controlled nodes brings performance better than on any of the chosen protocols, i.e. AODV, DSDV (see blue and green lines appearing above red and yellow lines).

9.4 Model 3 (20, 40, 80 nodes, 1 traffic flow to 1 destination)

The third model problem is built on the model presented in section 6.2. In this model, more nodes are introduced into the simulation, namely 20, 40 and 80 instead of 10 in Model 1. The placement strategy and the mobility model are the same ones that are discussed in section 6.1; but there are variations when more nodes are introduced. Due to the increased number of nodes in the network, in the first set of simulations with 20 nodes, node 0 transmits to node 19. In the second set with 40 nodes, node 0 transmits to node 39, and in the third set with 80 nodes, node 0 transmits to node 79. This is designed so that a node is not transmitting to another node within its defined group. The latter makes Model 3 more similar to Model 1 in terms of the presence of no-transmission zones (Lakes), see Figure 42 - Receiving Rates for Model 3 with 20 nodes, Figure 43 - Receiving Rates for Model 3 with 40 nodes, Figure 44 - Receiving Rates for Model 3 with 80 nodes.

9.4.1 Results

As previously, four simulations per variation were run:

Normal Network with AODV
Normal Network with DSDV
Cognitive Network No controlled nodes
Cognitive Network with two controlled nodes

In this model, the number of received packets when the simulations conclude is reported in Table 7 - Received Packets Summary for Model 3.

Number of Nodes	No Cognition		Cognition (no controlled nodes)	Cognition with controlled nodes
	AODV	DSDV		
20	198227	115587	116905	176963
40	57443	213395	175392	187815
80	182499	105381	125311	190288

Table 7 - Received Packets Summary for Model 3

In Model 3, the number of the nodes in the network is significantly increased, hence changing the network topology in comparison with Model 1 and 2. The reason for introducing Model 3 was based on the idea to compare it with Model 1 mostly since the only difference in their settings is the number of nodes; the number of traffic flows is the same. Thus in order to discuss the outcomes, the total numbers of received packets is compare first. This is shown in Table 5 Total Number of Received Packets for Model 1 and Table 7 - Received Packets Summary for Model 3.

In model 3 the second experiment consisted of 40 nodes which makes the topology of the network much more dynamic because there are more nodes moving, this had an interesting impact on the performance of the AODV protocol. Table 7 - Received Packets Summary for Model 3 shows that for the 40 nodes run, for 1 traffic flow DSDV is performing better than AODV.

The presence of multiple nodes makes it more difficult to discover routes for the packets so the performance varies significantly (AODV received 221888 packets with 10 nodes, and then 198227 with 20 nodes), dropping after (AODV gets 57443 with 40 nodes) and getting better with 80 nodes (182499). However, DSDV performs differently, increasing first, and then decreasing constantly with growing number of nodes, namely 201710 with 10 nodes, 115587 with 20 nodes, 213395 with 40 nodes and 105381 with 80 nodes. It should be noted that in the Model 3 scenario with 40 nodes, DSDV performs better than AODV, but on Model 5 with extended simulation time (Model 5 is based on Model 3); AODV again overtakes the role of the better performing protocol (see section 6.6).

During the simulation duration for the run with 20 nodes, AODV was performing better 40% of the time, while DSDV was better for 30% of the time; thus they performed equally for 30% of the time. During the simulation duration for the run with 40 nodes, AODV was performing better 7% of the time, while DSDV was better for 65% of the time; thus they performed equally for 28% of the time. During the simulation duration with 80 nodes, AODV was performing better 40% of the time, while DSDV was better for 21% of the time; thus they performed equally for 39% of the time.

However, the simulations with cognition applied behave quite differently. In Model 3 and in all cases (20, 40 and 80 nodes) the cognition applied does enhance the overall performance of the network. This is true with simulation runs where both, partial and full-cognition was applied. The most suitable routing protocol for a given scenario is unknown; therefore, a benchmark figure that relates the performance of the less well performing protocol is used when drawing comparisons. First, partial cognition (no controlled nodes present) is analysed. In comparison with the benchmark the following results are observed: an increase of +7% over DSDV with 10 nodes (Model 1, see section 6.2), +1.1% over DSDV with 20 nodes, +305% over AODV with 40 nodes, and +118% over DSDV with 80 nodes. The full cognition gives constantly increasing performance: +5% over DSDV with 10 nodes (Model 1, see section 6.2), +53% over DSDV with 20 nodes, +226% over AODV with 40 nodes, and +80% over DSDV with 80 nodes.

It should be highlighted that in the last simulation of Model 3, full cognition brought 4.2% better performance comparing to the best working protocol, AODV. Since all the percents are positive, the results are all above the benchmark. This shows that without accessing any information about which protocol would be most suitable at that given time, the network learns to make cognitive decisions bringing benefits, extra received packets, thus increased network performance.

Now, the rates of receiving packets as a function of time are discussed, modelled and represented in Figure 42 - Receiving Rates for Model 3 with 20 nodes, Figure 43 - Receiving Rates for Model 3 with 40 nodes and Figure 44 - Receiving Rates for Model 3 with 80 nodes.

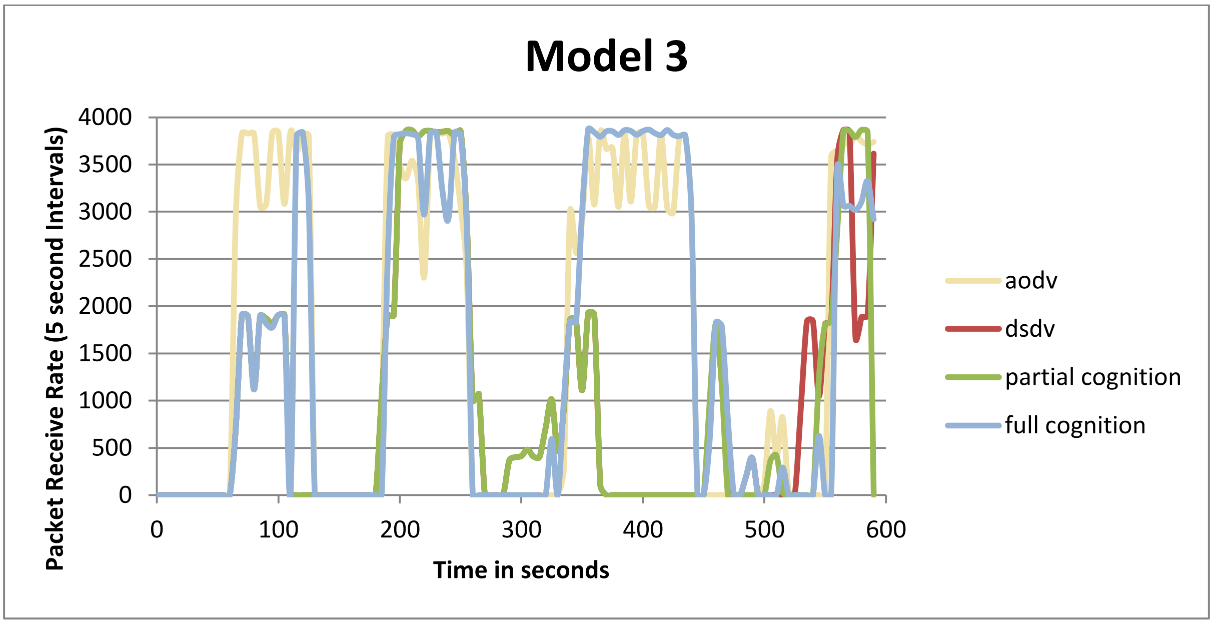


Figure 42 - Receiving Rates for Model 3 with 20 nodes

When examining Figure 42 - Receiving Rates for Model 3 with 20 nodes, it is clear, that the zones of non-transmission are clearly seen. However, in the full cognition simulation (blue line) the network requires time to learn how to use controlled nodes properly. In this case, the default simulation time of 600 seconds is not long enough for cognition to work effectively. There has not been enough time to learn and analyse the outcomes of decisions taken.

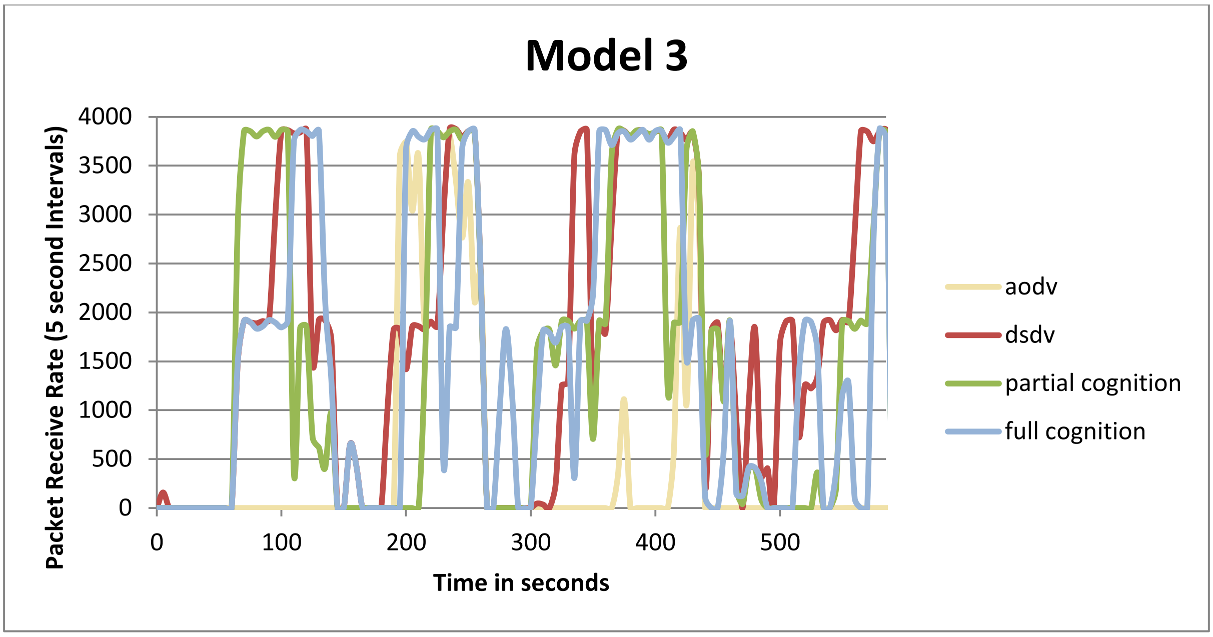


Figure 43 - Receiving Rates for Model 3 with 40 nodes

Surprising results were brought by AODV on the simulation in Model 3 with 40 nodes. One can see clear delays in increasing of performance for the full cognition network. It is believed that this is the result of back-off times given to the manager nodes by Network Command Centre based on requests to move the controlled nodes. This simulation with 40 nodes was chosen to be run for the extended time of 1200 seconds to find some evidence which would support the hypothesis that performance of a cognitive network increases provided that there is enough time to learn (see section 6.6).

Next presented are the receiving packets rates for Model 3 with 80 nodes, see Figure 44 - Receiving Rates for Model 3 with 80 nodes. The presence of such a significant number the mobility of nodes is greater which in turn makes the topology of the network much more dynamic. This has an impact on paths from source to destination, the higher the mobility the more likely routes from source to destination will have changed. In this case previous routes established may become out-dated which results in a degradation of performance. However, full cognition performs extremely well under such uncomfortable conditions.

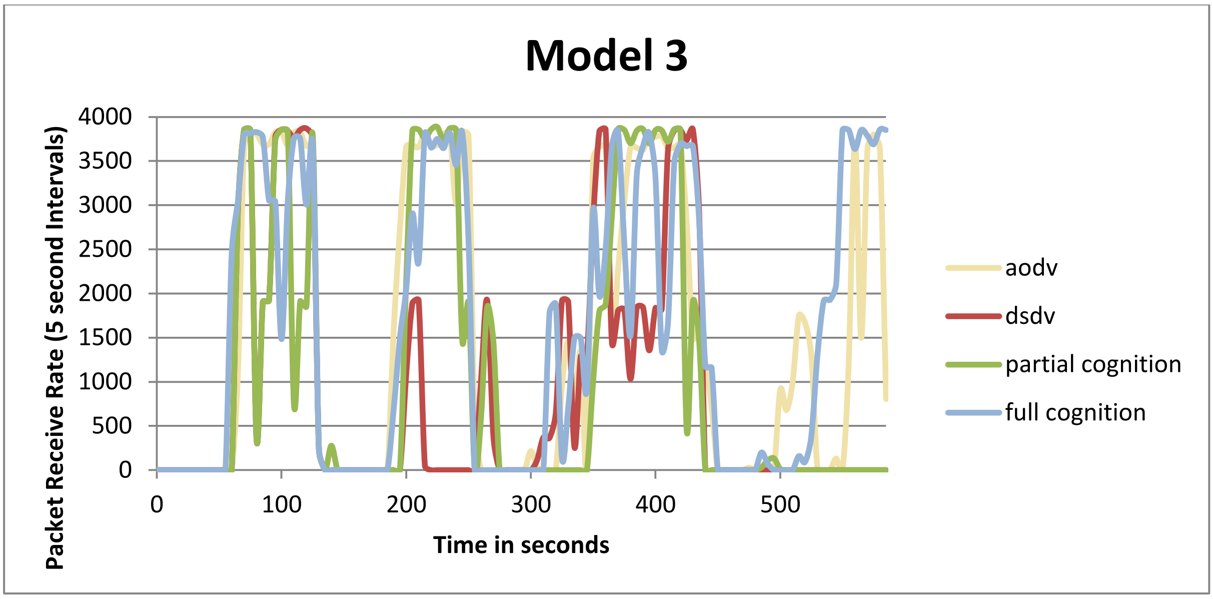


Figure 44 - Receiving Rates for Model 3 with 80 nodes

Finally, the outcomes of Model 3 also are very satisfying and promising.

9.5 Model 4 (20, 40, 80 nodes, 4 traffic flows to 4 destinations)

Model 4 was introduced mainly in order to be compared with Model 2. It allowed for the comparison against what happens when the number of nodes in the network is increased significantly. Reporting the results in Model 4 is similar to describing results for Model 3 to maintain consistency, the main difference between comparing Model 4 vs. Model 2, and comparing Model 3 vs. Model 1 is the same – the number of nodes. However, recall one major trait of Model 2 – the splitting into two groups (Eagles and Sharks) with 10 nodes makes transmission possible almost constantly throughout simulation (one of the sources belongs to the same group as its own destination) allowing better performance. Once the number of nodes is increased from 10 to 20 (40, 80), this feature is lost. Source nodes belong to eagles, their destination nodes belong to Sharks. Again, it is expected to see zones of no-transmission.

9.5.1 Results

As previously, four simulations per variation were run:

Normal Network with AODV for 20, 40, 80
Normal Network with DSDV for 20, 40, 80
Cognitive Network No controlled nodes for 20, 40, 80
Cognitive Network with Controlled nodes for 20, 40, 80

Thus, 12 simulations have been run for Model 4 and the results are presented next. The total amounts of received packets per simulation are shown in Table 8 - Received Packets Summary for Model 4.

Number of Nodes	No Cognition		Cognition (no controlled nodes)	Cognition with controlled nodes
	AODV	DSDV		
20	277195	220839	255890	248944
40	251010	173898	201757	239349
80	192928	126343	198497	194025

Table 8 - Received Packets Summary for Model 4

In Model 4, again, the number of nodes in the network is significantly increased, hence changed network topology comparing with Model 1 and 2. Since the major goal in this section was to compare Model 4 with Model2, in order to discuss the outcomes, the total numbers of received packets in Table 6 - Total Number of Received Packets for Model 2 and in Table 8 - Received Packets Summary for Model 4 are linked. The presence of multiple nodes makes it more difficult to sort out the routing for the packets so the performance drops significantly (AODV received 432020 packets with 10 nodes, and then 277195 with 20 nodes), dropping after to 251010 with 40 nodes and to 192928 with 80 nodes. However, DSDV performs differently, decreasing constantly with growing number of nodes, namely 426137 with 10 nodes, 220839 with 20 nodes, 173898 with 40 nodes and 126343 with 80 nodes. During the simulation duration with 20 nodes, AODV was performing better 55% of the time, while DSDV was better for 25% of the time; thus they performed equally for 20% of the time.

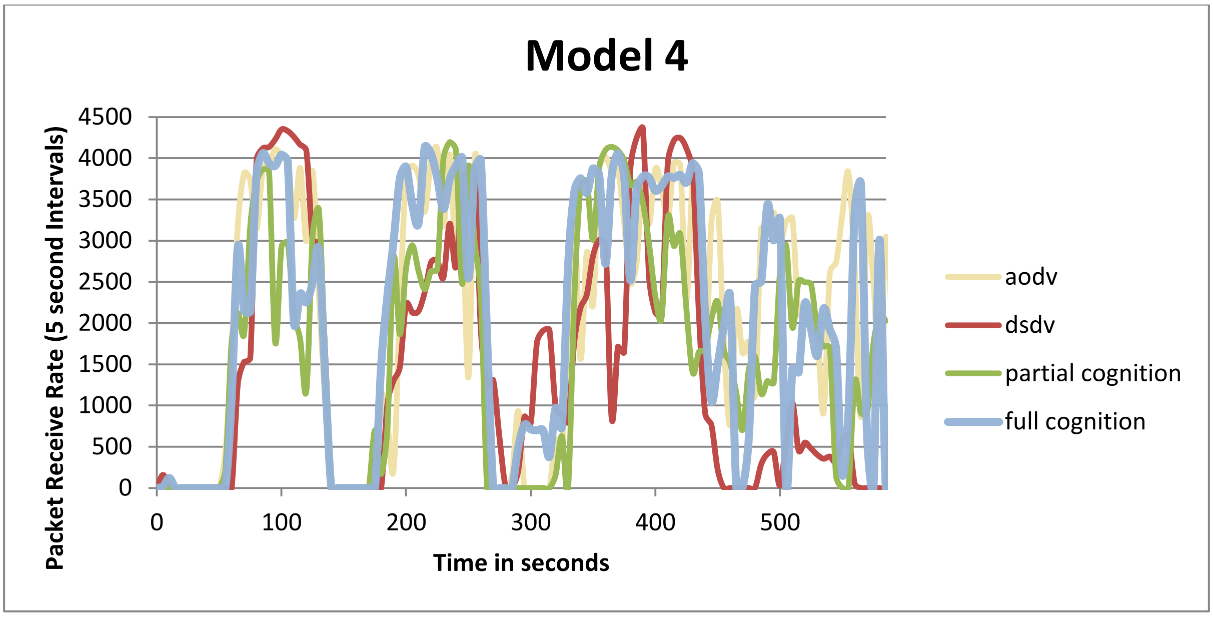


Figure 45- Receiving Rates for Model 4 with 40 nodes

Below are the graphs presenting the receiving rates for the last set of simulations in Model 4 (Figure 46 - Receiving Rates for Model 4 with 80 nodes and Figure 47 - Receiving Rates for Model 4 with 80 nodes (final stage)).

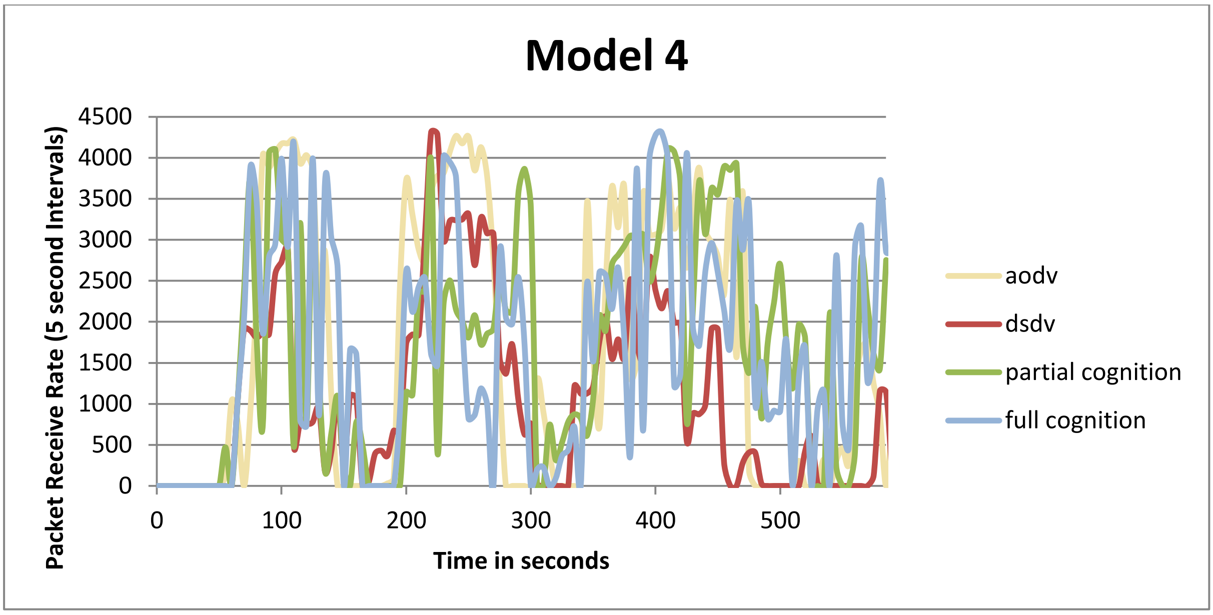


Figure 46 - Receiving Rates for Model 4 with 80 nodes

When analysing the final stage of the simulation run for Model 4 with 80 nodes, see Figure 47 - Receiving Rates for Model 4 with 80 nodes (final stage).

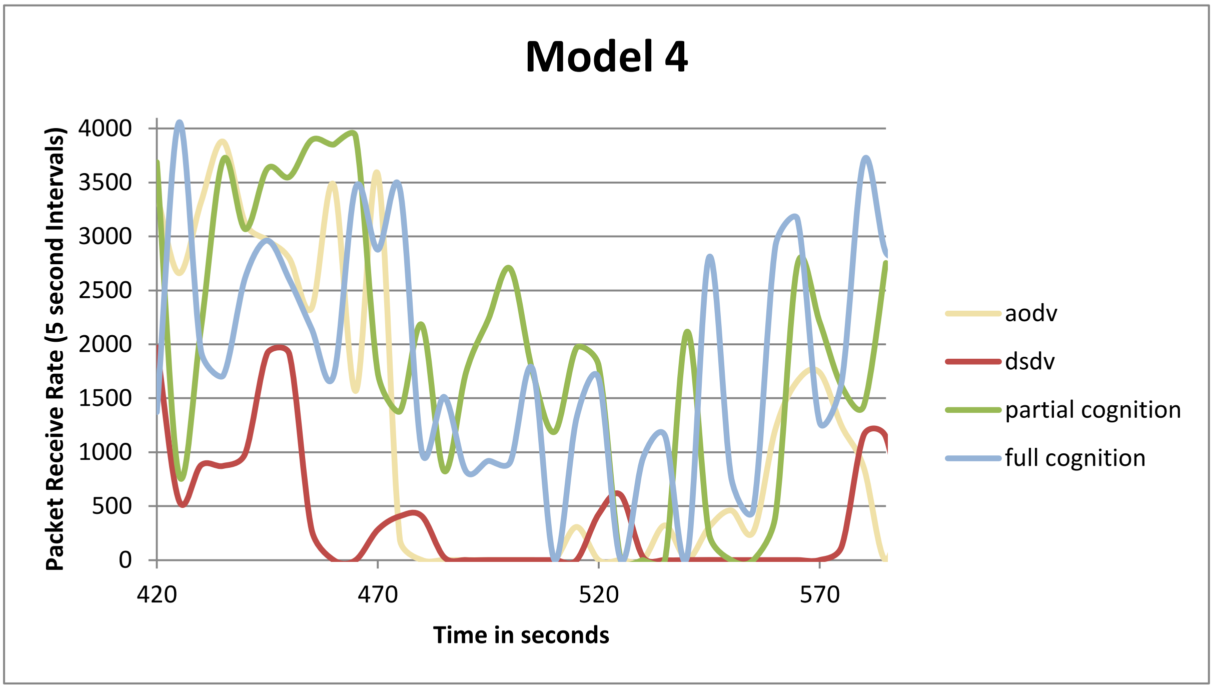


Figure 47 - Receiving Rates for Model 4 with 80 nodes (final stage)

As one can see, the results are very promising. The cognitive network (both with partial and full cognition) performs well above both non-cognitive networks (AODV and DSDV).

9.6 Model 5 (40 nodes 1200 seconds 1 traffic flow)

Model 5 takes a special place among the rest of the models as it is the only model that is run for extended time (20 minutes). Apart from simulation time, Model 5 is the direct extension of the 40 nodes simulated in Model 3. Therefore all the main characteristics of Model 5 have been already discussed in section 6.4. The receiving rates are displayed in Figure 48- Receiving Rates for Model 5.

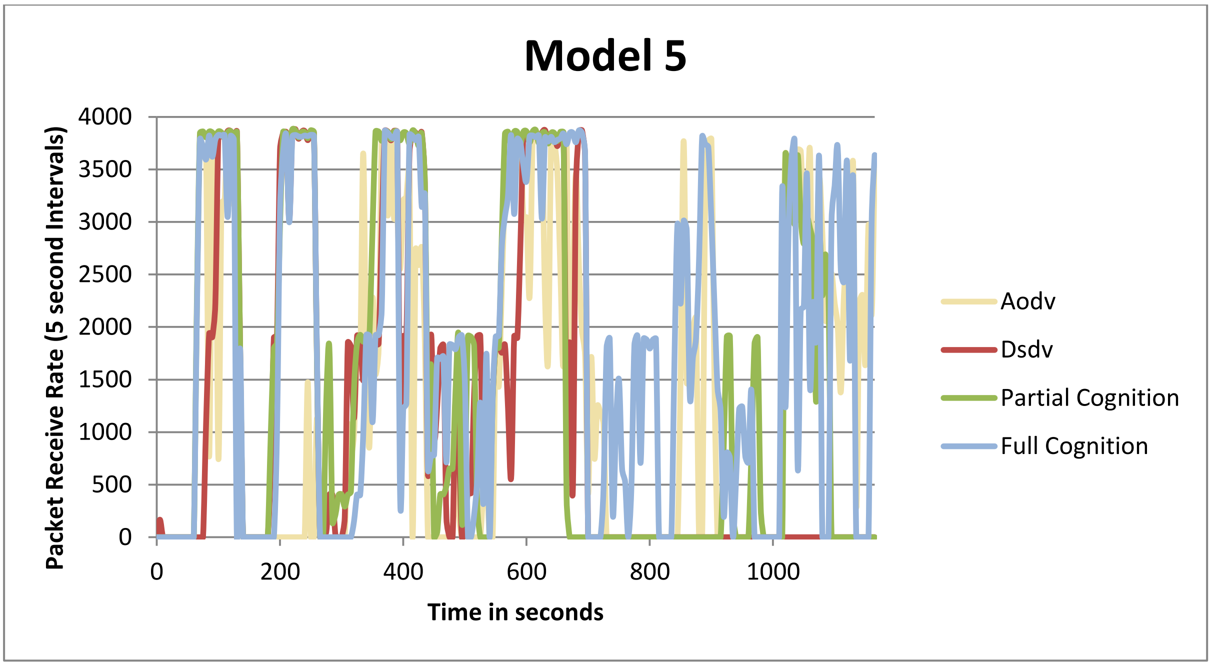


Figure 48- Receiving Rates for Model 5

To avoid repeating content already presented, the main effects are discussed. The total numbers of received packets are presented in Table 9 - Received Packets Summary for Model 5. The new totals have been combined with the totals reported above in the second row in Table 7 - Received Packets Summary for Model 3 to simplify analysis.

The totals received by networks with applied cognition (both, partial - 331910 and full - 398434) are very satisfying. If compared with the benchmark (the worse performing routing protocol) it can been seen that partial cognition performs 25% better than DSDV, and 7%

better than AODV, although full cognition performed 47% better than DSDV, 26% better than AODV, and 17% better, than partial cognition.

Number of Nodes	No Cognition		Cognition (no controlled nodes)	Cognition with controlled nodes	Time (seconds)
	AODV	DSDV			
40	57443	213395	175392	187815	600
40	308938	264677	331910	389434	1200

Table 9 - Received Packets Summary for Model 5

Due to lack of resources such as time and computer memory (this simulation runs for 8 hours in real-time and for each scenario several simulations had to be run) extended in time simulations could not be run for more models. The outputs of Model 5 are not enough evidence to be sure that cognitive network increases performance provided enough time, but the randomly chosen simulation gave very good results.

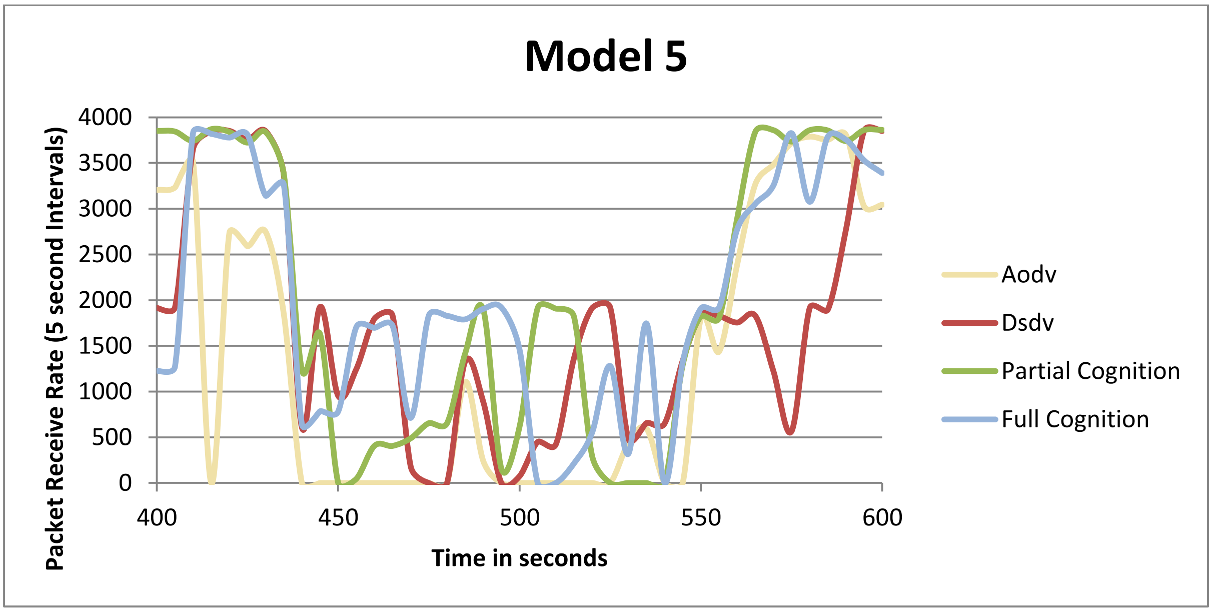


Figure 49 - Receiving Rates for Model 5 400-600 seconds

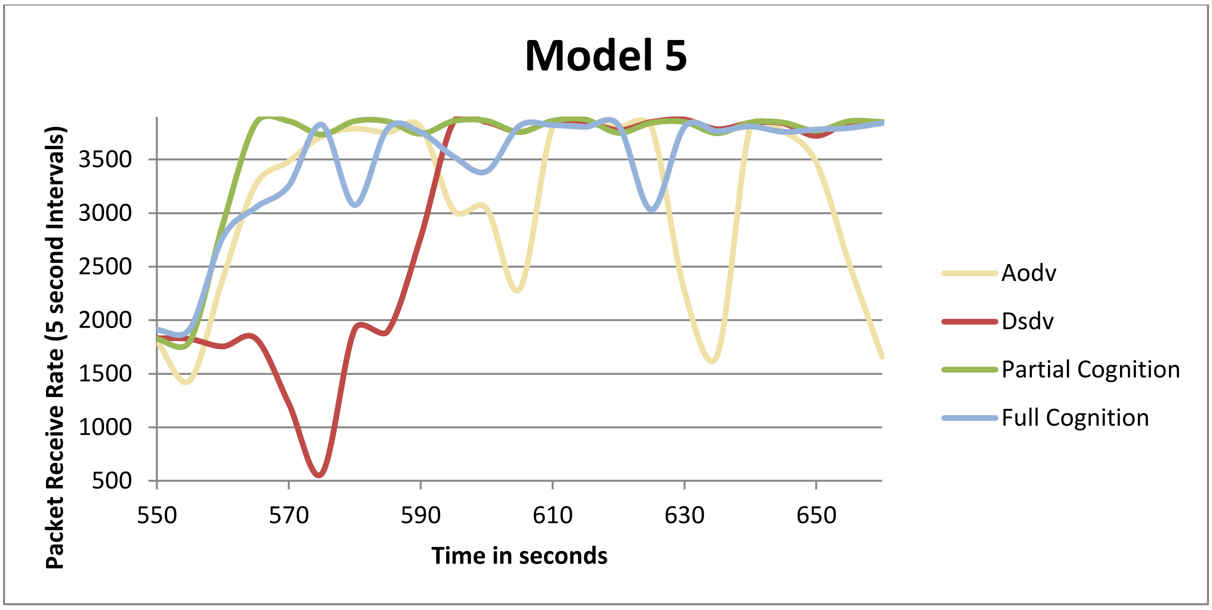


Figure 50- Receiving Rates for Model 5 550 - 650 seconds

Above are graphs of received packets as functions of time plotted for the partial duration of the simulation (Figure 49 - Receiving Rates for Model 5 400-600 seconds and Figure 50- Receiving Rates for Model 5 550 - 650 seconds) , in this chart it is evident that there is a period prior to learning (before 550 seconds), and this period the network has become much more stable for the simulation runs that consisted of partial and full cognition and clearly outperforms the simulation that used the AODV routing protocol when no cognition was applied.

9.7 Model 6 (20 and 40 nodes 1 traffic flow) RWP

Models 6 and 7 are special among all models since they were run with the well-established random Way Point mobility model. This is one of the standard models used to test new ideas implemented on networks. Model 6 discussed here are based on 20 and 40 nodes with one source node transmitting to their own single destination nodes. The receiving rates for 20 nodes are shown on Figure 51- Receiving Rates for Model 6 with 20 nodes. One can see the continuity of transmission – there are almost no clear gaps in receiving.

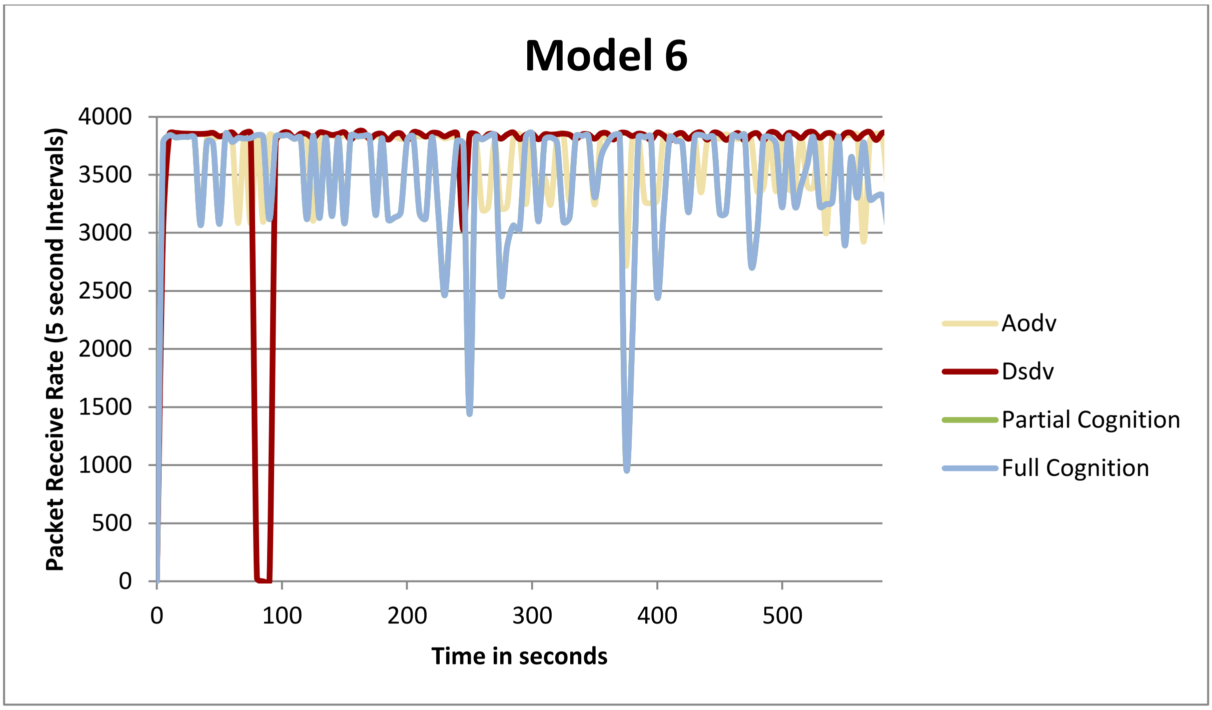


Figure 51- Receiving Rates for Model 6 with 20 nodes

The total number of received packets can be found in Table 10 - Received Packets Summary for Model 6 for both networks: with 20 nodes and with 40 nodes.

Number of Nodes	No Cognition		Cognition (no controlled nodes)	Cognition with controlled nodes
	AODV	DSDV		
20	432706	440208	411008	411008
40	268638	270352	261117	280297

Table 10 - Received Packets Summary for Model 6

During the simulation duration with 20 nodes, AODV was performing better 20% of the time, while DSDV was better for 74% of the time; thus they performed equally for 6% of the time. During the simulation duration with 40 nodes, AODV was performing better 30% of the time, while DSDV was better for 60% of the time; thus they performed equally for 10% of the time.

It is interesting, that in both cases (20 and 40 nodes), DSDV is performing better than AODV by 1.7% and 0.6% correspondingly. From Figure 51- Receiving Rates for Model 6 with 20 nodes one can see that DSDV is very stable, having only two significant performance drops, while AODV is having constant drops, but they are less dramatic.

Simulations with cognition did not prove to be good for 20 nodes, since the network seems to be coping quite well with the network topology without cognition applied. There were no link breakages during the simulation, thus the use of controlled nodes did not bring any new packets. In fact, it could be argued that the additional overhead required in order to implement cognition in networks with a low number of nodes (20) and with relatively low mobility (source node was always in range of destination) that it could have a detrimental effect on the performance of the network. Due to the stable performance of both AODV and DSDV cognition does not seem to be bringing any improvements. However, once the number of nodes in the network increase from 20 to 40, both AODV and DSDV routing protocol performance degrades. The number of packets received with the AODV routing protocol dropped from 432706 to 268638, and with the DSDV routing protocol from 440208 to 270352. This drop may initially seem strange; however, the source node and the destination node are in different groups (source node is in the Eagle's group, destination is in the Shark's group). Therefore, routes had to depend on intermediate nodes. At the same time, partial cognition brought 261117 packets and with full cognition 280297, the latter is 4% better than AODV and 3.6% better than DSDV.

Interesting thing happens around the half-way point of the simulation for 40 nodes, see Figure 52 - Receiving Rates for Model 6 with 40 nodes, where it would appear after 300 seconds of good performance, the network then becomes quite erratic with regards to performance.

At approximately 300 seconds, performance degrades, some nodes in each group have reached a way-point and are stationary whilst they wait for the slower nodes in their groups to arrive. During this period the paths from source to destination are erratic as each new member of the group reaches the way-point they become out of range and a new path has to be established by using nodes that are still in range and have not yet reached the way-point.

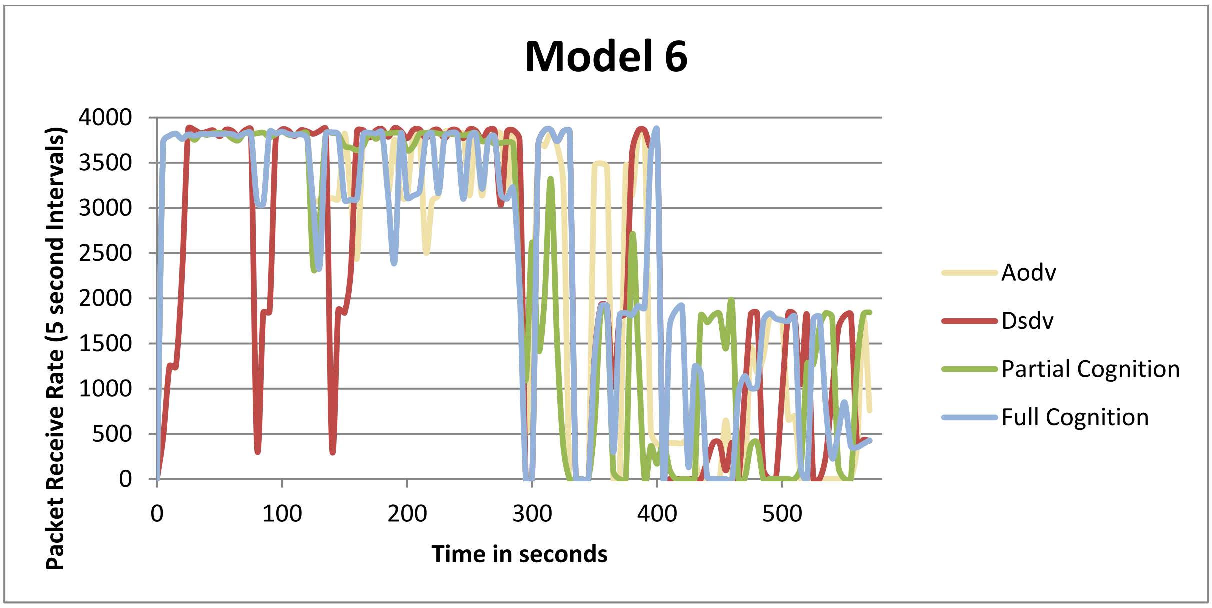


Figure 52 - Receiving Rates for Model 6 with 40 nodes

Finally, the obtained results are very satisfying. The procedure tested on the specially created mobility model proves to be useful on the Random Way Point model as well. Next the cognitive features are tested on the last scenario, Model 7.

9.8 Model 7 (20 and 40 nodes 4 traffic flows) RWP

Model 7 can be seen as a direct expansion of the simulation stated in Model 6, more traffic flows are introduced in a similar way that they were for Models 2 and 4.

Number of Nodes	No Cognition		Cognition (no controlled nodes)	Cognition with controlled nodes
	AODV	DSDV		
20	389332	295671	377527	361732
40	340151	260172	260208	272804

Table 11 - Received Packets Summary for Model 7

During the simulation duration with 40 nodes, AODV was performing better 62% of the time, while DSDV was better for 32% of the time; thus they performed equally for 6% of the time. In Model 7 AODV performs better than DSDV. The same pattern as in Model 6 (see Figure 51- Receiving Rates for Model 6 with 20 nodes) takes place here – AODV performs worse than DSDV more often, however the magnitude of the drops is less on average than the drops in performance of the network with the DSDV routing protocol (see Figure 53 - Receiving Rates for Model 7 with 20 nodes). Both routing protocols are coping well with 20 nodes randomly spread over the “World”, thus it seems to be difficult to improve already good performance.

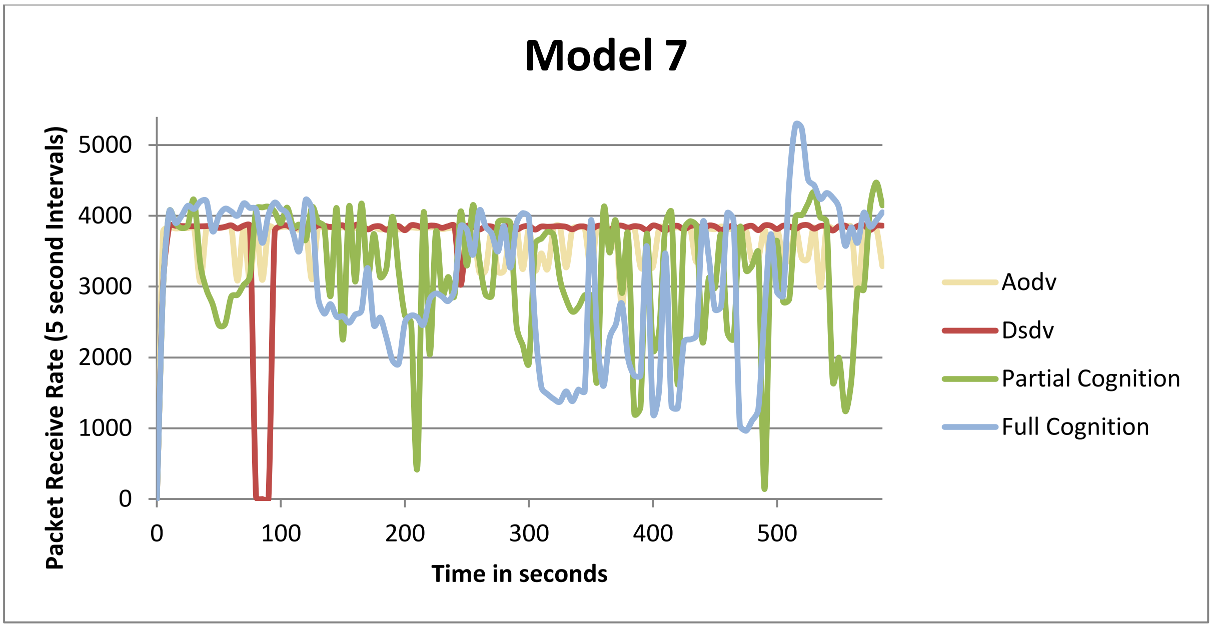


Figure 53 - Receiving Rates for Model 7 with 20 nodes

However, again when the number of nodes increases from 20 to 40, the standard networks on both AODV and DSDV start degrading in performance, see Table 11 - Received Packets Summary for Model 7. Receiving rates oscillate with large amplitude (see Figure 54 - Receiving Rates for Model 7 with 40 nodes). At the same time network with cognitive features applied, start bringing some profit in terms of received packets.

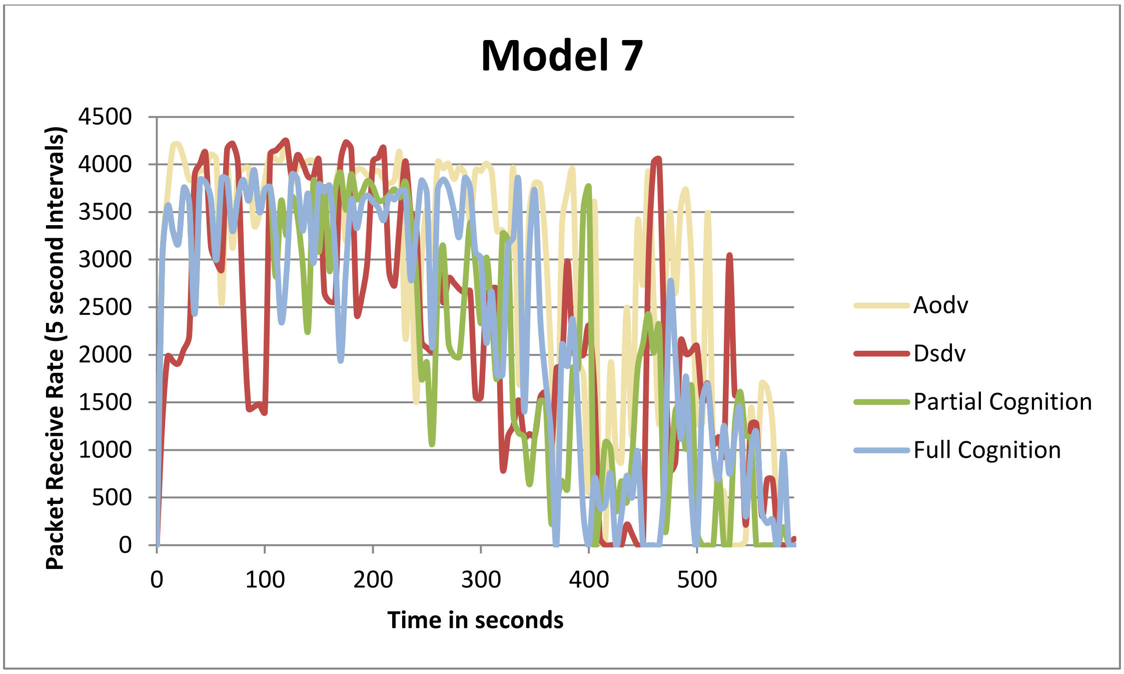


Figure 54 - Receiving Rates for Model 7 with 40 nodes

9.9 Conclusion

The initial simulated network model consisted of one source node and one destination node with a total number of ten nodes. This allowed one manager to make decisions on behalf of the source node. There was no contention in regards to required resources and this allowed the network to select the better performing routing protocol most of the time. This allowed a higher total rate of received packets. However, if there are a few or many source nodes their goals may contradict. When there are multiple source nodes, the managers' request(s) should be authorised by the network command centre once evaluating current and predicted performance. Moreover, since the network is cognitive, i.e. requires using short-term and long-term memory, therefore, some delays in the network making better-informed decisions are expected.

The summary table of receiving rates shows that in the majority of cases the cognitive network was performing quite closely to that of the better protocol at a given moment in time. The network is unable to compare the performance between multiple protocols at the same given moment in time and has to rely on the previous network state, the current network state and make a prediction of the future network state.

The controlled nodes did not perform perfectly well in some scenarios and this has revealed shortcomings in the algorithm that was used. The controlled node was sent to a location that indicated mid-point between the source node and the destination node. It would have been more efficient to determine where the link breakage had occurred in the path (which intermediate node?) and instruct the controllable node to move to a location based on where the link had failed. However, if the network command centre knows of all the current positions of the nodes within the network, it would not be difficult to send controlled nodes to fix link breakages by making calculations based on convex analysis and graph theory. However, for now, access to full network state was deliberately avoided and the decisions that the Network Command Centre made were based on the data available to each local manager.

The Network Command Centre is predicting positions of the nodes in order to send the controlled node to the required destination; however, it takes a certain amount of time for the controlled node to arrive at its destination, thus, a risk to move a node in order to fix a link breakage is taken because of the dynamic nature of the network topology.

The realistic scenarios (i.e. the ones that used the Lake Scenario) showed significant advantage of using the AODV routing protocol compared to the DSDV routing protocol.

The data communications network with cognitive features applied in this work showed that performance was constantly better than that of the worse performing protocol, bringing more received packets. Moreover, Model 5 gave promising results supporting that a data communications network with cognitive features needs longer stabilization time to outperform the AODV routing protocol (or, in general, the best working routing protocol).

It should be stressed that Model 6 proved that one should not assume that AODV will always be the better performing routing protocol, since DSDV was constantly working better for both 20 and 40 nodes when the Random Way Point mobility scheme was applied. Thus the main conclusion is that in all the cases choosing cognitive network was the solution beneficial for the data communications network, and just choosing AODV is more risky, than choosing network with cognition.

Thus, all the above results are found to be very satisfactory; however, once results are analysed there are new ideas developing on how to improve them. The first modification that can be done is to run the simulation for longer durations. Another improvement is that cognitivity proves to be useful for networks performing in difficult conditions (increased density, link breakages, etc.) which is exactly what makes the new procedures valuable. A number of further possible desired improvements are discussed in chapter 10.

Chapter 10 Future Plans

The outcomes of the introduced cognitivity applied to a MANET have been presented. Mainly tested were the introduced procedures on the specially created realistic scenarios. They were designed to highlight the significant changes that cognitivity bring in regards to the decisions made in order to improve the network performance.

The introduced procedures that are new and individually created by the author. Satisfactory improvements were shown with cognition applied when compared with network performance without cognitivity. However, many other things can be implemented and tested in the future. This section lists the ideas to be completed soon, together with discussing the methods already implemented and explained in this thesis.

10.9.1 Immediate Future Plans

The outcome of the results showed that a MANET with cognitive attributes does bring promising results. However, there have been some shortcomings in some of the algorithms that were implemented. For example, how the controllable node is used is simplistic and although it works some of the time, it does not really address the cause of the problem. The controllable node is instructed to move to a center point between the source node and destination node in order to re-establish or strengthen a link, however, consider the situation where nodes are moving in an area 300m x 300m and the transmission radius is 100m. If the nodes were at opposite boundaries of the landscape and the link breakage occurred at a node close to the source node then the current algorithm would achieve little results. It would be better to locate the link breakage and move the node to that vicinity.

In addition to this the cognition algorithm is very complicated; it performs many checks, one of which is checking the strike count (Section 8.5.1 Main Functionality). Although this works as intended, it has been observed that prior checks are undergone before the strike count check (i.e. calculating the dropped packet rate, send packet rate and received packet rates). If the strike count is greater than two, then the manager node is not allowed to make a request and the initial checks were a waste of time. It is proposed that checks are reordered so that they are only made if they are required. This should speed up the cognition process, conserve energy and conserve bandwidth.

Currently, the node objects can be configured to take many forms (i.e. soldiers, tanks, cars and so-forth), however, this requires configuring in the main simulation code. It would be better if this type of configuration is done as its own class and incorporated into the model. This would make coding each simulation an easier task. In addition to this, predefined models can be created and called upon when required.

The simulation world used in the models was quite simplistic, but in-line with most other research in this area. It would be desirable to incorporate real-life obstacles (i.e. buildings, trees and other everyday objects). It is unrealistic to assume a rectangle world with no obstructions or depth.

This section has discussed immediate plans that are already underway, in addition to this there are more ambitious plans that are discussed next.

10.9.2 Curvilinear Mobility

The mobility models are based on the piecewise constant motion. Since the simulator is a discrete one, provided time interval between changing the directions are small enough, it does not restrict much. The example of node's motion along an Archimedes' spiral is given in chapter 6. However, proper curvilinear motion equations could be introduced depending on time and with varying in time speed/acceleration. Predicting the position of the node will get more complicated, but if the prediction time is small enough, this difficulty should not cause much trouble. The journeys of the nodes would involve integration along the curves; other types of distances (non-Euclidean ones) thus more mathematical techniques would be applied.

10.9.3 Three-dimensional motion

A move to 3D motion is desirable, which is essential for MANETs involving airborne nodes. The latter is quite easy to implement in NS-3 and using vectors in C++, but some complications might appear on the way. It would require more powerful computers and more time to run simulations, together with some more advanced mathematical apparatus. Together with previous point mentioned, non-planar surfaces of the ground could be introduced.

10.9.4 Topology

The next idea is to use influence domains of the nodes and their intersection (say, Voronoi diagrams, Delaunay triangulations, etc.), the centre of the world, and the bounds of the world (say, convex hull, minimal bounding box, etc.) in order to trace the network topology structure. The network should know and predict splitting into two or more disconnected sub-networks working independently. In order to avoid the splitting the Network Command Centre would prioritize controlled nodes. At the moment a basic version of those procedures is used, and the way it worked makes future plans very promising.

10.9.5 Game Theory

More types of nodes representing cars, boats, motorbikes, helicopters etc. would be introduced. Each of these different types of nodes would have their own properties and missions. This is relatively easy to do based on the amount of different objects that were already used. The next step is to play the scenario involving two or more antagonistic data communication networks operating in the same area. This would increase the difficulty level significantly, as this military-type of scenario would require non-collaborative game theory.

This section meets Objective 9 (the formulation of a list of future objectives on cognitive MANET improvement), thus it concludes the thesis.

References

Al-Fuqaha, A.; Khan, B.; Rayes, A.; Guizani, M.; Awwad, O.; Ben Brahim, G.; , "Opportunistic Channel Selection Strategy for Better QoS in Cooperative Networks with Cognitive Radio Capabilities," Selected Areas in Communications, IEEE Journal on , vol.26, no.1, pp.156-167, Jan. 2008. doi: 10.1109/JSAC.2008.080114

Al-Soufy, K.A.M.; Abbas, A.M., "A path robustness-based quality of service routing for mobile ad hoc networks," Internet Multimedia Services Architecture and Application(IMSAA), 2010 IEEE 4th International Conference on , vol., no., pp.1,6, 15-17 Dec. 2010. doi: 10.1109/IMSAA.2010.5729417

Argyroudis, P.; Forde, T.; Doyle, L.; O'Mahony, D., "A Policy-Driven Trading Framework for Market-Based Spectrum Assignment," Policies for Distributed Systems and Networks, 2007. POLICY '07. Eighth IEEE International Workshop on , vol., no., pp.246,250, 13-15 June 2007 doi: 10.1109/POLICY.2007.2

Ash, D.; Ferguson,S.;, “The evolution of the telecommunications transport architecture: from megabit/s to terabit/s” Electronics & Communication Engineering Journal, February 2001.

Autorité de régulation. (2003). WIRELESS LAN, WIFI WLAN regulatory update. Available: http://www.arcep.fr/index.php?id=8571&L=1&tx_gsactualite_pi1%5Buid%5D=232&tx_gsactualite_pi1%5Bannee%5D=2003&tx_gsactualite_pi1%5Btheme%5D=0&tx_gsactualite_pi1%5Bmotscle%5D=&tx_gsactualite_pi1%5BbackI. Last accessed 12th Dec 2014.

Aziz, B.; Nourdine, E.; Mohamed, E.-K., "A Recent Survey on Key Management Schemes in MANET," Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008. 3rd International Conference on , vol., no., pp.1,6, 7-11 April 2008

doi: 10.1109/ICTTA.2008.4530182

Bakht, H.;, “The history of mobile ad hoc networks” School of Computing and Mathematical Sciences, Liverpool John Moores University (2005).

Basu, P.; Little, T.; , “Task-based self-organisation in large smart spaces:issues and challenges”. DARPA/NIST/NSF Workshop on Research: Issues in Smart Computing Environment, Atlanta, USA, 1999.

Bellman, R (1958).;, "On a routing problem". Quarterly of Applied Mathematics 16: 87–90. MR 0102435.

Beraldi R and Baldoni R (2003) A Caching Scheme for Routing in Mobile Ad Hoc Networks and Its Application to ZRP. IEEE TRANSACTIONS ON COMPUTERS, VOL. 52, NO. 8, AUGUST 2003

Bettstetter, C and Hartenstein, H and Pérez-Costa, X (2004). Stochastic properties of the random waypoint mobility model. ACM/Kluwer Wireless Networks: Special Issue on Modeling and Analysis of Mobile Networks, 10(5), September 2004.

Boston Dynamics. (2013). BigDog - The Most Advanced Rough-Terrain Robot on Earth. Available: http://www.bostondynamics.com/robot_bigdog.html. Last accessed November 2014.

Boutin, M.; Despins, C.; Denidni, T.; , "Interference Protection on Legacy Devices with Cognitive Radio using Smart Antennas," Vehicular Technology Conference Fall (VTC 2009-Fall), 2009 IEEE 70th , vol., no., pp.1-5, 20-23 Sept. 2009

Brayer, K.; , "Achieving timely message delivery quality-of-service in fixed and variable connectivity distributed routing networks," Selected Areas in Communications, IEEE Journal on , vol.22, no.7, pp. 1183- 1196, Sept. 2004. doi: 10.1109/JSAC.2004.829338

Blakeway, S. and Merabti, M. (2010) “Simulation Tools for use in Mobile Ad-hoc Networks” The 11th Annual Symposium on the Convergence of Telecommunications, Networking and Broadcasting

Chang, R; Chen, W; , "Mobility assessment on-demand (MAOD) routing protocol for mobile ad hoc networks," Global Telecommunications Conference, 2002. GLOBECOM '02. IEEE , vol.1, no., pp. 168- 172 vol.1, 17-21 Nov. 2002. doi: 10.1109/GLOCOM.2002.1188063

Chatterjee, S. Das, K. and Turgut, D. “WCA: A Weighted Clustering Algorithm for Ad hoc Networks,” Clustering Computing, vol. 5, pp. 193-204, 2002.

Cleveland, G; , “Packet Radio: Applications for Libraries in Developing Countries (1993)” Last Accessed [14/2/2011] Available at [http://archive.ifla.org/VI/5/reports/rep5/rep5.htm]

David B. Johnson and David A. Maltz. Protocols for Adaptive Wireless and Mobile Networking. IEEE Personal Communications, 3(1):34-42, February 1996.

Debnath, M.C.; Borah, P.; Mehedi, J.; Naskar, M.K.; , "A Distributed Algorithm for Self-Controled Mobile Ad-Hoc Network," Emerging Applications of Information Technology (EAIT), 2011 Second International Conference on , vol., no., pp.223-226, 19-20 Feb. 201

Duffy, R. B and Saull, J. W., 2008 “Managing Risk: The Human Element”, Wiley Publishers. ISBN: 978-0-470-71445-4

Electronic Communications Committee, (2002) “STRATEGIC PLANS FOR THE FUTURE USE OF THE FREQUENCY BANDS 862-870 MHz AND 2400-2483.5 MHz FOR SHORT RANGE DEVICES” European Conference of Postal and Telecommunications Administrations (CEPT). Helsinki, May 2002

Esa Hyytiä, Pasi Lassila, and Jorma Virtamo. Spatial Node Distribution of the Random Waypoint Mobility Model with Applications. to appear in IEEE Trans. Mobile Computing, 2006.

Esch, J.M., "Cognitive control," Proceedings of the IEEE , vol.100, no.12, pp.3154,3155, Dec. 2012. doi: 10.1109/JPROC.2012.2219193

Estrin, D.; Handley, M.; Heidemann, J.; McCanne, S.; Xu, Y.; Yu, H.;. 2000. Network Visualization with Nam, the VINT Network Animator. Computer 33, 11 (November 2000), 63-68. DOI=10.1109/2.881696 http://dx.doi.org/10.1109/2.881696

Feknous, Moufida; Houdoin, Thierry; Le Guyader, Bertrand; De Biasio, Joseph; Gravey, Annie; Gijon, Jose Alfonso Torrijos, "Internet traffic analysis: A case study from two major European operators," *Computers and Communication (ISCC), 2014 IEEE Symposium on* , vol., no., pp.1,7, 23-26 June 2014
doi: 10.1109/ISCC.2014.6912519

Flood, M. Blakeway S, (2010) “VANET: Vehicular Network Communications” The 11th Annual Symposium on the Convergence of Telecommunications, Networking and Broadcasting

Goldsmith A (2005) “Wireless Communications”. Cambridge University Press 2005. ISBN-10: 0521837162. pp2.

Grigorik I (2013) “High Performance Browser Networking”. O’Reilly Publishers. ISBN-10: 1449344763

Groth, David; Toby Skandier (2005). Network+ Study Guide, Fourth Edition'. Sybex, Inc. ISBN 0-7821-4406-3

Gundry, Stephen; Urrea, Elkin; Sahin, Cem Safak; Zou, Jianmin; Uyar, M. Umit; , "Formal convergence analysis for bio-inspired topology control in MANETs," Sarnoff Symposium, 2011 34th IEEE , vol., no., pp.1-5, 3-4 May 2011

Gupta, A.; Sadawarti, H.; Verma, A., “Performance analysis of AODV, DSR & TORA Routing Protocols” IACSIT International Journal of Engineering and Technology, Vol.2, No.2, April 2010. ISSN: 1793-8236

Hyunwoo Joe; Jonghyuk Lee; Duk-Kyun Woo; Pyeongsoo Mah; Hyungshin Kim, "Demo abstract: A high-fidelity sensor network simulator using accurate CC2420 model," Information Processing in Sensor Networks, 2009. IPSN 2009. International Conference on , vol., no., pp.429,430, 13-16 April 2009

Hyytia, E.; Lassila, P.; Virtamo, J., "Spatial node distribution of the random waypoint mobility model with applications," Mobile Computing, IEEE Transactions on , vol.5, no.6, pp.680,694, June 2006. doi: 10.1109/TMC.2006.86

IEEE. (2007). IEEE Standard for Information technology— Telecommunications and information exchange between systems— Local and metropolitan area networks— Specific requirements. Part 11: Wireless LAN Medium Access. Available: http://www.ie.itcr.ac.cr/marin/lic/el4515/antenas/802.11-2007.pdf. Last accessed 12th Dec 2014.

Istikmal; Leanna, V.Y.; Rahmat, B., "Comparison of proactive and reactive routing protocol in mobile adhoc network based on “Ant-algorithm”," Computer, Control, Informatics and Its Applications (IC3INA), 2013 International Conference on , vol., no., pp.153,158, 19-21 Nov. 2013. doi: 10.1109/IC3INA.2013.6819165

Jahani, S.; Bagherpour, M.; , "A clustering algorithm for mobile ad hoc networks based on spatial auto-correlation," Computer Networks and Distributed Systems (CNDS), 2011 International Symposium on , vol., no., pp.136-141, 23-24 Feb. 2011 doi: 10.1109/CNDS.2011.5764560

Jian Li; Mohapatra, P.; , "A novel mechanism for flooding based route discovery in ad hoc networks," Global Telecommunications Conference, 2003. GLOBECOM '03. IEEE , vol.2, no., pp. 692- 696 Vol.2, 1-5 Dec. 2003. doi: 10.1109/GLOCOM.2003.1258327

Jin-Hee Cho; Swami, A.; Ing-Ray Chen; , "Modeling and Analysis of Trust Management for Cognitive Mission-Driven Group Communication Systems in Mobile Ad Hoc Networks," Computational Science and Engineering, 2009. CSE '09. International Conference on , vol.2, no., pp.641-650, 29-31 Aug. 2009doi: 10.1109/CSE.2009.68

Karunakaran S. and Thangaraj P., “An Adaptive Weighted Cluster Based Routing (AWCBRP) Protocol for Mobile Ad Hoc Networks,” WSEAS TRANSACTIONS on COMMUNICATIONS, pp. 248-257, 2008.

Karygiannis, A.; Antonakakis, E.; , "mLab: An ad hoc network test bed," Consumer Communications and Networking Conference, 2006. CCNC 2006. 3rd IEEE , vol.2, no., pp. 1312- 1313, 8-10 Jan. 2006. doi: 10.1109/CCNC.2006.1593264

Kennedy, Robert A.; , "Psiactive Networking for Military Applications - A Predictive Cross-Layer Approach," Military Communications Conference, 2007. MILCOM 2007. IEEE , vol., no., pp.1-7, 29-31 Oct. 2007. doi: 10.1109/MILCOM.2007.4455188

Kiwior D and Lam L (2007) Routing Protocol Performance Over Intermittent Links. 1-4244-1513-06/07©2007 IEEE

Kumar, Aditya; Srivastava, Shiv Shakti; Ram, Babu; Singh, Pardeep; , "Exploring a new dimension in MANETs through a new routing protocol," Electronics Computer Technology (ICECT), 2011 3rd International Conference on , vol.5, no., pp.328-331, 8-10 April 2011

Kunzelman, R. (1978);, “Progress report on packet radio experimental network.” Quarerly Technical Report 14. SRI Project Number 8933. November 1978

Kushwaha, H.; Chandramouli, R., "Secondary Spectrum Access with LT Codes for Delay-Constrained Applications," Consumer Communications and Networking Conference, 2007. CCNC 2007. 4th IEEE , vol., no., pp.1017,1021, Jan. 2007. doi: 10.1109/CCNC.2007.205

Lee, D. (2014) “SanDisk SD memory card 'largest ever'”, BBC News Technology. Available at: http://www.bbc.co.uk/news/technology-29175093. Last Accessed: September 2014.

Leiner, B.M.; Ruther, R.J.; Sastry, A.R., "Goals and challenges of the DARPA GloMo program [global mobile information systems]," Personal Communications, IEEE , vol.3, no.6, pp.34,43, Dec 1996. doi: 10.1109/98.556477

Linkins, J., “Abundance of Web Video Crippling the Internet”. Huffington Post, 2008.

London Organising Committe of the Olympic Games and Paralymipic Games Limited;, “Preparing your Business for the Games” Planning Information for businesses, pp.12. April 2012.

Lusheng Ji; Corson, M.S., "A lightweight adaptive multicast algorithm," Global Telecommunications Conference, 1998. GLOBECOM 1998. The Bridge to Global Integration. IEEE , vol.2, no., pp.1036,1042 vol.2, 1998 doi: 10.1109/GLOCOM.1998.776885

Mackay, D.J.C., Information Theory, Inference, and Learning Algorithms. Cambridge University Press, Cambridge, UK., 2003.

Mahmoud, E.Q., Cognitive Networks: Towards Self-Aware Networks. 2007.

Meyer, D., “The Cloud expands its hotspot coverage”. ZDNet. Available: http://www.zdnet.com/article/the-cloud-expands-its-hotspot-coverage/ Accessed:26 May 2015.

Miniwatts Marketing Group, INTERNET USAGE STATISTICS The Internet Big Picture World Internet Users and Population Stats. 2009.

Myoupo, J.-F.; Naimi, M.; Thiare, O.; , "A clustering Group Mutual Exclusion algorithm for mobile ad hoc networks," Computers and Communications, 2009. ISCC 2009. IEEE Symposium on , vol., no., pp.693-696, 5-8 July 2009 doi: 10.1109/ISCC.2009.5202340

OfCom UK. (2014). UK Frequency Allocation Table (UKFAT). Available: http://www.ofcom.org.uk/static/spectrum/fat.html. Last accessed 12th Dec 2014.

Ousmane, T.; Mohamed, N,; Mourad, G.:, (2007) “A Group Mutual Exclusion Algorithm for Mobile Ad Hoc Networks”, Innovations and Advanced Techniques in Computer and Information Sciences and Engineering

Pan, M.; Sheng-Yan Chuang; Sheng-De Wang, "Local repair mechanisms for on-demand routing in mobile ad hoc networks," Dependable Computing, 2005. Proceedings. 11th Pacific Rim International Symposium on , vol., no., pp.8 pp.,, 12-14 Dec. 2005. doi: 10.1109/PRDC.2005.38

Papavassiliou, S. Xu, P. Orlik, M. Snyder, and P. Sass, "Scalability in global mobile information systems (GloMo): issues, evaluation methodology and experiences," Wirel. Netw., vol. 8, pp. 637-648, 2002Penman, P (2006) Give Me Wi-Fi—or Give Me Death. [Available online at http://homepage.mac.com/jdalisay/blog/PenmanMarch06.html] [Accessed 4/4/2006]

Park, V and Corson, S (2001) "Temporally-Ordered Routing Algorithm (TORA) Version 1 Functional Specification". IETF MANET Working Group.

Patidar, P.; Vijaykumar, C., "A model based channel shortening technique for IEEE 802.11a OFDM system," TENCON 2007 - 2007 IEEE Region 10 Conference , vol., no., pp.1,4, Oct. 30 2007-Nov. 2 2007. doi: 10.1109/TENCON.2007.4429029

Perkins, C (1997). Ad Hoc On Demand Distance Vector (AODV) Routing. Mobile Ad Hoc Networking Working Group, Internet Draft, 20 November 1997.

Perkins, C and Belding-Royer, E (2003). Ad Hoc On Demand Distance Vector (AODV) Routing. Mobile Ad Hoc Networking Working Group, Internet Draft, 17 February 2003.

Physicians for Social Responsibility (2006) "Preventing Nuclear Terrorism: Understanding the Health Consequences of a Nuclear Bomb Explosion" Press Release.

Pullin, A.J., S. Presland, and C. Pattinson (2008). Using Ship Movement in the Irish Sea for MANET Evaluation. in Computer Modeling and Simulation, 2008. EMS '08.Second UKSIM European Symposium on. 2008

Rachedi, A.; Benslimane, A.; Otrok, H.; Mohammed, N.; Debbabi, M., "A Mechanism Design-Based Secure Architecture for Mobile Ad Hoc Networks," Networking and Communications, 2008. WIMOB '08. IEEE International Conference on Wireless and Mobile Computing, , vol., no., pp.417,422, 12-14 Oct. 2008 doi: 10.1109/WiMob.2008.77

Rahman, K.A.; Tepe, K.E.; , "Mobility Assisted Routing in Mobile Ad Hoc Networks," New Technologies, Mobility and Security (NTMS), 2011 4th IFIP International Conference on , vol., no., pp.1-5, 7-10 Feb. 2011. doi: 10.1109/NTMS.2011.5720618

Rahman, J.; Hasan, M.A.M.; Islam, M.K.B., "Comparative analysis the performance of AODV, DSDV and DSR routing protocols in wireless sensor network," Electrical & Computer Engineering (ICECE), 2012 7th International Conference on , vol., no., pp.283,286, 20-22 Dec. 2012. doi: 10.1109/ICECE.2012.6471541

Ramesh, V.; Subbaiah, P.; Koteswar Rao, N.; Janardhana Raju, M., "Performance Comparison and Analysis of DSDV and AODV for MANET," International Journal on Computer Science and Engineering Vol. 02, No. 02, 2010

Reber, P., 2010, "What is the Memory Capacity of the Human Brain?", Scientific America. Professor of Psychology, Northwestern Univeristy.

Rohde,; Schwarz,;, (2012) "802.11ac Technology Introduction White Paper" White Paper. [Available at http://www2.rohde-schwarz.com/en/service_and_support/ Downloads/Application_Notes/?downid=7155][Last accessed July 2012] Published March 2012

Rosset, S.; Rho, S.; Redmond, T.; , "ActiveEdge-M: a survivable, multi-agent middleware platform for mobile ad-hoc networks," Military Communications Conference, 2005. MILCOM 2005.IEEE , vol., no., pp.1595-1601 Vol. 3, 17-20 Oct. 2005doi: 10.1109/MILCOM.2005.1605903

Santi, P and Blough, D and Bostelmann, H (2006) IEEE Transactions on Mobile Computing pp. 943-944

Santivanez, Cesar, "Transport Capacity of Opportunistic Spectrum Access (OSA) MANETs," Cognitive Radio Oriented Wireless Networks and Communications, 2007. CrownCom 2007. 2nd International Conference on , vol., no., pp.9,18, 1-3 Aug. 2007. doi: 10.1109/CROWNCOM.2007.4549765

Shih-Hsien Wang; Ming-Chieh Chan; Ting-Chao Hou, "Zone-based controlled flooding in mobile ad hoc networks," Wireless Networks, Communications and Mobile Computing, 2005 International Conference on , vol.1, no., pp.421,426 vol.1, 13-16 June 2005. doi: 10.1109/WIRLES.2005.1549446

Song Chong; Nagarajan, R.; Yung-Terng Wang, "Flow control in a high-speed bus-based ATM switching hub," *Broadband Switching Systems Proceedings, 1997. IEEE BSS '97., 1997 2nd IEEE International Workshop on* , vol., no., pp.137,147, 2-4 Dec 1997. doi: 10.1109/BSS.1997.658918

Sørensen, L,; Skouby E.; , "OUTLOOK Visions and research directions for the Wireless World" Wireless World Research Forum, July 2009, Volume 4. pp.4

Stallings, W. , "Data and Computer Communications" Eighth Edition, Pearson Education International Edition, ISBN 0-13-507139-9. pp.13

The_American_Heritage®_Dictionary_of_the_English_Language, Cognitive Definition. 2009

Thomas, R.W.; DaSilva, L.A.; MacKenzie, A.B.; , "Cognitive networks," New Frontiers in Dynamic Spectrum Access Networks, 2005. DySPAN 2005. 2005 First IEEE International Symposium on , vol., no., pp.352-360, 8-11 Nov. 2005

Vaman, D.R.; Lijun Qian; , "Cognitive radio mixed sensor and Mobile Ad Hoc Networks (SMANET) for dual use applications," Systems, Man and Cybernetics, 2008. SMC 2008. IEEE International Conference on , vol., no., pp.3304-3310, 12-15 Oct. 2008. doi: 10.1109/ICSMC.2008.4811806

Vyas, K.; Chaturvedi, A., "Comparative analysis of routing protocols in MANETS," Signal Propagation and Computer Technology (ICSPCT), 2014 International Conference on , vol., no., pp.692,697, 12-13 July 2014. doi: 10.1109/ICSPCT.2014.6884994

Wang Bao-Jun; Zhan Ying, "A survey and performance evaluation on sliding window for data stream," Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on , vol., no., pp.654,657, 27-29 May 2011. doi: 10.1109/ICCSN.2011.6014977

Wang L and Olariu S (2004) A Two-Zone Hybrid Routing Protocol for Mobile Ad Hoc Networks. IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 15, NO. 12, DECEMBER 2004

Warren R (2015) Lake District Search & Mountain Rescue Association Mountain Accidents 2015, ISSN 2046-6277

Wu, Kui; Harms, Janelle, "Multipath routing for mobile ad hoc networks," Communications and Networks, Journal of , vol.4, no.1, pp.48,58, March 2002. doi: 10.1109/JCN.2002.6596933

Xiuhua Fu; Wenan Zhou; JunliXu; Junde Song; , "Extended Mobility Management Challenges over Cellular Networks combined with Cognitive Radio by using Multi-hop Network," Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on , vol.2, no., pp.683-688, July 30 2007-Aug. 1 2007

Yang F; Sun B; , "Ad hoc on-demand distance vector multipath routing protocol with path selection entropy," Consumer Electronics, Communications and Networks (CECNet), 2011 International Conference on , vol., no., pp.4715-4718, 16-18 April 2011

Younis, O.; Kant, L.; Chang, K.; Young, K.; Graff, C.; , "Cognitive MANET design for mission-critical networks," Communications Magazine, IEEE , vol.47, no.10, pp.64-71, October 2009. doi: 10.1109/MCOM.2009.5273810

Zhai, h.; Fang, Y., "Performance of wireless LANs based on IEEE 802.11 MAC protocols," *Personal, Indoor and Mobile Radio Communications, 2003. PIMRC 2003. 14th IEEE*

Proceedings on , vol.3, no., pp.2586,2590 vol.3, 7-10 Sept. 2003 doi: 10.1109/PIMRC.2003.1259194

Zhen Li; Qi Liao; Striegel, A., "Toward a Socially Optimal Wireless Spectrum Management," Networking Technologies for Software Defined Radio (SDR) Networks, 2010 Fifth IEEE Workshop on , vol., no., pp.1,6, 21-21 June 2010. doi: 10.1109/SDR.2010.5507923

Zvonar, Z.; Mitola, J., "SDR and wireless infrastructure," Communications Magazine, IEEE , vol.41, no.1, pp.104,104, Jan. 2003. doi: 10.1109/MCOM.2003.1166665

Bibliography

Deborah Estrin, Mark Handley, John Heidemann, Steven McCanne, Ya Xu, and Haobo Yu. 2000. Network Visualization with Nam, the VINT Network Animator. Computer 33, 11 (November 2000), 63-68. DOI=10.1109/2.881696 http://dx.doi.org/10.1109/2.881696

Zhai, h.; Fang, Y., "Performance of wireless LANs based on IEEE 802.11 MAC protocols," *Personal, Indoor and Mobile Radio Communications, 2003. PIMRC 2003. 14th IEEE Proceedings on* , vol.3, no., pp.2586,2590 vol.3, 7-10 Sept. 2003 doi: 10.1109/PIMRC.2003.1259194

Almasri, Marwah M; Elleithy, Khaled M, "Data fusion models in WSNs: Comparison and analysis," American Society for Engineering Education (ASEE Zone 1), 2014 Zone 1 Conference of the , vol., no., pp.1,6, 3-5 April 2014

Bajaj,S; Estrin,D; Fall,K; Floyd,S; Haldar,M; Handley,P; Helmy,A; Heidemann,J; Huang,P; Kumar,S; McCanne,S; Rejajie,R; Punnet,S; Varadhan,S; Ya,X; Yu,H; Zappala,D;, "Improving Simulation for Network Research" University of Southern California, 1999(99-702b).

Ben-El-Kezadri, R; Kamoun, F;, "Graphic Visualization of the 802.11 DCF Protocol under the NS-2 Simulator" Simulation

Symposium, 2007. ANSS '07. 40th Annual. 2007.

Chengyu, Z; et al., "A comparison of active queue management algorithms using the OPNET Modeler" Communications Magazine, IEEE, 2002. 40(6): p. 158-167.

Desbrandes, S; Dunand, L; "Opnet 2.4: an environment for communication network modeling and simulation. Proc. of the European Simulation Symposium (ESS'93)" 1993: p. 609-614.

Dow, C, et al. "A study of recent research trends and experimental guidelines in mobile ad-hoc network. in Advanced Information

Networking and Applications" 2005. AINA 2005. 19th International Conference on. 2005.

IETF Website (2011) “Overview of the Internet Engineering Task Force”. Available at: [http://www.ietf.org/overview.html] Last Accessed [July 2011]

Kurkowski, S; Colagrossa, T;, "MANET Simulation Studies: The Current State and New Simulation Tools" 2004.

Kaufman, C; "Running Ns and Nam Under Windows 9x/2000/XP Using Cygwin" 2008(01/10/08)

Paxson,V; "Why we don't know how to simulate the Internet" Winter Simulation Conference, 1997: p. 1037-1044.

Rahman, K.A.; Tepe, K.E.; , "Mobility Assisted Routing in Mobile Ad Hoc Networks," New Technologies, Mobility and Security (NTMS), 2011 4th IFIP International Conference on , vol., no., pp.1-5, 7-10 Feb. 2011. doi: 10.1109/NTMS.2011.5720618

Zhai, H,; Kwon, Y,; Fang, Y.; , “Performance analysis of IEEE 802.11 MAC protocols in wireless LANs” WIRELESS COMMUNICATIONS AND MOBILE COMPUTING 2004; vol 4., pp.917–931. DOI: 10.1002/wcm.263

K. Fall and K. Varadhan, Eds. ns notes and documentation. The VINT Project, UC Berkeley, LBL, USC/ISI, and Xerox PARC, 1997.

Wibble.co.uk (2011) The Radio Spectrum – UK Allocations [Available at: http://www.wibble.co.uk/links/ukspectrum/spectrum.html] [accessed: 01/02/2011]

Chin, K (2005) The Behavior of MANET routing protocols in realistic environments, Asia-Pacific Conference on Communications, 3-5 October 2005, 906-910.

Blakeway, S and Pullin A (2007). The Effect of Node Density on Routing Protocol Performance in Mobile Ad-Hoc Networks. Proceedings of the 8th Symposium Conference on the Convergence of Telecommunications, Networking and Broadcasting pp 327-332. June 2007.

Johansson, P and Larsson, T and Hedman, N and Mielczarek, B and Degermark, M (1999) “Scenario-Based Performance Analysis of Routing Protocols for Mobile Ad Hoc Networks,” Proc. IEEE/ACM MOBICOM '99 Conf., pp. 195-206, 1999.

Kurkowski, S and Camp, T and Mushell, N and Colagrosso, M (2005) A visualization and Analysis Tool for NS-2 Wireless Simulations: iNSect. NSF Research Group

Johansson, P., Larsson, T., Hedman, N., Mielczarek, B., and Degermark, M. 1999. Scenario-based performance analysis of routing protocols for mobile ad-hoc networks. In Proceedings of the 5th Annual ACM/IEEE international Conference on Mobile Computing and Networking

(Seattle, Washington, United States, August 15 - 19, 1999). MobiCom '99. ACM, New York, NY, 195-206. DOI= http://doi.acm.org/10.1145/313451.313535

Aggarwal A and Bharadwaj (2003) ACC – ABCD Compliant NS-2. University of Windsor. School of Computer Science. Technical Report # 04-020

Al-Raweshidy, et al, (2004) "Vertical Handover between HiperLAN/2 and UMTS Cellular System"

BBC (2006) City-wide wi-fi rolls out in UK British Broadcasting Corportation, 3 January 2006, 14:52 GMT

Bradley T and Waring B (2005) Complete Guide to WiFi Security - ways to safeguard your data at a public wireless hotspot is to use a virtual private network, or VPN.

Cavilla, A (2005) MANET extension to ns2. Department of Computer Science. University of Toronto.

Cheng, C and Riley, R and Kumar, S and Garcia-Lunes-Aceves, J (1989). A Loop-Free Extended Bellman-Ford Routing Protocol Without Bouncing Effect. In Proceedings of ACM Sigcomm, pages 224--236, August 1989.

Cisco Systems (2002) Understanding TCP/IP Cisco Systems 1992-2002, Inc. [Available Online]
http://www.cisco.com/univercd/cc/td/doc/product/iaabu/centri4/user/scf4ap1.htm
[Accessed 2/5/2006]

Fairhurst, G. (2001) Communications Engineering

Fall, K and Varadhan, K (2003,) The ns-2 Manual - 16.1.1 Mobilenode: creating wireless. Available online at [http://www.isi.edu/nsnam/ns/ns-documentation.html][Accessed November 11, 2005]

Harte L (2004) Introduction to Bluetooth ALTHOS Publishing 2004

IEEE (2006) AIEE: Wire Communications, Light and Power [Available Online]
http://www.ieee.org/web/aboutus/history/index.html [Accessed 2/2/2006]

Impett, M. and Corson, M.S. and Park, V (2000). Wireless Communications and Networking Conference, 2000. WCNC. 2000 IEEE. Volume: 1, On page(s): 117-122 vol.1

Ke, Q and Maltz, D and Johnson B (2000). Emulation of Multi-Hop Wireless Ad Hoc Networks. In Proceedings of the Seventh International Workshop on Mobile Multimedia Communications (MOMUC 2000), IEEE Communications Society, Tokyo, Japan, October 2000.

Khalil, M. A (2005) Vision to Reality: Applications of Wireless Laptops in Accessing Information from Digital Libraries: End-Users' Viewpoints Volume 21, Issue 7, Pages 25 - 29

Kurkowski, S and Camp, T and Mushell, N and Colagrosso, M (2005) A visualization and Analysis Tool for NS-2 Wireless Simulations: iNSect. NSF Research Group

Lyman, J. (2004) Experts Warn of Critical TCP Flaw Technology News. April 2004. Copyright 1998-2006 ECT News Network

Meredith, T and Ehrlich J and Rieck M (2005) AD-HOC NETWORK ROUTING BASED ON AODV AND K-SPR SETS Department of Mathematics and Computer Sciences, Drake University

Meyer, E. E. (1970) ARPA Network Protocol Note, Network Working Group, Massachusetts Institute of Technology

Monarch Project (2000) Wireless and Mobility Extensions to ns-2 Available online at [http://www.monarch.cs.rice.edu/cmu-ns.html][Accessed December 3, 2005]

National Institute of Standards and Technology (2005), http://www.nist.gov/dads/HTML/bellmanford.html [Accessed 12/12/2005]

Neuman, B and Theodor T (2004) Kerberos: An Authentication Service for Computer Networks Copyright 1994 Institute of Electrical and Electronics Engineers.

Volume 32, Number 9, pages 33-38, September 1994.

NewsFactor Network (2006) Slow Standards Hurt Fast 802.11n Wireless. 2006 NewsFactor Network. August 2, 2006. Available online at [http://www.newsfactor.com/news/Slow-Standards-Hurt-Fast-Wireles/story.xhtml?story_id=0330038TEZZF] [Accessed August 10, 2006]

Perking, C.E. and Bhagwat, P (1994) Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers SIGCOMM 1994.

Perkins, C (2001). Ad hoc networking. Addison-Wesley, 2001. ISBN 0-201-30976-9.

Perkins, C and Bhagwat, P (1994) Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers.

Perkins, C and Royer, E (1998) Ad Hoc On Demand Distance Vector (AODV) Routing. Mobile Ad Hoc Networking Working Group Internet Draft: draft-ietf-manet-aodv-01.txt

Rushton-Jones, L (2005) Wifi - It's More Than A Swimming Pool [online: http://www.iyonder.com/news.asp] [Accessed 4/1/06]

Tarkowski, A (2003) Heterogeneous Engineers of the Internet. How Inventors Weaved Together Internet Technologies, Social Values and Blueprints for New Social Institutions. Central European University, Warsaw, Poland

Tian Y, Xu K, Ansari N (2005) TCP in Wireless Environments: Problems and Solutions. P529. IEEE Radio Communication March 2005

Ting-Yu L et al (2004) An Efficient Link Polling Policy by Pattern Matching for Bluetooth Piconets The Computer Journal. The British Computer Society

Wall Street (2003) WiMAX The next step in Internet Access. The Wall Street Journal. Printed 9th April 2003

Washington Post (2003) Broadband Wireless Access. The Daily Washington Post. Printed 9th April 2003

Appendix 1 - Code

```
// We are using advanced dynamic data structures that are only supported in the latest version
// of C++.  If the program does not compile and complains about needing to use C11++ then
// run the following command in the terminal:  CXXFLAGS="-std=c++0x" ./waf configure

/*   BEST WAY TO RUN THE SIMULATION
./waf --run "final_2
    --numOfNodes=10
    --numOfTrafficFlows=1
    --numOfCNodes=2
    --dataRate=64kb/s
    --cognition=true
    --simulationTime=100
    --radius=100
    --logFile=T1
    --mobilityModel=L
    --initialProtocol=aodv
    --cNodeSpeed=8.5
    --nodeSpeed=8.5
    --recordPositionInterval=5
    --folderName=F1
" > output.txt 2>&1
*/

#include "ns3/aodv-module.h"
#include "ns3/dsdv-module.h"
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/mobility-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/wifi-module.h"
#include "ns3/v4ping-helper.h"
#include <iostream>
#include <math.h>
#include <cmath>
#include "ns3/udp-echo-helper.h"  // for the applications
#include "ns3/netanim-module.h"     // so we can trace the mobility
#include "ns3/applications-module.h"
#include "ns3/aodv-rqueue.h"
#include "ns3/trace-helper.h"
#include "ns3/flow-monitor.h"
#include "ns3/flow-monitor-helper.h"
#include "ns3/flow-monitor-module.h"
#include "ns3/energy-module.h"  // so we can use battery
#include <string>
#include <vector>
#include <limits>
#include <sys/ioctl.h>    // get window width size
#include <stdio.h>        // get window width size
#include <sys/types.h>    // so we can create folders
#include <sys/stat.h>
#include <unistd.h>

using namespace std;
using namespace ns3;

// GLOBAL VARIABLES
const double PI = 3.14159265359;
ApplicationContainer sinkApps;
const int sourceNodesNumber = 4; // change and move
int myPacketCount = 0;
FlowMonitorHelper flowmon;  //required

// set up some logging and give a name.
NS_LOG_COMPONENT_DEFINE ("simpleWifi2");

  //  Set up which components are to be logged
  //  LogComponentEnableAll(LOG_INFO);
  //  LogComponentEnableAll("UdpEchoServerApplication", LOG_LEVEL_INFO);
  //  Each component has allows for different levels of logging.
  //  This level is at the INFO level resulting in messages when packets are sent and
received.

  // Some initilisation
  // Time::SetResolution (Time::NS);
  // From time.cc
  // Time is by default set to NS (nanoseconds)
  // Also calles the NS_LOG_FUNCTION
  // Other values are (Y,D,H,MIN,S,MS,US,NS,PS and FS)

class CogMan
{
private:
  int numOfNodes, dNode;
  string currentProtocol;
  double simulationTime;
  int txPower;
  string logFile;
  int bytesTotal, packetsReceived, port, worldSize;
  uint32_t m_nodeSpeed;
  double myRadius;
  double EnergyDetectionThreshold;
  double CcaMode1Threshold;
  double TxGain;
  double RxGain;
  double TxPowerLevels;
  double TxPowerEnd;
  double TxPowerStart;
  double RxNoiseFigure;
  double initialEnergy;

  bool switchProtocolbool;
  bool switchChannelbool;
  bool controlledNodesbool;

  double densityReference;
  double timeToStabilize;
  double checkingIntervalTime;

  int totalAodvPacketsDropped;
  int totalDsdvPacketsDropped;
  int totalAodvPacketsReceived;
  int totalDsdvPacketsReceived;
  int totalAodvPacketsSent;
  int totalDsdvPacketsSent;
  double totalSwitchingReceived;
  double totalSwitchingSent;
  double totalSwitchingDropped;
  double maxReceivingRate;

  bool justSwitched;
  double gradient;
  double receivedPacketsRatePrevious;

  // An'ka's calculations

  double ApreviousReceived;
  double DpreviousReceived;
  double SpreviousReceived;
  int counterGoodJob;
  int counterGoodDsdv;
  int counterGoodAodv;

  int totalswitchingA;
  int totalPreviousSwitchingA;
  int totalswitchingD;
  int totalPreviousSwitchingD;

  bool mInRange;
  double previousDistance;
  bool aodvBool;
  bool dsdvBool;

  double networkPerformance;
  int managementCallsCounter;
  int goodIntervalsNumber;

  uint16_t packetIdentification=0;

  string fn;
```

```
// packets that are sent
string dataRate               = "64kb/s";
int    sendPacketTotal;
int    sendDsdvDataPacket;
int    sendAodvDataPacket;
int    sendDsdvControlPacket;
int    sendAodvControlPacket;
double expectedEstimatedSum;

// packets that are dropped
int dropPacketTotal       = 0; //  T
int dropAodvTotal         = 0; //  O
int dropDsdvTotal         = 0; //  T
int dropDsdvDataPacket    = 0; //  A
int dropAodvDataPacket    = 0; //  L
int dropDsdvControlPacket = 0; //  S
int dropAodvControlPacket = 0; //
int dropCurrentProtocol   = 0; //

int dropAodvCTS           = 0; //  C
int dropDsdvCTS           = 0; //  O
int dropAodvRR            = 0; //  N
int dropDsdvRR            = 0; //  T
int dropAodvRQ            = 0; //  R
int dropDsdvRQ            = 0; //  O
int dropAodvMAC_RR        = 0; //  L
int dropDsdvMAC_RR        = 0; //
int dropAodvMAC_RQ        = 0; //
int dropDsdvMAC_RQ        = 0; //

int dropAodvFF            = 0; //  P
int dropDsdvFF            = 0; //  A
int dropAodvUR            = 0; //  Y
int dropDsdvUR            = 0; //  L
int dropAodvNFM           = 0; //  O
int dropDsdvNFM           = 0; //  A
int dropAodvOther         = 0; //  D
int dropDsdvOther         = 0; //

// packets that are recieved
int receivedPacketTotal       = 0;
int receivedDsdvDataPacket    = 0;
int receivedAodvDataPacket    = 0;
int receivedDsdvControlPacket = 0;
int receivedAodvControlPacket = 0;
int sourceNodeBeingFollowed = -1;

// management
bool   cognition                   = true;
double cNodeSpeed                  = 8.5;
double nodeSpeed                   = 8.5;
bool   goodRelationshipState       = true;
bool   allowedToSwitchChannel      = true;
bool   allowedToSwitchProcol       = true;
bool   allowedToMoveControlledNode = true;
bool   NCCScheduled                = false;
int    numOfControlledNodes        = 2;
char   mobilityModel               = 'L';
double recordPositionInterval      = 5.0;
int    numOfTrafficFlows           = 1;
string initialProtocol             = "aodv";
string folderName                  = "default";
//char * fn                          = "default";


// virgin reports
bool reportedSend = false;
bool reportedReceived = false;
bool reportMovingCN = false;
bool reportedStabalise = true;
bool reportedIncrease = false;
bool reportedDrop = false;


double stabilizingTime = 30;
```

```
  double nccCheckTime = 2;

  // Structures
  struct nodeDistance
  {
    int sourceNode;
    int neighbourNode;
    double distance;
    double myTime;
  };

  std::vector<nodeDistance> myNeighbours;        //required to store nodes in range from
source
  std::vector<nodeDistance> nodesInMyWorld;      //required to store ALL node distances a
source
  std::vector<nodeDistance> allNodesTheWorld;    //required to store ALL node distances
multiple sources

  struct flowInformation
  {
    unsigned int flowId;
    ns3::Ipv4Address sourceAddress;
    ns3::Ipv4Address destinationAddress;
    uint32_t txPackets;
    uint64_t txBytes;
    double rxOffered;
    uint64_t rxPackets;
    uint64_t rxBytes;
    double throughput;
  };

  std::vector<flowInformation> myDataFlows;


  struct currentPosition
  {
    int nodeId;
    double time;
    double x;
    double y;
  };

  vector<currentPosition> myPositions;          //required to store history of positions

  struct managerNode
  {
    int sourceNodeId;
    int managerNodeId;
    double timeElected;
    double timeInRangeUntil;
    double timePowerUntil;
    double density;
  };

  std::vector<managerNode> managersList;        //required to store History of managers
  std::vector<managerNode> candidatesList;
  std::vector<managerNode> candidatesListPercents;

struct IpAddresses
{
  Ipv4Address source;
  Ipv4Address destination;
  double      time;
};

  std::vector<IpAddresses> packetReceived;
  std::vector<IpAddresses> packetDropped;
  std::vector<IpAddresses> packetSent;

  std::vector<IpAddresses> packetReceivedByCurrentProtocol;
  std::vector<IpAddresses> packetSentByCurrentProtocol;
  std::vector<IpAddresses> packetDroppedByCurrentProtocol;

struct AllPackets
{
  Ipv4Address source;
  Ipv4Address destination;
  double      time;
  char        action;
};

  std::vector<AllPackets> allPackets;

struct controlledNode
{
  int nodeId;
};

  std::vector<controlledNode> AodvEaglesNode;
  std::vector<controlledNode> AodvSharksNode;
  std::vector<controlledNode> DsdvEaglesNode;
  std::vector<controlledNode> DsdvSharksNode;

struct nodeMoving
{
  int nodeId;
  double timeStart;
  double timeArrive;
};

  std::vector<nodeMoving> setNodeMove;
  std::vector<nodeMoving> setControlledNodeMove;

struct memorySystemState
{
  int sNode;
  int mNode;
  double time;
  double density;
  string protocol;
  int droppedPacketsAodv;
  int droppedPacketsDsdv;
  int receivedPacketsAodv;
  int receivedPacketsDsdv;
  int channel;
  int receivedPacketsSwitching;
};

  std::vector<memorySystemState> systemState;

struct cognitivity
{
  double atTime;
  int    sNode;
  int    mNode;
  double dropRate;
  double receiveRate;
  double sendRate;
  double density;
  int    channel;
  string routingProtocol;
  char   request;
  char   requestProgress;
  double backOffTime;
  int    performance;
};

  std::vector<cognitivity> managementNodeMemory;

struct request
{
  double atTime;
  int    mNode;
  char   request;
  double predictedReceivingRate;
};

  std::vector<request> newRequests;
  std::vector<request> requests;

struct traffic
{
  int sNode;
  int dNode;
  double timeStart;
  double timeStop;
};

  std::vector<traffic> myTrafficFlows;
  std::vector<traffic> myTrafficCongestionFlows;


struct relationshipState  //-----------------------------------//
{                         //  A vector called relationshipState //
  int sNode;              //  to store the relationshipState(s) //
  int mNode;              //                                   //
  double time;            //  The structure has variables      //
  double range;           //    sNode            int           //
  double energyLevel;     //    mNode            int           //
};                        //    time             double        //
                          //    range            double        //
                          //    energyLevel      double        //
                          //-----------------------------------//
  std::vector<relationshipState> relationshipStates;

struct packetRate
{
  double time;
  int source;
  double sent;
  double received;
  double dropped;
};

  std::vector<packetRate> packetRates;

public:
  CogMan();

  void NodeStuff(uint32_t n);

  // packets
  string GetDestinationMACAddress(ns3::Ptr<const ns3::Packet>);
  string GetSourceIPAddress(ns3::Ptr<const ns3::Packet>);
  string GetDestinationIPAddress(ns3::Ptr<const ns3::Packet>);
  void UpdateSentPacketCounters(Ptr<const Packet> p);
  void UpdateDroppedPacketCounters(int nodeId, string packetType);
  void PhyRxDropReason(Ptr<const Packet> p, int nodeId);
  void PhyRxDrop0  (Ptr<const Packet> p) { PhyRxDropReason(p,  0); }
  void PhyRxDrop1  (Ptr<const Packet> p) { PhyRxDropReason(p,  1); }
  void PhyRxDrop2  (Ptr<const Packet> p) { PhyRxDropReason(p,  2); }
  void PhyRxDrop3  (Ptr<const Packet> p) { PhyRxDropReason(p,  3); }
  void PhyRxDrop4  (Ptr<const Packet> p) { PhyRxDropReason(p,  4); }
  void PhyRxDrop5  (Ptr<const Packet> p) { PhyRxDropReason(p,  5); }
  void PhyRxDrop6  (Ptr<const Packet> p) { PhyRxDropReason(p,  6); }
  void PhyRxDrop7  (Ptr<const Packet> p) { PhyRxDropReason(p,  7); }
  void PhyRxDrop8  (Ptr<const Packet> p) { PhyRxDropReason(p,  8); }
  void PhyRxDrop9  (Ptr<const Packet> p) { PhyRxDropReason(p,  9); }
  void PhyRxDrop10 (Ptr<const Packet> p) { PhyRxDropReason(p, 10); }
  void PhyRxDrop11 (Ptr<const Packet> p) { PhyRxDropReason(p, 11); }
  void PhyRxDrop12 (Ptr<const Packet> p) { PhyRxDropReason(p, 12); }
  void PhyRxDrop13 (Ptr<const Packet> p) { PhyRxDropReason(p, 13); }
  void PhyRxDrop14 (Ptr<const Packet> p) { PhyRxDropReason(p, 14); }
  void PhyRxDrop15 (Ptr<const Packet> p) { PhyRxDropReason(p, 15); }
  void PhyRxDrop16 (Ptr<const Packet> p) { PhyRxDropReason(p, 16); }
  void PhyRxDrop17 (Ptr<const Packet> p) { PhyRxDropReason(p, 17); }
  void PhyRxDrop18 (Ptr<const Packet> p) { PhyRxDropReason(p, 18); }
  void PhyRxDrop19 (Ptr<const Packet> p) { PhyRxDropReason(p, 19); }
  void PhyRxDrop20 (Ptr<const Packet> p) { PhyRxDropReason(p, 20); }
  void PhyRxDrop21 (Ptr<const Packet> p) { PhyRxDropReason(p, 21); }
  void PhyRxDrop22 (Ptr<const Packet> p) { PhyRxDropReason(p, 22); }
  void PhyRxDrop23 (Ptr<const Packet> p) { PhyRxDropReason(p, 23); }
  void PhyRxDrop24 (Ptr<const Packet> p) { PhyRxDropReason(p, 24); }
  void PhyRxDrop25 (Ptr<const Packet> p) { PhyRxDropReason(p, 25); }
  void PhyRxDrop26 (Ptr<const Packet> p) { PhyRxDropReason(p, 26); }
  void PhyRxDrop27 (Ptr<const Packet> p) { PhyRxDropReason(p, 27); }
  void PhyRxDrop28 (Ptr<const Packet> p) { PhyRxDropReason(p, 28); }
  void PhyRxDrop29 (Ptr<const Packet> p) { PhyRxDropReason(p, 29); }
  void PhyRxDrop30 (Ptr<const Packet> p) { PhyRxDropReason(p, 30); }
  void PhyRxDrop31 (Ptr<const Packet> p) { PhyRxDropReason(p, 31); }
  void PhyRxDrop32 (Ptr<const Packet> p) { PhyRxDropReason(p, 32); }
  void PhyRxDrop33 (Ptr<const Packet> p) { PhyRxDropReason(p, 33); }
  void PhyRxDrop34 (Ptr<const Packet> p) { PhyRxDropReason(p, 34); }
  void PhyRxDrop35 (Ptr<const Packet> p) { PhyRxDropReason(p, 35); }
  void PhyRxDrop36 (Ptr<const Packet> p) { PhyRxDropReason(p, 36); }
  void PhyRxDrop37 (Ptr<const Packet> p) { PhyRxDropReason(p, 37); }
  void PhyRxDrop38 (Ptr<const Packet> p) { PhyRxDropReason(p, 38); }
  void PhyRxDrop39 (Ptr<const Packet> p) { PhyRxDropReason(p, 39); }
  void PhyRxDrop40 (Ptr<const Packet> p) { PhyRxDropReason(p, 40); }
  void PhyRxDrop41 (Ptr<const Packet> p) { PhyRxDropReason(p, 41); }
  void PhyRxDrop42 (Ptr<const Packet> p) { PhyRxDropReason(p, 42); }
  void PhyRxDrop43 (Ptr<const Packet> p) { PhyRxDropReason(p, 43); }
  void PhyRxDrop44 (Ptr<const Packet> p) { PhyRxDropReason(p, 44); }
  void PhyRxDrop45 (Ptr<const Packet> p) { PhyRxDropReason(p, 45); }
  void PhyRxDrop46 (Ptr<const Packet> p) { PhyRxDropReason(p, 46); }
  void PhyRxDrop47 (Ptr<const Packet> p) { PhyRxDropReason(p, 47); }
  void PhyRxDrop48 (Ptr<const Packet> p) { PhyRxDropReason(p, 48); }
  void PhyRxDrop49 (Ptr<const Packet> p) { PhyRxDropReason(p, 49); }
  void PhyRxDrop50 (Ptr<const Packet> p) { PhyRxDropReason(p, 50); }
  void PhyRxDrop51 (Ptr<const Packet> p) { PhyRxDropReason(p, 51); }
  void PhyRxDrop52 (Ptr<const Packet> p) { PhyRxDropReason(p, 52); }
  void PhyRxDrop53 (Ptr<const Packet> p) { PhyRxDropReason(p, 53); }
  void PhyRxDrop54 (Ptr<const Packet> p) { PhyRxDropReason(p, 54); }
  void PhyRxDrop55 (Ptr<const Packet> p) { PhyRxDropReason(p, 55); }
  void PhyRxDrop56 (Ptr<const Packet> p) { PhyRxDropReason(p, 56); }
  void PhyRxDrop57 (Ptr<const Packet> p) { PhyRxDropReason(p, 57); }
  void PhyRxDrop58 (Ptr<const Packet> p) { PhyRxDropReason(p, 58); }
  void PhyRxDrop59 (Ptr<const Packet> p) { PhyRxDropReason(p, 59); }
  void PhyRxDrop60 (Ptr<const Packet> p) { PhyRxDropReason(p, 60); }
  void PhyRxDrop61 (Ptr<const Packet> p) { PhyRxDropReason(p, 61); }
  void PhyRxDrop62 (Ptr<const Packet> p) { PhyRxDropReason(p, 62); }
  void PhyRxDrop63 (Ptr<const Packet> p) { PhyRxDropReason(p, 63); }
  void PhyRxDrop64 (Ptr<const Packet> p) { PhyRxDropReason(p, 64); }
  void PhyRxDrop65 (Ptr<const Packet> p) { PhyRxDropReason(p, 65); }
  void PhyRxDrop66 (Ptr<const Packet> p) { PhyRxDropReason(p, 66); }
  void PhyRxDrop67 (Ptr<const Packet> p) { PhyRxDropReason(p, 67); }
  void PhyRxDrop68 (Ptr<const Packet> p) { PhyRxDropReason(p, 68); }
  void PhyRxDrop69 (Ptr<const Packet> p) { PhyRxDropReason(p, 69); }
  void PhyRxDrop70 (Ptr<const Packet> p) { PhyRxDropReason(p, 70); }
  void PhyRxDrop71 (Ptr<const Packet> p) { PhyRxDropReason(p, 71); }
  void PhyRxDrop72 (Ptr<const Packet> p) { PhyRxDropReason(p, 72); }
  void PhyRxDrop73 (Ptr<const Packet> p) { PhyRxDropReason(p, 73); }
  void PhyRxDrop74 (Ptr<const Packet> p) { PhyRxDropReason(p, 74); }
  void PhyRxDrop75 (Ptr<const Packet> p) { PhyRxDropReason(p, 75); }
  void PhyRxDrop76 (Ptr<const Packet> p) { PhyRxDropReason(p, 76); }
  void PhyRxDrop77 (Ptr<const Packet> p) { PhyRxDropReason(p, 77); }
  void PhyRxDrop78 (Ptr<const Packet> p) { PhyRxDropReason(p, 78); }
  void PhyRxDrop79 (Ptr<const Packet> p) { PhyRxDropReason(p, 79); }
  void PhyRxDrop80  (Ptr<const Packet> p) { PhyRxDropReason(p, 80); }
  void PhyRxDrop81  (Ptr<const Packet> p) { PhyRxDropReason(p, 81); }
  void PhyRxDrop82  (Ptr<const Packet> p) { PhyRxDropReason(p, 82); }
  void PhyRxDrop83  (Ptr<const Packet> p) { PhyRxDropReason(p, 83); }
  void PhyRxDrop84  (Ptr<const Packet> p) { PhyRxDropReason(p, 84); }
  void PhyRxDrop85  (Ptr<const Packet> p) { PhyRxDropReason(p, 85); }
  void PhyRxDrop86  (Ptr<const Packet> p) { PhyRxDropReason(p, 86); }
  void PhyRxDrop87  (Ptr<const Packet> p) { PhyRxDropReason(p, 87); }
  void PhyRxDrop88  (Ptr<const Packet> p) { PhyRxDropReason(p, 88); }
  void PhyRxDrop89  (Ptr<const Packet> p) { PhyRxDropReason(p, 89); }
  void PhyRxDrop90  (Ptr<const Packet> p) { PhyRxDropReason(p, 90); }
  void PhyRxDrop91  (Ptr<const Packet> p) { PhyRxDropReason(p, 91); }
  void PhyRxDrop92  (Ptr<const Packet> p) { PhyRxDropReason(p, 92); }
  void PhyRxDrop93  (Ptr<const Packet> p) { PhyRxDropReason(p, 93); }
  void PhyRxDrop94  (Ptr<const Packet> p) { PhyRxDropReason(p, 94); }
  void PhyRxDrop95  (Ptr<const Packet> p) { PhyRxDropReason(p, 95); }
  void PhyRxDrop96  (Ptr<const Packet> p) { PhyRxDropReason(p, 96); }
  void PhyRxDrop97  (Ptr<const Packet> p) { PhyRxDropReason(p, 97); }
  void PhyRxDrop98  (Ptr<const Packet> p) { PhyRxDropReason(p, 98); }
  void PhyRxDrop99  (Ptr<const Packet> p) { PhyRxDropReason(p, 99); }
  void PhyRxDrop100  (Ptr<const Packet> p) { PhyRxDropReason(p,100); }
  void PhyRxDrop101  (Ptr<const Packet> p) { PhyRxDropReason(p,  101); }
  void PhyRxDrop102  (Ptr<const Packet> p) { PhyRxDropReason(p,  102); }
  void PhyRxDrop103  (Ptr<const Packet> p) { PhyRxDropReason(p,  103); }
  void PhyRxDrop104  (Ptr<const Packet> p) { PhyRxDropReason(p,  104); }
  void PhyRxDrop105  (Ptr<const Packet> p) { PhyRxDropReason(p,  105); }
  void PhyRxDrop106  (Ptr<const Packet> p) { PhyRxDropReason(p,  106); }
  void PhyRxDrop107  (Ptr<const Packet> p) { PhyRxDropReason(p,  107); }
  void PhyRxDrop108  (Ptr<const Packet> p) { PhyRxDropReason(p,  108); }
  void PhyRxDrop109  (Ptr<const Packet> p) { PhyRxDropReason(p,  109); }
  void PhyRxDrop110  (Ptr<const Packet> p) { PhyRxDropReason(p,  110); }
  void PhyRxDrop111  (Ptr<const Packet> p) { PhyRxDropReason(p,  111); }
  void PhyRxDrop112  (Ptr<const Packet> p) { PhyRxDropReason(p,  112); }
  void PhyRxDrop113  (Ptr<const Packet> p) { PhyRxDropReason(p,  113); }
  void PhyRxDrop114  (Ptr<const Packet> p) { PhyRxDropReason(p,  114); }
  void PhyRxDrop115  (Ptr<const Packet> p) { PhyRxDropReason(p,  115); }
  void PhyRxDrop116  (Ptr<const Packet> p) { PhyRxDropReason(p,  116); }
  void PhyRxDrop117  (Ptr<const Packet> p) { PhyRxDropReason(p,  117); }
  void PhyRxDrop118  (Ptr<const Packet> p) { PhyRxDropReason(p,  118); }
  void PhyRxDrop119  (Ptr<const Packet> p) { PhyRxDropReason(p,  119); }
  void PhyRxDrop120  (Ptr<const Packet> p) { PhyRxDropReason(p,  120); }
  void PhyRxDrop121  (Ptr<const Packet> p) { PhyRxDropReason(p,  121); }
  void PhyRxDrop122  (Ptr<const Packet> p) { PhyRxDropReason(p,  122); }
  void PhyRxDrop123  (Ptr<const Packet> p) { PhyRxDropReason(p,  123); }
  void PhyRxDrop124  (Ptr<const Packet> p) { PhyRxDropReason(p,  124); }
  void PhyRxDrop125  (Ptr<const Packet> p) { PhyRxDropReason(p,  125); }
  void PhyRxDrop126  (Ptr<const Packet> p) { PhyRxDropReason(p,  126); }
  void PhyRxDrop127  (Ptr<const Packet> p) { PhyRxDropReason(p,  127); }
  void PhyRxDrop128  (Ptr<const Packet> p) { PhyRxDropReason(p,  128); }
  void PhyRxDrop129  (Ptr<const Packet> p) { PhyRxDropReason(p,  129); }
  void PhyRxDrop130  (Ptr<const Packet> p) { PhyRxDropReason(p,  130); }
  void PhyRxDrop131  (Ptr<const Packet> p) { PhyRxDropReason(p,  131); }
  void PhyRxDrop132  (Ptr<const Packet> p) { PhyRxDropReason(p,  132); }
  void PhyRxDrop133  (Ptr<const Packet> p) { PhyRxDropReason(p,  133); }
  void PhyRxDrop134  (Ptr<const Packet> p) { PhyRxDropReason(p,  134); }
  void PhyRxDrop135  (Ptr<const Packet> p) { PhyRxDropReason(p,  135); }
  void PhyRxDrop136  (Ptr<const Packet> p) { PhyRxDropReason(p,  136); }
  void PhyRxDrop137  (Ptr<const Packet> p) { PhyRxDropReason(p,  137); }
  void PhyRxDrop138  (Ptr<const Packet> p) { PhyRxDropReason(p,  138); }
  void PhyRxDrop139  (Ptr<const Packet> p) { PhyRxDropReason(p,  139); }
  void PhyRxDrop140  (Ptr<const Packet> p) { PhyRxDropReason(p,  140); }
  void PhyRxDrop141  (Ptr<const Packet> p) { PhyRxDropReason(p,  141); }
  void PhyRxDrop142  (Ptr<const Packet> p) { PhyRxDropReason(p,  142); }
  void PhyRxDrop143  (Ptr<const Packet> p) { PhyRxDropReason(p,  143); }
  void PhyRxDrop144  (Ptr<const Packet> p) { PhyRxDropReason(p,  144); }
  void PhyRxDrop145  (Ptr<const Packet> p) { PhyRxDropReason(p,  145); }
  void PhyRxDrop146  (Ptr<const Packet> p) { PhyRxDropReason(p,  146); }
  void PhyRxDrop147  (Ptr<const Packet> p) { PhyRxDropReason(p,  147); }
  void PhyRxDrop148  (Ptr<const Packet> p) { PhyRxDropReason(p,  148); }
  void PhyRxDrop149  (Ptr<const Packet> p) { PhyRxDropReason(p,  149); }
  void PhyRxDrop150  (Ptr<const Packet> p) { PhyRxDropReason(p,  150); }
  void PhyRxDrop151  (Ptr<const Packet> p) { PhyRxDropReason(p,  151); }
  void PhyRxDrop152  (Ptr<const Packet> p) { PhyRxDropReason(p,  152); }
  void PhyRxDrop153  (Ptr<const Packet> p) { PhyRxDropReason(p,  153); }
  void PhyRxDrop154  (Ptr<const Packet> p) { PhyRxDropReason(p,  154); }
  void PhyRxDrop155  (Ptr<const Packet> p) { PhyRxDropReason(p,  155); }
  void PhyRxDrop156  (Ptr<const Packet> p) { PhyRxDropReason(p,  156); }
  void PhyRxDrop157  (Ptr<const Packet> p) { PhyRxDropReason(p,  157); }
  void PhyRxDrop158  (Ptr<const Packet> p) { PhyRxDropReason(p,  158); }
  void PhyRxDrop159  (Ptr<const Packet> p) { PhyRxDropReason(p,  159); }



  bool IsCTL_ACKPacket(ns3::Ptr<const ns3::Packet>);
  bool IsAODV_RREPPacket(ns3::Ptr<const ns3::Packet>);
  bool IsAODV_RREQPacket(ns3::Ptr<const ns3::Packet>);
  bool IsARPPacketRequest(Ptr<const ns3::Packet> p);
  bool IsARPPacketReply(Ptr<const ns3::Packet> p);
  bool IsPayloadPacket(ns3::Ptr<const ns3::Packet>);
  bool IsDSDV_RREPPacket(ns3::Ptr<const ns3::Packet>);
  bool IsDSDV_RREQPacket(ns3::Ptr<const ns3::Packet>);
  bool IsIPv4Header(ns3::Ptr<const ns3::Packet>);
  void ForwardPacket(const Ipv4Header &header, Ptr<const Packet> packet, uint32_t interface);
  void ReceivedPacket(Ptr<Socket> skt);
  Ptr<Socket> SetupPacketReceive (Ipv4Address addr, Ptr<Node> node);


  // Traces
  void Callbacks();

  // Print Information
  void PrintDevices(NodeContainer myNodes, string containerName);
  void PrintNodesIPAddress(ns3::NodeContainer, std::string);

  // Stuff
  void NodeListStuff();
  void NetDeviceStuff();
```

```
  void LogStuff();

  // Management
  void   NetworkCommandCentre();
  void   DenyAllRequestForControlledNode(double backOffUntil);
  void   DenyARequestForControlledNode(double backOffUntil, int mNode);
  void   GrantRequestForControlledNode(int manager, double backOffTime);
  void   GrantRequestForSwitchProtocol(double backOffTime);
  void   GrantRequestForChangeChannel(double backOffTime);
  double TimeAllowedToMoveControlledNode();
  int    GetSourceNodeForControlledNode(int mNode);
  int    GetDestinationNodeForSourceNode(int sNode);
  Vector GetControlNodeMidPointXY(int sNode, int dNode, Vector cNode);
  double MoveControlledNode(int mNode);
  void   MoveControlledNodeTo (int nodeId, double x, double y);
  int    GetClosestControlledNode(int sNode, int dNode);
  double GetTimeOfJourneyControlledNode (Vector node, double x, double y, double speed);
  void   PrintVectorOfMemoryEntries();
  void   UpdatePerformance(double atTime, int manager, char action);
  void   AutonomousControlledNode(int n);
  void   CourseChange(Ptr<const MobilityModel> model);

  uint32_t GetLengthOfInteger(uint32_t lengthOfThis);



  string InRangeLongestDuration(NodeContainer nodesInRange, double duration, int sourceNode);
  void GetNeighboursInCommon (NodeContainer nodes, uint32_t nodeId1, uint32_t nodeId2);
  NodeContainer CheckNeighbours(int nodes, int nodeId1);

  void TraceEvents();
  string SpacesAndTail(uint32_t noOfSpaces);
  void SourceNodeStuff(uint32_t sourceNodeToAdd);
  bool Configure(int argc, char **argv);
  void SetBatteries();
  void PlaceNodes();
  void NodeMobility();
  void ApplicationOnOffStuff();
  void MobilityRandomWayPoint(double);
  void MobilityOurStuff();
  Ipv4Address GetIPAddressFromNodeId(int nodeId);
  void DoStats();
  void LogDevQueueEnqueue();
  void Stats();
  void SwitchProtocol();
  string PrintReceivedPacket (Ptr<Socket> socket, Ptr<Packet> packet);
  //static inline  std::string PrintReceivedPacket (Ptr<Socket> socket, Ptr<Packet> packet,
NodeContainer allNodes);
  void ReceivePacket(ns3::Ptr<ns3::Socket>);

  void ScheduleStuff();                             //required  - start simulation and
schedule events
  void FlowMonitorStuff(Ptr<FlowMonitor> monitor);  //required  - record udp / tcp flows
  void RecordPosition(int nodeId);                  //required  - record current position of a
node
  bool InRange(int distance);                       //required  - given a distance is it
within a given range?
  void PrintNodePosition(int nodeId);               //required  - Print X and Y for a given
node
  Vector GetNodePosition(uint32_t nodeId);     //required  - Get a vector containing x and y
for a given node
  Vector GetNodePosition(uint32_t nodeId, NodeContainer nodes);
  Vector GetControlledNodePosition(uint32_t nodeId);     //required  - Get a vector containing
x and y for a given node
  double GetDistance( ns3::Vector position1,
                      ns3::Vector position2);       //required  - Get distance between two
Vectors
  void CheckNeighbours(int sourceNodeId);           //required  - record neighbour nodes in
range of sourceNode
  void GetAllDistancesFromSource(int sourceNodeId); //required  - record ALL nodes distances
from sourceNode
  ns3::Vector PredictPosition(int nodeId, double futureTime); //required  - Get Predicted
position for a given node at some time in the future
  void MoveNodeTo (int nodeId, double x, double y); //required  - Move node to new position

  void NodeWalkTo (int nodeId, double xGoal,        //required  - Walk a node to a new
position in steps.
```

```
                   double yGoal, double stepSize,   //            stepSize is used to
calculate number of steps
                   double speed);
                       //           dependent on distance.  Step speed is how often to take a
step.


  void ControlledNodeWalkTo (int nodeId, double xGoal, double yGoal, double stepSize, double
speed);
  void PrintNodesContainerInfo(NodeContainer myNodes, string containerName);  //required -
Prints out Node Container information.
  void CreateVectorOfDistancesAllNodes();           //required  - Store ALL information about
distances for each and every node
  void WriteVectorOfNodeDistanceToDisk(vector<nodeDistance> theDistances, string theFilename);
//required - save vector of positions to disk
  void WriteVectorOfMemorySystemStateToDisk(string theFilename);
  void PrintFlowMonitorInformation(Ptr<FlowMonitor> monitor);
  void CreateVectorFlowMonitorInformation(Ptr<FlowMonitor> monitor);
  void WriteVectorFlowMonitorToDisk(string theFilename);
  void Election();
  int GetManagerToMoveNodeFor();

  void PlaceNodesRandomLocation();                           //required  - place nodes randomly
in the world
  void PlaceNodesRandomLocation(NodeContainer nodes);                          //required  -
place nodes randomly in the world
  void PlaceNodesOneLocation();
  void PlaceNodesTwoGroups();                            //required  - all nodes in one place
  void PlaceNodesTopLCRMiddleLCRBottomLCM();
  void DsdvPositionMatchAodv();
  void AodvPositionMatchDsdv();
  void InstallConstantMobility();
  void PrintVectorOfDistances(std::vector<CogMan::nodeDistance> theDistances);
  void WriteVectorOfNodePositionsToDisk(string theFilename);
  void PrintVectorOfNodePositions(vector<currentPosition> theDistances, string theFilename);
  void ArchimedeanSpiral(int nodeId);
  void ArchimedeanSpiralClockwise(int nodeId);
  bool SortByTime(double &a, double &b);
  void CreateTrafficFlow(int sourceNodeId, int destinationNodeId, double startAt, double
stopAt);
  void CreateAodvTrafficFlow(int sourceNodeId, int destinationNodeId, double startAt, double
stopAt);
  void CreateDsdvTrafficFlow(int sourceNodeId, int destinationNodeId, double startAt, double
stopAt);
  void SortVector();  // not required at the moment
  void PrintRemainingEnergy(int nodeId);
  void Antenna();
  void MacTxDrop(Ptr<const ns3::Packet>);
  void PhyTxDrop(Ptr<const ns3::Packet>);
  void PhyRxDrop(Ptr<const ns3::Packet>);
  double GetRemainingEnergy(int nodeId);
  double GetRemainingEnergyES(int nodeId);
  void RemoveDepleatedNodes();
  void SetLogging();
  void Print(string message, int type);
  void SendPacket(Ptr<const Packet> p);
  double GetPredictedMaximalTimeInRange(int sourceNodeId, int destinationNodeId);
  void RecordElectionWinner(int sourceNodeId, int managerNodeId, double timeNow, double
timeInRangeUntil, double timePowerUntil, double density);
  void RecordElectionCandidate(int sourceNodeId, int neighbourNodeId, double timeNow, double
distance, double timeInRangeUntil, double density);
  void RecordElectionCandidatePercents(int sourceNodeId, int neighbourNodeId, double timeNow,
double distance, double timeInRangeUntil, double density);
  double GetPredictedTimePowerUntil(int nodeId);
  int GetSecondsFromSeconds(int seconds);
  int GetMinutesFromSeconds(int seconds);
  int GetHoursFromSeconds(int seconds);
  void SwitchProtocolToDSDV();
  void PrintNodeInformation(Ptr<Node> node);
  string GetProtocol(Ipv4Address addr);
  void RequestAction(double atTime, int mNode, char request, double prr);

  int GetNumberOfNeighbours(int sourceNodeId, NodeContainer nodes);
  Vector GetNewVectorUsingAngleAndDistance(Vector currentPosition, double distance, double
degree);
  double GetDensityFromNode(int nodeId);
  void LakeScenario();
  void PlaceNodesOneLocation(NodeContainer nodes);
  void InstallConstantMobility(NodeContainer nodes);
  void ScheduleEvents();
  void RecordSystemState(int sNode, int mNode);
  void WriteVectorOfReceivedPacketsToDisk(string theFilename);
  void WriteVectorOfDroppedPacketsToDisk(string theFilename);
  void WriteVectorOfSentPacketsToDisk(string theFilename);
  void WriteVectorOfSentPacketsBCPToDisk(string theFilename);
  void WriteVectorOfAllPacketsToDisk(string theFilename);
  void WriteVectorOfPacketRatesToDisk(string theFilename);
  void WriteVectorOfManagementNodeMemoryToDisk(string theFilename);
  void WriteAODVvsDSDVStats(double interval);

  double GetChannel(int mNode);
  void PrintAllChannels();
  bool CriticalDensity(double currentDensity, double refDef);

  void ChangeEnergy();
  void SetRemainingEnergy(int nodeId);
  void SetDsdvRemainingEnergy(int nodeId);
  void SetAodvRemainingEnergy(int nodeId);
  void RecordReceivedPacket(Ipv4Address src, Ipv4Address dst);
  void RecordDroppedPacket(Ipv4Address src, Ipv4Address dst);
  void RecordSentPacket(Ipv4Address src, Ipv4Address dst);
  void PrintAodvFrequency();
  void Management(int sNode, int mNode);
  void SplitNodesTwoEvenGroups();
  double GetTimeOfJourney (int nodeId, double xGoal, double yGoal, double speed);
  double GetTimeOfJourney (double x, double y, double xGoal, double yGoal, double speed);

  void MoveNodeDiamondPattern();
  void CreateTrafficFlows(int n);

  void SelectManagementNode(int nodeId, double timeNow);
  void SetChannel(int mNode, int channelNumber);
  void SetChannelControlledNodes (int mNode, int channelNumber);
  double GetCurrentReceiveRate();
  double GetCurrentDropRate();
  void TotalNumberOfPackets();
  bool IsNodeMoving(int nodeId, int x, int y, double stride, double speed);
  bool IsControlledNodeMoving(int nodeId, int x, int y, double stride, double speed);
  double IsControlledNodeMoving(int nodeId);
  double GetTimeNow(char t);
  void RecordMovement(int nodeId, double endTime);
  void RecordControlledNodeMovement(int nodeId, double endTime);
  void CreateCongestionNodes(int n);
  void CreateCongestionInterface();
  void CreateCongestionAddresses();
  void CreateCongestionStack();
  void CreateCongestionDevices();
  void CreateTrafficFlowsForCongestion();
  void CreateTrafficFlowForCongestion(int sourceNodeId, int destinationNodeId, double startAt,
double stopAt);
  void CreateCongestionTrafficFlow(int sourceNodeId, int destinationNodeId, double startAt,
double stopAt);

  bool WillBeAlive(int, double);
  bool RecordRelationshipState(int, int);
  bool StillTransmitting(int);
  double GetCurrentReceiveRate(int);
  void AnkaSwitching(int, int);
  bool WillBeInRange(int, int, double);
  void ConfigureNetAmin(AnimationInterface& anim);
  void WriteInformationToDisk(string &fn);
  void RecordSendPacketInformation(Ptr<const Packet> p);
  bool RecordDroppedPacketInformation(Ptr<const Packet> p);
  int GetCurrentReceivedPacketsForSource(int nodeId, double timeFrom, double timeUntil);
  int GetCurrentSentPacketsForSource(int nodeId, double timeFrom, double timeUntil);
  int GetCurrentReceivedPacketsForDestinationFromSource(int sNode, int dNode, double timeFrom,
double timeUntil);
  int GetCurrentDroppedPacketsForSource(int nodeId, double timeFrom, double timeUntil);
  string ConvertIPAddressToString(Ipv4Address addr);
  string ConvertDoubleToString(double doubleObject);
  void RecordPacketRate(double timeNow, int source, double sent, double received, double
dropped);
```

```
  bool CaseToSwitchProtocol();
  void RecordActionRequest(double timeNow, int sNode, int mNode, double droPerSec, double
recPerSec, double senPerSec, double density, int channel, string currentProtocol, char
request);
  string GetAction(char gutFeeling, int mNode, double droPerSec, double recPerSec, double
senPerSec, double density, int channel, string currentProtocol);
  void RecordActionOutcome(double timeBefore, int mNode, char action);
  double GetPeriodToStablise(string action);
  void PerformAction(string action, int sNode);
  void PrintAllPacketStats(double fromTime, double untilTime);
  void TestRadius();
  void Set21mRadius();
  void Set100mRadius();
//std::stringstream sstm;
//sstm << name << age;
//result = sstm.str();

  AodvHelper                  aodv;
  NodeContainer               aodvNodes;
  NetDeviceContainer          aodvDevices;
  InternetStackHelper         aodvStack;
  Ipv4AddressHelper           aodvAddresses;
  Ipv4InterfaceContainer      aodvInterface;
  BasicEnergySourceHelper     aodvBasicSource;
  WifiRadioEnergyModelHelper  aodvRadioEnergy;
  DeviceEnergyModelContainer  aodvDeviceEnergyModel;
  EnergySourceContainer       aodvEnergySources;

  DsdvHelper                  dsdv;
  NodeContainer               dsdvNodes;
  NetDeviceContainer          dsdvDevices;
  InternetStackHelper         dsdvStack;
  Ipv4AddressHelper           dsdvAddresses;
  Ipv4InterfaceContainer      dsdvInterface;
  BasicEnergySourceHelper     dsdvBasicSource;
  WifiRadioEnergyModelHelper  dsdvRadioEnergy;
  DeviceEnergyModelContainer  dsdvDeviceEnergyModel;
  EnergySourceContainer       dsdvEnergySources;

  NodeContainer               congestionNodes;
  NetDeviceContainer          congestionDevices;
  InternetStackHelper         congestionStack;
  Ipv4AddressHelper           congestionAddresses;
  Ipv4InterfaceContainer      congestionInterface;

  NodeContainer               aodvControlledNodes;
  NetDeviceContainer          aodvControlledDevices;
  InternetStackHelper         aodvControlledStack;
  Ipv4AddressHelper           aodvControlledAddresses;
  Ipv4InterfaceContainer      aodvControlledInterface;
  BasicEnergySourceHelper     aodvControlledBasicSource;
  WifiRadioEnergyModelHelper  aodvControlledRadioEnergy;
  DeviceEnergyModelContainer  aodvControlledDeviceEnergyModel;
  EnergySourceContainer       aodvControlledEnergySources;

  NodeContainer               dsdvControlledNodes;
  NetDeviceContainer          dsdvControlledDevices;
  InternetStackHelper         dsdvControlledStack;
  Ipv4AddressHelper           dsdvControlledAddresses;
  Ipv4InterfaceContainer      dsdvControlledInterface;
  BasicEnergySourceHelper     dsdvControlledBasicSource;
  WifiRadioEnergyModelHelper  dsdvControlledRadioEnergy;
  DeviceEnergyModelContainer  dsdvControlledDeviceEnergyModel;
  EnergySourceContainer       dsdvControlledEnergySources;

  NodeContainer               allNodes;
  NetDeviceContainer          allDevices;
  Ipv4InterfaceContainer      allInterfaces;

  NodeContainer               allControlledNodes;

  MobilityHelper mobility;
  MobilityHelper mobility2;
  //BulkSendHelper myBulkApplication;

  BasicEnergySourceHelper    basicSourceHelper;
  WifiRadioEnergyModelHelper radioEnergyHelper;
  DeviceEnergyModelContainer deviceModels;
  EnergySourceContainer      sources;
  YansWifiPhyHelper wifiPhy;



  //AnimationInterface anim (const string "netanim.xml");

  // for our logging

  std::stringstream message;

  bool functionMessageStart             = false;
  bool functionMessageEnd               = false;
  bool outputVariableValue              = false;
  bool packetDropInfo                   = false;
  bool packetSentInfo                   = false;
  bool packetRecdInfo                   = false;
  bool packetFwrdInfo                   = false;
  bool packetRouteInfo                  = false;
  bool distanceInfo                     = false;
  bool deviceInfo                       = false;
  bool addressInfo                      = false;
  bool trafficFlowInfo                  = false;
  bool electionInfo                     = true;
  bool destinationInfo                  = false;
  bool recordPositionInfo               = false;
  bool remainingEnergyInfo              = true;
  bool allDistancesFromSourceInfo       = false;
  bool neigbourDistancesFromSourceInfo  = false;
  bool debugInSteps                     = false;
  bool routingInformation               = false;
  bool nodeInfo                         = false;
  int  numberOfSteps                    = 5;
  int  debugStepCount                   = 0;


  double timeInRangeWeight  = 0.8;
  double timeInPowerWeight  = 0.1;
  double densityWeight      = 0.1;



private:
  void CreateAodvDevices();
  void CreateAodvStack();
  void CreateAodvAddresses();
  void CreateAodvInterface();

  void CreateDsdvDevices();
  void CreateDsdvStack();
  void CreateDsdvAddresses();
  void CreateDsdvInterface();

  void CreateControlledAodvDevices();
  void CreateControlledAodvStack();
  void CreateControlledAodvAddresses();
  void CreateControlledAodvInterface();

  void CreateControlledDsdvDevices();
  void CreateControlledDsdvStack();
  void CreateControlledDsdvAddresses();
  void CreateControlledDsdvInterface();

public: // methods
  void CreateNodes(int n);
  void CreateDsdvNodes(int n);
  void CreateAodvNodes(int n);
  void CreateControlledAodvNodes(int n);
  void CreateControlledDsdvNodes(int n);
} test;

int main (int argc, char **argv)
{
  test.Configure(argc, argv);
  test.SetLogging();
  test.ScheduleEvents();
  return 0;
};

CogMan::CogMan() :
  numOfNodes (10),           // Default values for simulation
  dNode (9),
  currentProtocol ("aodv"),
  simulationTime (600.0), // we initialise here so that the values can be overridden from the
command prompt
  txPower (10),          // transmission power.  Typicallay between -30dBm and +4dBm (Decibel-
milliwatts)
  logFile ("logfile"),      // name of the log file.
  bytesTotal(0),
  packetsReceived(0),
  port(9),
  worldSize(50000),
  myRadius(100),        // Range of transmission in meters

  EnergyDetectionThreshold(-61.76),  // original value was -96.0
  CcaMode1Threshold(-61.76),         // original value was -99.0
  TxGain(4.5),                      // original value was 4.5
  RxGain(4.5),                      // original value was 4.5
  TxPowerLevels(1),                 // attributes are actually from
  TxPowerEnd(16),                   // YansWifiPhy class
  TxPowerStart(16),                 // has full list of attributes
  RxNoiseFigure(4),
  initialEnergy(11016),
  checkingIntervalTime(2),
  timeToStabilize(6)
{
}

bool
CogMan::Configure(int argc, char **argv)
{
  Print("Configure",1);
  CommandLine cmd;
  cmd.AddValue("numOfNodes",             "Number of Nodes",                  numOfNodes);
  cmd.AddValue("numOfTrafficFlows",      "Number of Traffic Flows",
numOfTrafficFlows);
  cmd.AddValue("numOfCNodes",            "Number of Controllable Nodes",
numOfControlledNodes);
  cmd.AddValue("dataRate",               "Default data rate",                dataRate);
  cmd.AddValue("cognition",              "Cognition on/off",                 cognition);
  cmd.AddValue("simulationTime",         "Simulation Time in Seconds",       simulationTime);
  cmd.AddValue("radius",                 "Transmission Radius",              myRadius);
  cmd.AddValue("logFile",                "Log File",                         logFile);
  cmd.AddValue("mobilityModel",          "Mobility Model",                   mobilityModel);
  cmd.AddValue("initialProtocol",        "Initial Starting Protocol",        initialProtocol);
  cmd.AddValue("cNodeSpeed",             "Speed of Controlled Nodes",        cNodeSpeed);
  cmd.AddValue("nodeSpeed",              "Speed of Nodes",                   nodeSpeed);
  cmd.AddValue("recordPositionInterval", "The interval to record positions",
recordPositionInterval);
  cmd.AddValue("folderName",             "Folder for trace files",           folderName);
  cmd.Parse(argc, argv);

  fn = folderName.c_str();
  const char *tt = folderName.c_str();
  if (fn != "")
    mkdir(tt, 0700);

  Print("Configure",2);
  return true;
}

void CogMan::SetLogging()
{
  functionMessageStart            = false;     // type 1
  functionMessageEnd              = false;     // type 2

  outputVariableValue             = false;    // type 3

  packetDropInfo                  = false;     // type 4
  packetSentInfo                  = false;     // type 5
  packetRecdInfo                  = false;     // type 6
  packetFwrdInfo                  = false;    // type 43
  packetRouteInfo                 = false;    // type 42
  distanceInfo                    = false;     // type 7

  deviceInfo                      = false;     // type 8

  addressInfo                     = false;     // type 9

  trafficFlowInfo                 = false;      // type 10

  electionInfo                    = false;      // type 11

  destinationInfo                 = false;     // type 12

  recordPositionInfo              = false;    // type 13

  remainingEnergyInfo             = false;     // type 14

  allDistancesFromSourceInfo      = false; // type 15
  neigbourDistancesFromSourceInfo = false; // type 16

  routingInformation              = false;     // type 17

  nodeInfo                        = false;     // type 18


  Print("DeviceContainerStuff",2);


  debugInSteps          = false;    //
  numberOfSteps         = 50;       //
}

// Scheduling
void
CogMan::ScheduleEvents()
{
  Print("ScheduleEvents",1);
  folderName = folderName + "/";

  cout << GetTimeNow('s') << "s Simulation Started\n";

  totalAodvPacketsSent=0;
  totalAodvPacketsDropped=0;
  totalAodvPacketsReceived=0;
  totalDsdvPacketsSent=0;
  totalDsdvPacketsDropped=0;
  totalDsdvPacketsReceived=0;
  expectedEstimatedSum = 0;
  // packets that are sent
  sendPacketTotal=0;
  sendDsdvDataPacket=0;
  sendAodvDataPacket=0;
  sendDsdvControlPacket=0;
  sendAodvControlPacket=0;
  // packets for current protocol in use
  totalSwitchingReceived = 0;
  totalSwitchingSent     = 0;
  totalSwitchingDropped  = 0;
  maxReceivingRate       = 0;

  // cognition switches
  allowedToSwitchChannel      = true;
  allowedToSwitchProcol       = true;
  allowedToMoveControlledNode = true;

  bool switchProtocolbool  = true;
  bool switchChannelbool   = true;
  bool controlledNodesbool = true;

  Config::SetDefault("ns3::WifiMacQueue::MaxPacketNumber", UintegerValue(1));
  Config::SetDefault("ns3::WifiMacQueue::MaxDelay",        TimeValue(Seconds(0.001)));
  Config::SetDefault("ns3::OnOffApplication::PacketSize",  StringValue ("64"));//bytes?
  Config::SetDefault("ns3::OnOffApplication::DataRate",    StringValue ("108Mbps"));



  if (myRadius = 100)
```

```
    Set100mRadius();

  CreateAodvNodes(numOfNodes);
  CreateDsdvNodes(numOfNodes);
  allNodes = NodeContainer::GetGlobal();

  if (mobilityModel == 'L')
  {
  PlaceNodesTwoGroups();
  InstallConstantMobility(aodvNodes);
  InstallConstantMobility(dsdvNodes);
  Simulator::Schedule (Seconds(0.1), &CogMan::LakeScenario, this);
  }
  else if (mobilityModel == 'R')
  {
    MobilityRandomWayPoint(nodeSpeed);
    InstallConstantMobility(dsdvNodes);
  }

  DsdvPositionMatchAodv();

  for (int nodeId=0; nodeId<aodvNodes.GetN(); nodeId++)
  {
    RecordPosition(nodeId);
  }

  if (cognition)
  {
    CreateControlledAodvNodes(numOfControlledNodes);
    CreateControlledDsdvNodes(numOfControlledNodes);
    allControlledNodes = NodeContainer(aodvControlledNodes, dsdvControlledNodes);

    for (int n=0; n<numOfControlledNodes; n++) // number of controlled nodes
    {
      AutonomousControlledNode(n);
    }

    Simulator::Schedule (Seconds (2), &CogMan::Election, this);
  }

  Callbacks();
  CreateTrafficFlows(numOfTrafficFlows);





  // PrintAodvFrequency();

  //  NodeWalkTo (int nodeId, double xGoal, double yGoal, double stepSize (per meter), double
speed (mps))
  //  Average step size (stide) is 0.8 meters per stride
  //  Average walking speed is 1.79 meters per second
  //  Simulator::Schedule (Seconds (i), &CogMan::NodeWalkTo, this, 1,   i,      150.0,    0.8,
3.79);

    // Simulator::Schedule (Seconds (1), &CogMan::SetRemainingEnergy, this, 1);
    //    // election in 5 seconds, node 1 source.
    // Simulator::Schedule (Seconds(6.0), &CogMan::MoveNodeDiamondPattern, this); // back
    //
    //Simulator::Schedule (Seconds(4.0), &CogMan::MoveControlledNode, this);
    //Simulator::Schedule (Seconds(stabilizingTime), &CogMan::NetworkCommandCentre,
this);//back
    //Simulator::Schedule (Seconds(0.0), &CogMan::TotalNumberOfPackets, this);
    //Simulator::Schedule (Seconds(10.0), &CogMan::SetChannel, this, 1,4);
    // cout << "\n\n\n density is " << GetDensityFromNode(1) << "\n\n\n";
    //  Simulator::Schedule (Seconds (3), &CogMan::SwitchProtocolToDSDV, this);
    //Simulator::Schedule (Seconds (49), &CogMan::PrintRemainingEnergy, this, 1);
    //Simulator::Schedule (Seconds (3), &CogMan::RemoveDepleatedNodes, this);



    Ptr<FlowMonitor> flowmon;
    FlowMonitorHelper flowmonHelper;
    flowmon = flowmonHelper.InstallAll ();
    //Config::ConnectWithoutContext
("/NodeList/*/DeviceList/*/$ns3::TcpL4Protocol/UnicastForward",
    //  MakeCallback (&forwardedhmm));

    //Ipv4RoutingProtocol::UnicastForwardCallback ucb = MakeCallback (&forwardedhmm);
    DoStats();
    Simulator::Stop (Seconds (simulationTime));

    // for trace metrics.
    //AsciiTraceHelper wifiAscii;
    //Ptr<OutputStreamWrapper> wifiStream = wifiAscii.CreateFileStream(folderName +
"tracemetrics.tr");
    //wifiPhy.EnableAsciiAll(wifiStream);
    //wifiPhy.EnablePcapAll (logFile);


    AnimationInterface anim(folderName + "netAnim.xml");
    ConfigureNetAmin(anim);
    anim.SetMobilityPollInterval (Seconds (100.0));

    Simulator::Run ();
    Simulator::Destroy ();
    NS_LOG_INFO ("Done.");

    flowmon->SerializeToXmlFile (folderName + "flowmonFile", false, false);
    flowmon->CheckForLostPackets ();

    //test.PrintFlowMonitorInformation(monitor);
      cout << "END OF SIMULATION number 1!";





  /*
  TotalNumberOfPackets();
  int h=100*counterGoodJob*checkingIntervalTime/simulationTime;
  int hh=100*counterGoodAodv*checkingIntervalTime/simulationTime;
  int hhh=100*counterGoodDsdv*checkingIntervalTime/simulationTime;
  cout << "\nTotal Packets received in Aodv is: " << totalAodvPacketsReceived <<endl;
  cout << "\nTotal Packets received in Dsdv is: " << totalDsdvPacketsReceived << endl;
  cout << " We were on the best working protocol " << counterGoodJob
       << " times, which is: " << h
       << " percent of all times!!!"<< endl;
  cout << " Dsdv was working better " << counterGoodDsdv << " times, which is " << hhh << "
percent of all times"<<endl;
  cout << " Aodv was working better " << counterGoodAodv << " times, which is " << hh << "
percent of all times"<<endl;
  */

  Print("ScheduleEvents",2);
  cout << "END OF SIMULATION!";
  DoStats();
  WriteInformationToDisk(folderName);
  WriteAODVvsDSDVStats(1.0);

  cout << "\nEnd of Simulation Run";

  //cout << "\nTotal Packets dropped is: " << totalPacketsDropped;
 /* TotalNumberOfPackets();
  int h=100*counterGoodJob*checkingIntervalTime/simulationTime;
  int hh=100*counterGoodAodv*checkingIntervalTime/simulationTime;
  int hhh=100*counterGoodDsdv*checkingIntervalTime/simulationTime;
  cout << "\nTotal Packets received in Aodv is: " << totalAodvPacketsReceived <<endl;
  cout << "\nTotal Packets received in Dsdv is: " << totalDsdvPacketsReceived << endl;
  cout << " We were on the best working protocol " << counterGoodJob
       << " times, which is: " << h
       << " percent of all times!!!"<< endl;
  cout << " Dsdv was working better " << counterGoodDsdv << " times, which is " << hhh << "
percent of all times"<<endl;
  cout << " Aodv was working better " << counterGoodAodv << " times, which is " << hh << "
percent of all times"<<endl;

  */
}
```

```
double
CogMan::GetTimeNow(char t)
{
  if (t == 's')
  {
    return Simulator::Now().GetSeconds();
  }
}

int
CogMan::GetHoursFromSeconds(int seconds)
{
  return seconds / 3600;
  //
}

int
CogMan::GetMinutesFromSeconds(int seconds)
{
  int hours = seconds / 3600;
  seconds = seconds - (hours * 3600);
  return seconds / 60;
}

int
CogMan::GetSecondsFromSeconds(int seconds)
{
  int hours = seconds / 3600;
  seconds = seconds - (hours * 3600);
  int minutes = seconds / 60;
  seconds = seconds - (minutes * 60);
  return seconds;
}

// NODE CONFIGURATION

void
CogMan::CreateAodvNodes(int n)
{
  Print("CreateAodvNodes",1);
  aodvNodes.Create(n);
  message << "Created " << n << " nodes in AODV Container";
  Print(message.str(),18);

  CreateAodvDevices();
  CreateAodvStack();
  CreateAodvAddresses();
  CreateAodvInterface();

  SetBatteries();

  if (nodeInfo)
  {
    for (NodeContainer::Iterator n = aodvNodes.Begin(); n < aodvNodes.End(); n++)
    {
      PrintNodeInformation((*n));
    }
  }
  Print("CreateAodvNodes",2);
}

void
CogMan::CreateAodvDevices()
{
  Print("CreateAodvDevices",1);
  wifiPhy = YansWifiPhyHelper::Default();     /* set to default working state */

  wifiPhy.Set ("EnergyDetectionThreshold", DoubleValue (EnergyDetectionThreshold));   //
original value was -96.0
  wifiPhy.Set ("CcaMode1Threshold", DoubleValue (CcaMode1Threshold) );                //
original value was -99.0
  wifiPhy.Set ("TxGain", DoubleValue (TxGain) );                                      //
original value was 4.5
  wifiPhy.Set ("RxGain", DoubleValue (RxGain) );                                      //
original value was 4.5
```

```
  wifiPhy.Set ("TxPowerLevels", UintegerValue (TxPowerLevels) );                      //
attributes are actually from
  wifiPhy.Set ("TxPowerEnd", DoubleValue (TxPowerEnd) );                              //
YansWifiPhy class
  wifiPhy.Set ("TxPowerStart", DoubleValue (TxPowerStart) );                          // has
full list of attributes
  wifiPhy.Set ("RxNoiseFigure", DoubleValue (RxNoiseFigure) );

  NqosWifiMacHelper wifiMac;              // Non quality of service mac helper
  wifiMac.Default();                      // with default attributes
  wifiMac.SetType ("ns3::AdhocWifiMac");  // We call the create message to create the MAC
(later)

  YansWifiChannelHelper wifiChannel;
  wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");
  //wifiChannel.AddPropagationLoss ("ns3::FriisPropagationLossModel"); // we can create
channel later
  wifiChannel.AddPropagationLoss ("ns3::LogDistancePropagationLossModel"); // we can create
channel later

  wifiPhy.Set("ChannelNumber", UintegerValue(1));
  wifiPhy.SetChannel (wifiChannel.Create ());

  WifiHelper wifi;
  wifi = WifiHelper::Default();
  wifi.SetStandard (WIFI_PHY_STANDARD_80211b);  // change the standard to 802.11b

  aodvDevices = wifi.Install (wifiPhy, wifiMac, aodvNodes);
  //cout << "SystemLoss is: " << wifiChannel.GetSystemLoss();
  Print("CreateAodvDevices",2);
}

void
CogMan::CreateAodvStack()
{
  Print("CreateAodvStack",1);
  aodvStack.SetRoutingHelper (aodv); // has effect on the next Install ()
  aodvStack.Install (aodvNodes);
  Print("CreateAodvStack",2);
}

void
CogMan::CreateAodvAddresses()
{
  Print("CreateAodvAddresses",1);
  Ipv4Address IP_Start = "10.2.0.0";
  Ipv4Mask IP_Subnet = "255.255.0.0";
  aodvAddresses.SetBase (IP_Start, IP_Subnet);
  message << "IP Start Address: " << IP_Start << endl;
  message << "IP Subnet: " << IP_Subnet;
  Print (message.str(), 9);

  Print("CreateAodvAddresses",2);
}

void
CogMan::CreateAodvInterface()
{
  Print("CreateAodvInterface",1);
  aodvInterface = aodvAddresses.Assign (aodvDevices);
  Print("CreateAodvInterface",2);
}

void
CogMan::CreateDsdvNodes(int n)
{
  Print("CreateDsdvNodes",1);
  dsdvNodes.Create(n);
  message << "Created " << n << " nodes in DSDV Container";
  Print(message.str(),18);

  CreateDsdvDevices();
  CreateDsdvStack();
  CreateDsdvAddresses();
  CreateDsdvInterface();
  //SetBatteries();
  if (nodeInfo)
  {
    for (NodeContainer::Iterator n = dsdvNodes.Begin(); n < dsdvNodes.End(); n++)
    {
      PrintNodeInformation((*n));
    }
  }
  Print("CreateDsdvNodes",2);
}

void
CogMan::CreateDsdvDevices()
{
  Print("CreateDsdvDevices",1);
  wifiPhy = YansWifiPhyHelper::Default();     /* set to default working state */

  wifiPhy.Set ("EnergyDetectionThreshold", DoubleValue (EnergyDetectionThreshold));   //
original value was -96.0
  wifiPhy.Set ("CcaMode1Threshold", DoubleValue (CcaMode1Threshold) );                //
original value was -99.0
  wifiPhy.Set ("TxGain", DoubleValue (TxGain) );                                      //
original value was 4.5
  wifiPhy.Set ("RxGain", DoubleValue (RxGain) );                                      //
original value was 4.5
  wifiPhy.Set ("TxPowerLevels", UintegerValue (TxPowerLevels) );                      //
attributes are actually from
  wifiPhy.Set ("TxPowerEnd", DoubleValue (TxPowerEnd) );                              //
YansWifiPhy class
  wifiPhy.Set ("TxPowerStart", DoubleValue (TxPowerStart) );                          // has
full list of attributes
  wifiPhy.Set ("RxNoiseFigure", DoubleValue (RxNoiseFigure) );

  NqosWifiMacHelper wifiMac;              // Non quality of service mac helper
  wifiMac.Default();                      // with default attributes
  wifiMac.SetType ("ns3::AdhocWifiMac");  // We call the create message to create the MAC
(later)

  YansWifiChannelHelper wifiChannel;
  wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");
  //wifiChannel.AddPropagationLoss ("ns3::FriisPropagationLossModel"); // we can create
channel later
  wifiChannel.AddPropagationLoss ("ns3::LogDistancePropagationLossModel"); // we can create
channel later

  wifiPhy.Set("ChannelNumber", UintegerValue(4));// used to be 4
  wifiPhy.SetChannel (wifiChannel.Create ());

  WifiHelper wifi;
  wifi = WifiHelper::Default();
  wifi.SetStandard (WIFI_PHY_STANDARD_80211b);  // change the standard to 802.11b

  dsdvDevices = wifi.Install (wifiPhy, wifiMac, dsdvNodes);

  Print("CreateDsdvDevices",2);
}

void
CogMan::CreateDsdvStack()
{
  Print("CreateDsdvStack",1);
  dsdvStack.SetRoutingHelper (dsdv); // has effect on the next Install ()
  dsdvStack.Install (dsdvNodes);
  Print("CreateDsdvStack",2);
}

void
CogMan::CreateDsdvAddresses()
{
  Print("CreateDsdvAddresses",1);
  Ipv4Address IP_Start = "10.1.0.0";
  Ipv4Mask IP_Subnet = "255.255.0.0";
  dsdvAddresses.SetBase (IP_Start, IP_Subnet);
  message << "IP Start Address: " << IP_Start << endl;
  message << "IP Subnet: " << IP_Subnet;
  Print (message.str(), 9);
  Print("CreateDsdvAddresses",2);
}
```

```
void
CogMan::CreateDsdvInterface()
{
  Print("CreateDsdvInterface",1);
  dsdvInterface = dsdvAddresses.Assign (dsdvDevices);
  Print("CreateDsdvInterface",2);
}

void
CogMan::CreateControlledAodvNodes(int n)
{
  Print("CreateControlledAodvNodes",1);
  aodvControlledNodes.Create(n);
  message << "Created " << n << " nodes in AODV Container";
  Print(message.str(),18);

  CreateControlledAodvDevices();
  CreateControlledAodvStack();
  CreateControlledAodvAddresses();
  CreateControlledAodvInterface();
  PlaceNodesOneLocation(aodvControlledNodes);
  InstallConstantMobility(aodvControlledNodes);
  SetBatteries();

  if (nodeInfo)
  {
    for (NodeContainer::Iterator n = aodvControlledNodes.Begin(); n <
aodvControlledNodes.End(); n++)
    {
      PrintNodeInformation((*n));
    }
  }
  Print("CreateControlledAodvNodes",2);
}

void
CogMan::CreateControlledAodvDevices()
{
  Print("CreateControlledAodvDevices",1);
  wifiPhy = YansWifiPhyHelper::Default();     /* set to default working state */

  wifiPhy.Set ("EnergyDetectionThreshold", DoubleValue (EnergyDetectionThreshold));   //
original value was -96.0
  wifiPhy.Set ("CcaMode1Threshold", DoubleValue (CcaMode1Threshold) );                //
original value was -99.0
  wifiPhy.Set ("TxGain", DoubleValue (TxGain) );                                      //
original value was 4.5
  wifiPhy.Set ("RxGain", DoubleValue (RxGain) );                                      //
original value was 4.5
  wifiPhy.Set ("TxPowerLevels", UintegerValue (TxPowerLevels) );                      //
attributes are actually from
  wifiPhy.Set ("TxPowerEnd", DoubleValue (TxPowerEnd) );                              //
YansWifiPhy class
  wifiPhy.Set ("TxPowerStart", DoubleValue (TxPowerStart) );                          // has
full list of attributes
  wifiPhy.Set ("RxNoiseFigure", DoubleValue (RxNoiseFigure) );

  NqosWifiMacHelper wifiMac;              // Non quality of service mac helper
  wifiMac.Default();                      // with default attributes
  wifiMac.SetType ("ns3::AdhocWifiMac");  // We call the create message to create the MAC
(later)

  YansWifiChannelHelper wifiChannel;
  wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");
  //wifiChannel.AddPropagationLoss ("ns3::FriisPropagationLossModel"); // we can create
channel later
  wifiChannel.AddPropagationLoss ("ns3::LogDistancePropagationLossModel"); // we can create
channel later



  wifiPhy.Set("ChannelNumber", UintegerValue(1));
  wifiPhy.SetChannel (wifiChannel.Create ());

  WifiHelper wifi;
  wifi = WifiHelper::Default();
  wifi.SetStandard (WIFI_PHY_STANDARD_80211b);  // change the standard to 802.11b

  aodvControlledDevices = wifi.Install (wifiPhy, wifiMac, aodvControlledNodes);


  //cout << "SystemLoss is: " << wifiChannel.GetSystemLoss();


  Print("CreateControlledAodvDevices",2);
}

void
CogMan::CreateControlledAodvStack()
{
  Print("CreateAodvStack",1);
  aodvControlledStack.SetRoutingHelper (aodv); // has effect on the next Install ()
  aodvControlledStack.Install (aodvControlledNodes);
  Print("CreateAodvStack",2);
}

void
CogMan::CreateControlledAodvAddresses()
{
  Print("CreateAodvAddresses",1);
  // Ipv4Address IP_Start = "10.2.0.0";
  // Ipv4Mask IP_Subnet = "255.255.0.0";
  // aodvAddresses.SetBase (IP_Start, IP_Subnet);
  // message << "IP Start Address: " << IP_Start << endl;
  // message << "IP Subnet: " << IP_Subnet;
  Print (message.str(), 9);

  Print("CreateAodvAddresses",2);
}

void
CogMan::CreateControlledAodvInterface()
{
  Print("CreateAodvInterface",1);
  aodvControlledInterface = aodvAddresses.Assign (aodvControlledDevices);
  Print("CreateAodvInterface",2);
}

void
CogMan::CreateControlledDsdvNodes(int n)
{
  Print("CreateDsdvNodes",1);
  dsdvControlledNodes.Create(n);
  message << "Created " << n << " nodes in DSDV Container";
  Print(message.str(),18);

  CreateControlledDsdvDevices();
  CreateControlledDsdvStack();
  CreateControlledDsdvAddresses();
  CreateControlledDsdvInterface();
  PlaceNodesOneLocation(dsdvControlledNodes);
  InstallConstantMobility(dsdvControlledNodes);

  if (nodeInfo)
  {
    for (NodeContainer::Iterator n = dsdvControlledNodes.Begin(); n <
dsdvControlledNodes.End(); n++)
    {
      PrintNodeInformation((*n));
    }
  }
  Print("CreateDsdvNodes",2);
}

void
CogMan::CreateControlledDsdvDevices()
{
  Print("CreateDsdvDevices",1);
  wifiPhy = YansWifiPhyHelper::Default();     /* set to default working state */

  wifiPhy.Set ("EnergyDetectionThreshold", DoubleValue (EnergyDetectionThreshold));
  wifiPhy.Set ("CcaMode1Threshold", DoubleValue (CcaMode1Threshold) );
  wifiPhy.Set ("TxGain", DoubleValue (TxGain) );
  wifiPhy.Set ("RxGain", DoubleValue (RxGain) );
  wifiPhy.Set ("TxPowerLevels", UintegerValue (TxPowerLevels) );
  wifiPhy.Set ("TxPowerEnd", DoubleValue (TxPowerEnd) );
  wifiPhy.Set ("TxPowerStart", DoubleValue (TxPowerStart) );
  wifiPhy.Set ("RxNoiseFigure", DoubleValue (RxNoiseFigure) );

  NqosWifiMacHelper wifiMac;              // Non quality of service mac helper
  wifiMac.Default();                      // with default attributes
  wifiMac.SetType ("ns3::AdhocWifiMac");  // We call the create message to create the MAC
(later)

  YansWifiChannelHelper wifiChannel;
  wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");
  //wifiChannel.AddPropagationLoss ("ns3::FriisPropagationLossModel"); // we can create
channel later
  wifiChannel.AddPropagationLoss ("ns3::LogDistancePropagationLossModel"); // we can create
channel later

  wifiPhy.Set("ChannelNumber", UintegerValue(4));// used to be 4
  wifiPhy.SetChannel (wifiChannel.Create ());

  WifiHelper wifi;
  wifi = WifiHelper::Default();
  wifi.SetStandard (WIFI_PHY_STANDARD_80211b);  // change the standard to 802.11b
  dsdvControlledDevices = wifi.Install (wifiPhy, wifiMac, dsdvControlledNodes);
  Print("CreateDsdvDevices",2);
}

void
CogMan::CreateControlledDsdvStack()
{
  Print("CreateDsdvStack",1);
  dsdvControlledStack.SetRoutingHelper (dsdv); // has effect on the next Install ()
  dsdvControlledStack.Install (dsdvControlledNodes);
  Print("CreateDsdvStack",2);
}

void
CogMan::CreateControlledDsdvAddresses()
{
  //Print("CreateDsdvAddresses",1);
  //Ipv4Address IP_Start = "10.1.0.0";
  //Ipv4Mask IP_Subnet = "255.255.0.0";
  //dsdvAddresses.SetBase (IP_Start, IP_Subnet);
  //message << "IP Start Address: " << IP_Start << endl;
  //message << "IP Subnet: " << IP_Subnet;
  Print (message.str(), 9);

  Print("CreateDsdvAddresses",2);
}

void
CogMan::CreateControlledDsdvInterface()
{
  Print("CreateDsdvInterface",1);
  dsdvControlledInterface = dsdvAddresses.Assign (dsdvControlledDevices);
  Print("CreateDsdvInterface",2);
}

void
CogMan::CreateCongestionInterface()
{
  Print("CreateCongestionInterface",1);
  congestionInterface = congestionAddresses.Assign (congestionDevices);
  Print("CreateCongestionInterface",2);
}

void
CogMan::CreateCongestionNodes(int n)
{
  Print("CreateCongestionNodes",1);
  congestionNodes.Create(n);
  message << "Created " << n << " nodes in Congestion Container";
  Print(message.str(),18);

  CreateCongestionDevices();
  CreateCongestionStack();
  CreateCongestionAddresses();
  CreateCongestionInterface();
  PlaceNodesOneLocation(congestionNodes);
  InstallConstantMobility(congestionNodes);

  if (nodeInfo)
  {
    for (NodeContainer::Iterator n = congestionNodes.Begin(); n < congestionNodes.End(); n++)
    {
      PrintNodeInformation((*n));
    }
  }
  Print("CreateCongestionNodes",2);
}

void
CogMan::CreateCongestionDevices()
{
  Print("CreateCongestionDevices",1);

  std::string phyMode ("DsssRate1Mbps");
  double Prss = -80;  // -dBm
  double Irss = -95;  // -dBm
  double delta = 0;  // microseconds
  uint32_t PpacketSize = 1000; // bytes
  uint32_t IpacketSize = 1000; // bytes
  bool verbose = false;

  // these are not command line arguments for this version
  uint32_t numPackets = 100;
  double interval = 1.0; // seconds
  double startTime = 10.0; // seconds
  double distanceToRx = 100.0; // meters

  double offset = 91;  // This is a magic number used to set the
                       // transmit power, based on other configuration



  // Convert to time object
  Time interPacketInterval = Seconds (interval);

  // disable fragmentation for frames below 2200 bytes
  Config::SetDefault ("ns3::WifiRemoteStationManager::FragmentationThreshold", StringValue
("2200"));
  // turn off RTS/CTS for frames below 2200 bytes
  Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThreshold", StringValue ("2200"));
  // Fix non-unicast data rate to be the same as that of unicast
  Config::SetDefault ("ns3::WifiRemoteStationManager::NonUnicastMode",
                      StringValue (phyMode));


  // The below set of helpers will help us to put together the wifi NICs we want
  WifiHelper wifi;

  wifi.SetStandard (WIFI_PHY_STANDARD_80211b);

  YansWifiPhyHelper wifiPhy =  YansWifiPhyHelper::Default ();
  // set it to zero; otherwise, gain will be added
  wifiPhy.Set ("RxGain", DoubleValue (0) );
  wifiPhy.Set ("CcaMode1Threshold", DoubleValue (0.0) );

  // ns-3 supports RadioTap and Prism tracing extensions for 802.11b
  wifiPhy.SetPcapDataLinkType (YansWifiPhyHelper::DLT_IEEE802_11_RADIO);

  YansWifiChannelHelper wifiChannel;
  wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");
  wifiChannel.AddPropagationLoss ("ns3::LogDistancePropagationLossModel");

  wifiPhy.Set("ChannelNumber", UintegerValue(1));
  wifiPhy.SetChannel (wifiChannel.Create ());

  // Add a non-QoS upper mac, and disable rate control
  NqosWifiMacHelper wifiMac = NqosWifiMacHelper::Default ();
  wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
                                "DataMode",StringValue (phyMode),
                                "ControlMode",StringValue (phyMode));
  // Set it to adhoc mode
  wifiMac.SetType ("ns3::AdhocWifiMac");

  congestionDevices = wifi.Install (wifiPhy, wifiMac, congestionNodes);
  // This will disable these sending devices from detecting a signal
  // so that they do not backoff
  wifiPhy.Set ("EnergyDetectionThreshold", DoubleValue (0.0) );

  wifiPhy.Set ("TxGain", DoubleValue (offset + Irss) );
  congestionDevices.Add (wifi.Install (wifiPhy, wifiMac, congestionNodes));



  /*

  TypeId tid = TypeId::LookupByName ("ns3::UdpSocketFactory");
  Ptr<Socket> recvSink = Socket::CreateSocket (c.Get (0), tid);
  InetSocketAddress local = InetSocketAddress (Ipv4Address ("10.1.1.1"), 80);
  recvSink->Bind (local);
  recvSink->SetRecvCallback (MakeCallback (&ReceivePacket));

  Ptr<Socket> source = Socket::CreateSocket (c.Get (1), tid);
  InetSocketAddress remote = InetSocketAddress (Ipv4Address ("255.255.255.255"), 80);
  source->SetAllowBroadcast (true);
  source->Connect (remote);

  // Interferer will send to a different port; we will not see a
  // "Received packet" message
  Ptr<Socket> interferer = Socket::CreateSocket (c.Get (2), tid);
  InetSocketAddress interferingAddr = InetSocketAddress (Ipv4Address ("255.255.255.255"),
49000);
  interferer->SetAllowBroadcast (true);
  interferer->Connect (interferingAddr);

  // Tracing
  wifiPhy.EnablePcap ("wifi-simple-interference", devices.Get (0));

  // Output what we are doing
  NS_LOG_UNCOND ("Primary packet RSS=" << Prss << " dBm and interferer RSS=" << Irss << " dBm
at time offset=" << delta << " ms");

  Simulator::ScheduleWithContext (source->GetNode ()->GetId (),
                                  Seconds (startTime), &GenerateTraffic,
                                  source, PpacketSize, numPackets, interPacketInterval);

  Simulator::ScheduleWithContext (interferer->GetNode ()->GetId (),
                                  Seconds (startTime ), &GenerateTraffic,
                                  interferer, IpacketSize, numPackets, interPacketInterval);

  */

  /*
  wifiPhy = YansWifiPhyHelper::Default();     // set to default working state

  wifiPhy.Set ("EnergyDetectionThreshold", DoubleValue (0));      // original value was -96.0
  wifiPhy.Set ("CcaMode1Threshold", DoubleValue (CcaMode1Threshold) );                 //
original value was -99.0
  wifiPhy.Set ("TxGain", DoubleValue (TxGain) );                                       //
original value was 4.5
  wifiPhy.Set ("RxGain", DoubleValue (RxGain) );                                       //
original value was 4.5
  wifiPhy.Set ("TxPowerLevels", UintegerValue (TxPowerLevels) );                       //
attributes are actually from
  wifiPhy.Set ("TxPowerEnd", DoubleValue (TxPowerEnd) );                               //
YansWifiPhy class
  wifiPhy.Set ("TxPowerStart", DoubleValue (TxPowerStart) );                           // has
full list of attributes
  wifiPhy.Set ("RxNoiseFigure", DoubleValue (RxNoiseFigure) );

  NqosWifiMacHelper wifiMac;              // Non quality of service mac helper
  wifiMac.Default();                      // with default attributes
  wifiMac.SetType ("ns3::AdhocWifiMac");  // We call the create message to create the MAC
(later)

  YansWifiChannelHelper wifiChannel;
  wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");
  //wifiChannel.AddPropagationLoss ("ns3::FriisPropagationLossModel"); // we can create
channel later
  wifiChannel.AddPropagationLoss ("ns3::LogDistancePropagationLossModel"); // we can create
channel later

  wifiPhy.Set("ChannelNumber", UintegerValue(1));
  wifiPhy.SetChannel (wifiChannel.Create ());

  WifiHelper wifi;
  wifi = WifiHelper::Default();
  wifi.SetStandard (WIFI_PHY_STANDARD_80211b);  // change the standard to 802.11b

  congestionDevices = wifi.Install (wifiPhy, wifiMac, congestionNodes);


  //cout << "SystemLoss is: " << wifiChannel.GetSystemLoss();
  */

  Print("CreateCongestionDevices",2);
}

void
CogMan::CreateCongestionStack()
{
  Print("CreateCongestionStack",1);
  congestionStack.SetRoutingHelper (aodv); // has effect on the next Install ()
  congestionStack.Install (congestionNodes);
  Print("CreateCongestionStack",2);
}

void
CogMan::CreateCongestionAddresses()
{
  Print("CreateCongestionAddresses",1);
  Ipv4Address IP_Start = "10.5.0.0";
  Ipv4Mask IP_Subnet = "255.255.0.0";
  congestionAddresses.SetBase (IP_Start, IP_Subnet);
  message << "IP Start Address: " << IP_Start << endl;
  message << "IP Subnet: " << IP_Subnet;
  Print (message.str(), 9);

  Print("CreateCongestionAddresses",2);
}

// Node Grouping

void
CogMan::SplitNodesTwoEvenGroups()
{
  // we have given groups names
  // group 1 is Eagles
  // group 2 is Sharks

  cout << "\nTotal number of nodes is: " << numOfNodes;

  int nodeCounter = 0;
  while (nodeCounter < numOfNodes)
  {
    AodvEaglesNode.push_back ({ nodeCounter });
    DsdvEaglesNode.push_back ({ nodeCounter });
    //cout << "\nI've pushed back node " << nodeCounter;
    if (nodeCounter+1 < numOfNodes)
    {
      AodvSharksNode.push_back ({ nodeCounter+1 });
      DsdvSharksNode.push_back ({ nodeCounter+1 });
      //cout << "\nI've pushed back node " << nodeCounter+1;
    }
    nodeCounter = nodeCounter + 2;
  }
}

// Placement Strategies

void
CogMan::PlaceNodesTwoGroups()
{
  Print("PlaceNodesTwoGroups",1);
  Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator> ();
  for (int n=0; n<floor(aodvNodes.GetN()/3); n++)
  {
    // approximately -40..25 meters
    positionAlloc->Add (Vector (0.0+rand()*0.00000003-40, 0.0+rand()*0.00000003-40, 0.0));
    mobility.SetPositionAllocator (positionAlloc);
    AodvEaglesNode.push_back ({ n });
    DsdvEaglesNode.push_back ({ n });
  }

  for (int n=floor(aodvNodes.GetN()/3); n<aodvNodes.GetN(); n++)
  {
    positionAlloc->Add (Vector (130.0+rand()*0.00000003-40, 0.0+rand()*0.00000003-40, 0.0));
    mobility.SetPositionAllocator (positionAlloc);
    AodvSharksNode.push_back ({ n });
    DsdvSharksNode.push_back ({ n });
  }
  Print("PlaceNodesOneLocation",2);
}

void
CogMan::PlaceNodesRandomLocation(NodeContainer nodes)
{
  Print("PlaceNodesRandomLocation",1);
  Ptr<ListPositionAllocator> positionAlloc = CreateObject <ListPositionAllocator>();

  for (int i=0; i<nodes.GetN(); i++)
  {
  double xRandom=rand()*0.00000007-4;
  double yRandom=rand()*0.00000007-4;
  positionAlloc ->Add(Vector(xRandom, yRandom, 0)); // x,y,z
  }

  mobility.SetPositionAllocator(positionAlloc);
  Print("PlaceNodesRandomLocation",2);
}

void
CogMan::PlaceNodesOneLocation()
{
  Print("PlaceNodesOneLocation",1);
  Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator> ();
  for (int n=0; n<allNodes.GetN(); n++)
    positionAlloc->Add (Vector (0.0, 120.0, 0.0));
  mobility.SetPositionAllocator (positionAlloc);
  Print("PlaceNodesOneLocation",2);
}

void
CogMan::PlaceNodesOneLocation(NodeContainer nodes)
{
  Print("PlaceNodesOneLocation",1);
  Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator> ();
  for (int n=0; n<nodes.GetN(); n++)
    positionAlloc->Add (Vector (100.0, 0.0, 0.0));
  mobility.SetPositionAllocator (positionAlloc);
  Print("PlaceNodesOneLocation",2);
}

void
CogMan::PlaceNodesTopLCRMiddleLCRBottomLCM()
{
  Print("PlaceNodesTopLCRMiddleLCRBottomLCM",1);
  Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator> ();
  positionAlloc->Add (Vector (0.0, 0.0, 0.0));
  positionAlloc->Add (Vector (0.0, 150.0, 0.0));
  positionAlloc->Add (Vector (0.0, 300.0, 0.0));
  positionAlloc->Add (Vector (150.0, 0.0, 0.0));
  positionAlloc->Add (Vector (150.0, 150.0, 0.0));
  positionAlloc->Add (Vector (150.0, 300.0, 0.0));
  positionAlloc->Add (Vector (300.0, 0.0, 0.0));
  positionAlloc->Add (Vector (300.0, 150.0, 0.0));
  positionAlloc->Add (Vector (300.0, 300.0, 0.0));

  mobility.SetPositionAllocator (positionAlloc);
  Print("PlaceNodesTopLCRMiddleLCRBottomLCM",2);
}
```

```
void
CogMan::DsdvPositionMatchAodv()
{
  for (int n=0; n<numOfNodes; n++)
  {
    Vector position = GetNodePosition(n);
    MoveNodeTo (n+numOfNodes, position.x, position.y);
  }
}

void
CogMan::AodvPositionMatchDsdv()
{
  for (int n=0; n<numOfNodes; n++)
  {
    Vector position = GetNodePosition(n+numOfNodes);
    MoveNodeTo (n, position.x, position.y);
  }
}

// MOBILITY MODELS

void
CogMan::InstallConstantMobility()
{
  Print("InstallConstantMobility",1);
  mobility.SetMobilityModel("ns3::ConstantPositionMobilityModel");
  mobility.Install(allNodes);
  Print("InstallConstantMobility",2);
}

void
CogMan::InstallConstantMobility(NodeContainer nodes)
{
  Print("InstallConstantMobility",1);
  mobility.SetMobilityModel("ns3::ConstantPositionMobilityModel");
  mobility.Install(nodes);
  Print("InstallConstantMobility",2);
}

void
CogMan::MoveControlledNodeTo (int nodeId, double x, double y)
{
  Print("MoveNodeTo",1);

  Ptr<MobilityModel> mobility;
  Vector pos;

  mobility = allControlledNodes.Get(nodeId)->GetObject<MobilityModel> ();
  pos = mobility->GetPosition();
  pos.x = x;
  pos.y = y;
  mobility->SetPosition(pos);

  message << "At time "<< GetTimeNow('s') <<
  "s \tNode " << nodeId << "\tx: " << pos.x <<
  "\ty: " << pos.y ;
  cout << message.str();
  Print(message.str(), 12);
  Print("MoveNodeTo",2);
}

void
CogMan::MoveNodeTo (int nodeId, double x, double y)
{
  Print("MoveNodeTo",1);

  if (allNodes.GetN() > 0)
  {
    Ptr<MobilityModel> mobility;
    Vector pos;

    mobility = allNodes.Get(nodeId)->GetObject<MobilityModel> ();
    pos = mobility->GetPosition();
    pos.x = x;
    pos.y = y;
    mobility->SetPosition(pos);

    message << "At time "<< GetTimeNow('s') <<
    "s \tNode " << nodeId << "\tx: " << pos.x <<
    "\ty: " << pos.y ;

    Print(message.str(), 12);
  }
  else
  {
    cout << "\nFATAL ERROR - allNodes container is empty";
  }
  Print("MoveNodeTo",2);
}

void
CogMan::ArchimedeanSpiral(int nodeId)
{
  Print("ArchimedeanSpiral",1);
  for (double myTime=0 ; myTime < simulationTime ; myTime+=0.05)
  {
  Vector currentPosition = GetNodePosition(nodeId);
  double xGoal = currentPosition.x+myTime/(2*3.14159265359)*cos(myTime);
  double yGoal = currentPosition.y+myTime/(2*3.14159265359)*sin(myTime);

  Simulator::Schedule (Seconds (myTime), &CogMan::MoveNodeTo, this, nodeId, xGoal, yGoal);
  }
  Print("ArchimedeanSpiral",2);
}

void
CogMan::ArchimedeanSpiralClockwise(int nodeId)
{
  Print("ArchimedeanSpiralClockwise",1);
  pow(-1,nodeId);
  for (double myTime=0 ; myTime < simulationTime ; myTime+=0.05)
  {
  Vector currentPosition = GetNodePosition(nodeId);
  double xGoal = currentPosition.x+myTime/(2*3.14159265359)*cos((-1)*myTime);
  double yGoal = currentPosition.y+myTime/(2*3.14159265359)*sin((-1)*myTime);

  Simulator::Schedule (Seconds (myTime), &CogMan::MoveNodeTo, this, nodeId, xGoal, yGoal);
  }
  Print("ArchimedeanSpiralClockwise",2);
}

void
CogMan::ControlledNodeWalkTo (int nodeId, double xGoal, double yGoal, double stepSize, double
speed)
{
  Print("NodeWalkTo",1);
  // First we check such node exists!
  NS_ASSERT_MSG (nodeId < aodvControlledNodes.GetN(), "There is no such node to get position
for.");

  double myTime = GetTimeNow('s');

  // We are working with x2 node containers.  With the aim to mirror the mobility.
  // Therefore we need to get the corresponding node in the other container.
  int nodeId2 = nodeId + aodvControlledNodes.GetN();

  if (!IsControlledNodeMoving(nodeId, xGoal, yGoal, stepSize, speed))
  {
    Vector currentPosition = GetControlledNodePosition(nodeId);
    Vector goalPosition;
      goalPosition.x = xGoal+rand()*0.00000001-10;
      goalPosition.y = yGoal+rand()*0.00000001-10;

    double distance      = GetDistance(currentPosition, goalPosition);
    double duration      = distance / (speed);//+(rand()*0.00000000006-0.06));
    double numberOfSteps = distance / stepSize;
    double timeEachStep  = duration / numberOfSteps;
    double xStep         = (xGoal - currentPosition.x) / numberOfSteps;
    double yStep         = (yGoal - currentPosition.y) / numberOfSteps;

    RecordControlledNodeMovement(nodeId, duration + myTime);
    message << "\nNode " << nodeId << "\n------------"
            << "\ncurrent x is " << currentPosition.x << " current y is " << currentPosition.y
            << "\ndesitnation x is " << xGoal << " destination y is " << yGoal
            << "\ndistance is " << distance
            << "\nNode " << nodeId << " will arrive at destination in: " << duration << "s"
            << "\nand will have to take: " << numberOfSteps << " steps."
            << "\nEach step will take: " << timeEachStep << "s"
            << "\nI will arrive at the destination at " << duration + myTime;
    Print (message.str(),12);

    for (int n=0; n<numberOfSteps; n++)
    {
      currentPosition.x = currentPosition.x + xStep;
      currentPosition.y = currentPosition.y + yStep;

      message << "\nStep " << n << " x " << currentPosition.x << " y " << currentPosition.y;
      Print (message.str(),12);

      EventId id = Simulator::Schedule (Seconds (n*timeEachStep),
        &CogMan::MoveControlledNodeTo, this, nodeId, currentPosition.x, currentPosition.y);
      Simulator::Schedule (Seconds (n*timeEachStep),
        &CogMan::MoveControlledNodeTo, this, nodeId2, currentPosition.x, currentPosition.y);

      //message << "The scheduled event ID is: " << id.GetUid();
      //Print (message.str(),12);
    }
    message << "\nI ended up at x " << currentPosition.x << " y " << currentPosition.y <<
"\n";
    Print (message.str(),12);
  }
  Print("NodeWalkTo", 2);
}

void
CogMan::NodeWalkTo (int nodeId, double xGoal, double yGoal, double stepSize, double speed)
{
  Print("NodeWalkTo",1);
  // First we check such node exists!
  NS_ASSERT_MSG (nodeId < aodvNodes.GetN(), "There is no such node to get position for.");

  double myTime = GetTimeNow('s');

  // We are working with x2 node containers.  With the aim to mirror the mobility.
  // Therefore we need to get the corresponding node in the other container.
  int nodeId2 = nodeId + aodvNodes.GetN();

  if (!IsNodeMoving(nodeId, xGoal, yGoal, stepSize, speed))
  {
    Vector currentPosition = GetNodePosition(nodeId);
    Vector goalPosition;
      goalPosition.x = xGoal; //+rand()*0.00000001-10;
      goalPosition.y = yGoal; //+rand()*0.00000001-10;

    double distance      = GetDistance(currentPosition, goalPosition);
    double duration      = distance / (speed);//+(rand()*0.00000000006-0.06));
    double numberOfSteps = distance / stepSize;
    double timeEachStep  = duration / numberOfSteps;
    double xStep         = (xGoal - currentPosition.x) / numberOfSteps;
    double yStep         = (yGoal - currentPosition.y) / numberOfSteps;

    RecordMovement(nodeId, duration + myTime);

    message << "\nNode " << nodeId << "\n------------"
            << "\ncurrent x is " << currentPosition.x << " current y is " << currentPosition.y
            << "\ndesitnation x is " << xGoal << " destination y is " << yGoal
            << "\ndistance is " << distance
            << "\nNode " << nodeId << " will arrive at destination in: " << duration << "s"
            << "\nand will have to take: " << numberOfSteps << " steps."
            << "\nEach step will take: " << timeEachStep << "s"
            << "\nI will arrive at the destination at " << duration + myTime;
    Print (message.str(),12);

    for (int n=0; n<numberOfSteps; n++)
    {
      currentPosition.x = currentPosition.x + xStep;
      currentPosition.y = currentPosition.y + yStep;
      message << "\nStep " << n << " x " << currentPosition.x << " y " << currentPosition.y;
      Print (message.str(),12);

      EventId id = Simulator::Schedule (Seconds (n*timeEachStep),
        &CogMan::MoveNodeTo, this, nodeId, currentPosition.x, currentPosition.y);
      Simulator::Schedule (Seconds (n*timeEachStep),
        &CogMan::MoveNodeTo, this, nodeId2, currentPosition.x, currentPosition.y);

      //message << "The scheduled event ID is: " << id.GetUid();
      //Print (message.str(),12);
    }
    message << "\nI ended up at x " << currentPosition.x << " y " << currentPosition.y << 
"\n";
    Print (message.str(),12);
  }
  Print("NodeWalkTo", 2);
}

void
CogMan::MobilityRandomWayPoint(double speed)
{
  Print("MobilityRandomWaypointStuff",1);

  mobility2.SetPositionAllocator ("ns3::RandomDiscPositionAllocator",
                                  "X", StringValue ("100.0"),
                                  "Y", StringValue ("100.0"),
                                  "Rho", StringValue 
("ns3::UniformRandomVariable[Min=0|Max=30]"));

  std::ostringstream oss;
  oss << "Constant:"<< speed;
  std::string speedValue = oss.str();

  NS_LOG_INFO ("Installing velocity: speed: " << speedValue << " .");

  mobility2.SetMobilityModel ("ns3::RandomWalk2dMobilityModel",
                              "Mode", StringValue ("Time"),
                              "Time", StringValue ("10s"),
                              "Speed", StringValue 
("ns3::ConstantRandomVariable[Constant=1.0]"),
                              "Bounds", StringValue ("0|1000|0|1000"),
                              "Distance", DoubleValue (40.0)
                              );
  mobility2.Install (aodvNodes);
  Print("MobilityRandomWaypointStuff",2);
}

void
CogMan::MoveNodeDiamondPattern()
{
  Print("MoveNodeDiamondPattern",1);
  vector<controlledNode>::iterator vectorIndex;
  vector<controlledNode>::iterator vectorEnd;
  vector<controlledNode>::iterator vectorIndex2;
  vector<controlledNode>::iterator vectorEnd2;
  double scheduledTime = Simulator::Now().GetSeconds();
  double timeOfJourney;
  double x=100;
  double x2;
  double y=0;

  cout << "\nScheduled Time is: " << scheduledTime;

  while (scheduledTime < simulationTime)
  {
    vectorIndex = AodvEaglesNode.begin();
    vectorEnd = AodvEaglesNode.end();

    vectorIndex2 = AodvSharksNode.begin();
    vectorEnd2 = AodvSharksNode.end();

    if (x == 100)
    {
      x = 0;
      x2 = 200;
    }
    else
    {
      x = 100;
      x2 = 100;
    }

    y = y + 50;

    while (vectorIndex < vectorEnd)
    {                                                                                 // NodeId,
X, Y, Stride, Speed.
      Simulator::Schedule (Seconds (scheduledTime), &CogMan::NodeWalkTo, this, vectorIndex-
>nodeId,     x, y, 0.8, 1.48);
      vectorIndex++;
    }

    while (vectorIndex2 < vectorEnd2)
    {                                                                                 // NodeId,
X, Y, Stride, Speed.
      Simulator::Schedule (Seconds (scheduledTime), &CogMan::NodeWalkTo, this, vectorIndex2-
>nodeId,     x2, y, 0.8, 1.48);
      vectorIndex2++;
    }

    timeOfJourney = GetTimeOfJourney(vectorIndex->nodeId, x, y, 1.48);
    scheduledTime = scheduledTime + timeOfJourney;

    vectorIndex = AodvEaglesNode.begin();
    vectorEnd = AodvEaglesNode.end();

    vectorIndex2 = AodvSharksNode.begin();
    vectorEnd2 = AodvSharksNode.end();

    if (x == 100)
    {
      x = 0;
      x2 = 200;
    }
    else
    {
      x = 100;
      x2 = 100;
    }

    y = y + 50;

    while (vectorIndex < vectorEnd)
    {                                                                                 // NodeId,
X, Y, Stride, Speed.
      Simulator::Schedule (Seconds (scheduledTime), &CogMan::NodeWalkTo, this, vectorIndex-
>nodeId,   x,    y,    0.8, 1.48);
      vectorIndex++;
    }
    while (vectorIndex2 < vectorEnd2)
    {                                                                                 // NodeId,
X, Y, Stride, Speed.
      Simulator::Schedule (Seconds (scheduledTime), &CogMan::NodeWalkTo, this, vectorIndex2-
>nodeId,   x2,    y,    0.8, 1.48);
      vectorIndex2++;
    }
    timeOfJourney = GetTimeOfJourney(vectorIndex->nodeId, x, y, 1.48);
    scheduledTime = scheduledTime + timeOfJourney;
  }
  Print("MoveNodeDiamondPattern",2);
}

void
CogMan::LakeScenario()
{
  vector<controlledNode>::iterator vectorIndex;
  vector<controlledNode>::iterator vectorEnd;

  vectorIndex = AodvEaglesNode.begin();
  vectorEnd = AodvEaglesNode.end();

  double journeyTime = 0;
  double randomNum   = 0;
  double randomSpeed = 0;
  int x = 0;
  int y = 25;

  while (vectorIndex < vectorEnd)
  {
    randomSpeed = nodeSpeed + rand()*0.00000000006-0.06;
    Simulator::Schedule (Seconds (0.1), &CogMan::NodeWalkTo, this, vectorIndex->nodeId, x, y,
0.8, randomSpeed);
    journeyTime = GetTimeOfJourney(vectorIndex->nodeId, x, y, randomSpeed);

    while (journeyTime < simulationTime)
    {
      if (x < 20)
      {
        x = 100;
        randomNum = rand() % 10 + 1;
        x = x + randomNum - 5;
      }
      else
      {
        x = 0;
        randomNum = rand() % 10 + 1;
        x = x + randomNum - 5;
      }
      y = y + 25;
      randomNum = rand() % 5 + 1;
      y = y + randomNum - 2.5;
      randomSpeed = nodeSpeed + rand()*0.00000000006-0.06;
      Simulator::Schedule (Seconds (journeyTime), &CogMan::NodeWalkTo, this, vectorIndex-
>nodeId, x, y, 0.8, randomSpeed);
      journeyTime = journeyTime + GetTimeOfJourney(vectorIndex->nodeId, x, y, 1.48);
    }
    vectorIndex++;
    journeyTime = 0;
    x = 0;
    y = 25;
  }

  vectorIndex = AodvSharksNode.begin();
  vectorEnd = AodvSharksNode.end();
  journeyTime = 0;
  x = 200;
  y = 25;

  while (vectorIndex < vectorEnd)
  {
    randomSpeed = nodeSpeed + rand()*0.00000000006-0.06;
    Simulator::Schedule (Seconds (0.1),    &CogMan::NodeWalkTo, this, vectorIndex->nodeId, x,
y, 0.8, randomSpeed);
    journeyTime = GetTimeOfJourney(vectorIndex->nodeId, x, y, randomSpeed);

    while (journeyTime < simulationTime)
    {
      if (x < 120)
      {
        x = 200;
      }
      else
      {
        x = 100;
      }
      y = y + 25;
      randomSpeed = nodeSpeed + rand()*0.00000000006-0.06;
      Simulator::Schedule (Seconds (journeyTime),    &CogMan::NodeWalkTo, this, vectorIndex-
>nodeId,   x,    y,    0.8, randomSpeed);
      journeyTime = journeyTime + GetTimeOfJourney(vectorIndex->nodeId, x, y, randomSpeed);
    }
    vectorIndex++;
    journeyTime = 0;
    x = 200;
    y = 25;
  }
}

// MOBILITY MODELS SUPPORT METHODS

bool
```

```
CogMan::IsNodeMoving(int nodeId, int x, int y, double stride, double speed)
{
  vector<nodeMoving>::iterator vectorIndex;
  vector<nodeMoving>::iterator vectorEnd;
  vectorIndex = setNodeMove.begin();
  vectorEnd = setNodeMove.end();

  bool moving = false;
  double timeNow  = GetTimeNow('s');
  while (vectorIndex < vectorEnd)
  {
    if (vectorIndex->nodeId == nodeId)
    {
      if (vectorIndex->timeArrive > timeNow)
        {
          double inXSec = vectorIndex->timeArrive + 1 - timeNow;
          Simulator::Schedule (Seconds (inXSec), &CogMan::NodeWalkTo, this, nodeId, x, y, 
stride, speed);
          moving = true;
          break;
        }
    }
    vectorIndex++;
  }
  return moving;
}

double
CogMan::IsControlledNodeMoving(int nodeId)
{
  double timeNow = GetTimeNow('s');
  double inXSec = 0;
  // Then we check node is not already moving
  vector<nodeMoving>::iterator vectorIndex;
  vector<nodeMoving>::iterator vectorEnd;
  vectorIndex = setControlledNodeMove.begin();
  vectorEnd = setControlledNodeMove.end();

  while (vectorIndex < vectorEnd)
  {
    if (vectorIndex->nodeId == nodeId)
    {
      if (vectorIndex->timeArrive > timeNow)
      {
        inXSec = vectorIndex->timeArrive + 1 - timeNow;
        break;
      }
    }
    vectorIndex++;
  }
  return inXSec;
}

bool
CogMan::IsControlledNodeMoving(int nodeId, int x, int y, double stride, double speed)
{
  double timeNow = GetTimeNow('s');
  // Then we check node is not already moving
  vector<nodeMoving>::iterator vectorIndex;
  vector<nodeMoving>::iterator vectorEnd;
  vectorIndex = setControlledNodeMove.begin();
  vectorEnd = setControlledNodeMove.end();

  bool moving = false;

  while (vectorIndex < vectorEnd)
  {
    if (vectorIndex->nodeId == nodeId)
    {
      if (vectorIndex->timeArrive > timeNow)
        {
          double inXSec = vectorIndex->timeArrive + 1 - timeNow;
          Simulator::Schedule (Seconds (inXSec), &CogMan::ControlledNodeWalkTo, this, nodeId, 
x, y, stride, speed);
          moving = true;
          break;
        }
    }
    vectorIndex++;
  }
  return moving;
}

double
CogMan::GetTimeOfJourney (int nodeId, double xGoal, double yGoal, double speed)
{
  Print("NodeWalkTo",1);
  NS_ASSERT_MSG (nodeId < allNodes.GetN(), "There is no such node to get position for.");

  double myTime = GetTimeNow('s');

  Vector currentPosition = GetNodePosition(nodeId);
  Vector goalPosition;
    goalPosition.x = xGoal;
    goalPosition.y = yGoal;

  double duration  = GetDistance(currentPosition, goalPosition) / speed;
  message << "Node " << nodeId << " will arrive at destination in: " << duration << "s";
  Print (message.str(),12);
  return duration;
}

double
CogMan::GetTimeOfJourney (double x, double y, double xGoal, double yGoal, double speed)
{
  Print("NodeWalkTo",1);

  Vector currentPosition;
    currentPosition.x = x;
    currentPosition.y = y;
  Vector goalPosition;
    goalPosition.x = xGoal;
    goalPosition.y = yGoal;

  double duration  = GetDistance(currentPosition, goalPosition) / speed;
  message << "Journey will take " << duration << "s";
  Print (message.str(),12);
  return duration;
}

double
CogMan::GetTimeOfJourneyControlledNode (Vector currentPosition, double xGoal, double yGoal,
double speed)
{
  Print("NodeWalkTo",1);
  double myTime = GetTimeNow('s');

  Vector goalPosition;
    goalPosition.x = xGoal;
    goalPosition.y = yGoal;

  double duration  = GetDistance(currentPosition, goalPosition) / speed;
  message << "Node will arrive at destination in: " << duration << "s";
  return duration;
  Print (message.str(),12);
}

ns3::Vector
CogMan::GetNodePosition(uint32_t nodeId)
{
  Print("GetNodePosition",1);
  Ptr<MobilityModel> mobility;
  mobility = allNodes.Get(nodeId)->GetObject<MobilityModel>();
  Print("GetNodePosition",2);
  return mobility->GetPosition();
}

ns3::Vector
CogMan::GetNodePosition(uint32_t nodeId, NodeContainer nodes)
{
  Print("GetNodePosition",1);
  Ptr<MobilityModel> mobility;
  mobility = nodes.Get(nodeId)->GetObject<MobilityModel>();
  Print("GetNodePosition",2);
  return mobility->GetPosition();
}

ns3::Vector
CogMan::GetControlledNodePosition(uint32_t nodeId)
{
  Print("GetNodePosition",1);
  Ptr<MobilityModel> mobility;
  mobility = allControlledNodes.Get(nodeId)->GetObject<MobilityModel>();
  Print("GetNodePosition",2);
  return mobility->GetPosition();
}

double
CogMan::GetDistance(ns3::Vector position1, ns3::Vector position2)
{
  Print("GetDistance",1);
  return sqrt(  (position1.x-position2.x) *
                (position1.x-position2.x) +
                (position1.y-position2.y) *
                (position1.y-position2.y) );
  Print("GetDistance",2);
}

void
CogMan::GetAllDistancesFromSource(int sourceNodeId)
{
  Print("GetAllDistancesFromSource",1);
  Vector sourcePosition = GetNodePosition(sourceNodeId);

  // get distance for each node
  for (int destinationNodeId=0; destinationNodeId<allNodes.GetN(); destinationNodeId++)
  {
    if (destinationNodeId!=sourceNodeId)
    {
      Vector destinationPosition = GetNodePosition(destinationNodeId);
      double distance = GetDistance(sourcePosition, destinationPosition);

      message << "At " << Simulator::Now().GetSeconds() << "s Distance from " <<
        sourceNodeId << " to " << destinationNodeId << " is " << distance ;
      Print(message.str(), 15);
      if (InRange(distance)) // node in range                    //---------------------------
-------//
      {                                                          //  A vector called
nodesInMyWorld  //
        nodesInMyWorld.push_back({  sourceNodeId,                //  to store many
nodeDistance      //
                                    destinationNodeId,           //
//
                                    distance,                    //  The structure has
variables     //
                                    Simulator::Now().GetSeconds()//    sourceNode      int
//
                                });                              //    destinationNode int
//
      }                                                          //    distance        double
//
    }                                                            //    myTime          double
//
  }                                                              //---------------------------
-------//
  Print("GetAllDistancesFromSource",2);
}

void
CogMan::RecordControlledNodeMovement(int nodeId, double endTime)
{
  Print("RecordControlledNodeMovement",1);
  double timeNow = GetTimeNow('s');
  message << "\nNode " << nodeId << " will move at " << timeNow << " until " << endTime;
  Print (message.str(),12);
  setControlledNodeMove.push_back ({
                            nodeId,
                            timeNow,
                            endTime
                        });
  Print("RecordControlledNodeMovement",2);
```

```
}

void
CogMan::RecordMovement(int nodeId, double endTime)
{
  Print("RecordMovement",1);
  double timeNow = GetTimeNow('s');
  message << "\nNode " << nodeId << " will move at " << timeNow << " until " << endTime;
  Print (message.str(),12);
  setNodeMove.push_back ({
                           nodeId,
                           timeNow,
                           endTime
                       });
  Print("RecordMovement",2);
}

void
CogMan::CourseChange(Ptr<const MobilityModel> model)
{
  int nodeId = model->GetObject<Node> ()->GetId ();
  Vector position = GetNodePosition(nodeId);
  int dsdvNodeId = nodeId + numOfNodes;
  MoveNodeTo(dsdvNodeId, position.x, position.y );
}

double
CogMan::GetDensityFromNode(int nodeId)
{
  int totalInRange = 0;
  Ptr<Node> sourceNode;
  NodeContainer::Iterator vectorIndex;
  NodeContainer::Iterator vectorEnd;

  if (currentProtocol == "aodv")
  {
    sourceNode  = aodvNodes.Get(nodeId);
    vectorIndex = aodvNodes.Begin();
    vectorEnd   = aodvNodes.End();
  }
  else
  {
    sourceNode  = dsdvNodes.Get(nodeId);
    vectorIndex = dsdvNodes.Begin();
    vectorEnd   = dsdvNodes.End();
  }

  Ptr<MobilityModel> mob = sourceNode->GetObject<MobilityModel>();
  Vector position1 = mob->GetPosition();

  while (vectorIndex < vectorEnd)
  {
    if (sourceNode != *vectorIndex) // don't want to check ourselves
    {
      Ptr<MobilityModel> mob2 = (*vectorIndex)->GetObject<MobilityModel>();

      Vector position2 = mob2->GetPosition();
      if (InRange(GetDistance(position1, position2)))
      {
        totalInRange++;
      }
    }
    vectorIndex++;
  }
  return totalInRange / (myRadius * myRadius * PI);
}

bool
CogMan::InRange(int distance)
{
  Print("InRange",1);
  return distance < myRadius; //  true or false;
  Print("InRange",2);
}

double
CogMan::GetPredictedMaximalTimeInRange(int sourceNodeId, int destinationNodeId)
```

```
{
  ns3::Vector sNodePredictedPosition;
  ns3::Vector dNodePredictedPosition;
  double futureTime = Simulator::Now().GetSeconds();
  do
  {
    futureTime++;
    sNodePredictedPosition = PredictPosition(sourceNodeId,      futureTime);
    dNodePredictedPosition = PredictPosition(destinationNodeId, futureTime);
  }
  while (InRange(GetDistance(sNodePredictedPosition, dNodePredictedPosition))
          && futureTime < simulationTime);

  //cout << "Node " << sourceNodeId << " will be in range of " << destinationNodeId
  //  << " until " << futureTime << "s.\n";
  return futureTime;
}

ns3::Vector
CogMan::PredictPosition(int nodeId, double futureTime)
{
  Print("PredictPosition",1);
  double currentTime = floor (Simulator::Now().GetSeconds());
  ns3::Vector currentNodePosition;
  ns3::Vector previousPosition;
  ns3::Vector predictedPosition;
  double previousTime=0;
  double velPredictedX;
  double velPredictedY;
  double predictedTime;
  currentNodePosition = GetNodePosition(nodeId);
  vector<currentPosition>::iterator vectorIndex;
  vectorIndex = myPositions.end();               //----------------------------------//
                                                 //  A vector called myPositions     //
  while (vectorIndex != myPositions.begin())     //  to store many currentPositions  //
  {                                              //                                  //
    --vectorIndex;                               //  The structure has variables     //
    if (vectorIndex->nodeId == nodeId)           //    nodeId      int               //
    {                                            //    time        double            //
    previousPosition.x = vectorIndex->x;         //    x           double            //
    previousPosition.y = vectorIndex->y;         //    y           double            //
    previousTime = vectorIndex->time;            //----------------------------------//
    predictedTime=futureTime;
    velPredictedX=(currentNodePosition.x-previousPosition.x)/(currentTime-previousTime);
    velPredictedY=(currentNodePosition.y-previousPosition.y)/(currentTime-previousTime);
    predictedPosition.x=currentNodePosition.x+velPredictedX*(predictedTime-currentTime);
    predictedPosition.y=currentNodePosition.y+velPredictedY*(predictedTime-currentTime);
    return predictedPosition;
    }
  }
  Print("PredictPosition",2);
}

int
CogMan::GetNumberOfNeighbours(int sourceNodeId, NodeContainer nodes)
{
  int neighbourCount = 0;
  Print("GetNumberOfNeighbours",1);
  //cout << "\nChecking Neighbours for node " << sourceNodeId << endl;
  Ptr<Node> sourceNode = allNodes.Get(sourceNodeId);
  // get position of node pointer passed
  Ptr<MobilityModel> mob = sourceNode->GetObject<MobilityModel>();
  Vector sourcePosition = mob->GetPosition();

  // get distance for each node
  for (int destinationNodeId=0; destinationNodeId<nodes.GetN(); destinationNodeId++)
  {
    if (destinationNodeId!=sourceNodeId)
    {
      // get position of node destinationNodeId
      Ptr<MobilityModel> mob2 = nodes.Get(destinationNodeId)->GetObject<MobilityModel>();
      Vector destinationPosition = mob2->GetPosition();

      double distance = GetDistance(sourcePosition, destinationPosition);

      message << "At " << Simulator::Now().GetSeconds() << "s Distance from " <<
        sourceNodeId << " to " << destinationNodeId << " is " << distance ;
      Print(message.str(), 16);

      if (InRange(distance)) // node in range
      {
        neighbourCount++;
      }
    }
  }
  return neighbourCount;
  Print("GetNumberOfNeighbours",2);
}

void
CogMan::CheckNeighbours(int sourceNodeId)
{
  Print("CheckNeighbours",1);
  //cout << "\nChecking Neighbours for node " << sourceNodeId << endl;
  Ptr<Node> sourceNode = allNodes.Get(sourceNodeId);
  // get position of node pointer passed
  Ptr<MobilityModel> mob = sourceNode->GetObject<MobilityModel>();
  Vector sourcePosition = mob->GetPosition();

  // get distance for each node
  for (int destinationNodeId=0; destinationNodeId<allNodes.GetN(); destinationNodeId++)
  {
    if (destinationNodeId!=sourceNodeId)
    {
      // get position of node destinationNodeId
      Ptr<MobilityModel> mob2 = allNodes.Get(destinationNodeId)->GetObject<MobilityModel>();
      Vector destinationPosition = mob2->GetPosition();

      double distance = GetDistance(sourcePosition, destinationPosition);

      message << "At " << Simulator::Now().GetSeconds() << "s Distance from " <<
        sourceNodeId << " to " << destinationNodeId << " is " << distance ;
      Print(message.str(), 16);

      if (InRange(distance)) // node in range
      {
        message << "Node " << destinationNodeId << " is in range.";  //----------------------
------------//
        Print (message.str(), 16);                                   //  A vector called
myNeighbours    //
        myNeighbours.push_back({  sourceNodeId,                      //  to store local
nodeDistance     //
                                  destinationNodeId,                 //
//
                                  distance,                          //  The structure has
variables     //
                                  Simulator::Now().GetSeconds()      //    sourceNode
int           //
                               });                                   //    neighbourNode
int           //
      }                                                              //    distance
double        //
    }                                                                //    myTime
double        //
  }                                                                  //----------------------
------------//
  Print("CheckNeighbours",2);
}

double
CogMan::MoveControlledNode(int mNode)
{
  int    sNode       = GetSourceNodeForControlledNode(mNode);
  int    dNode       = GetDestinationNodeForSourceNode(sNode);
  int    cNode       = GetClosestControlledNode(sNode, dNode);
  Vector cNodePos    = GetControlledNodePosition(cNode);
  Vector cNodeNewPos = GetControlNodeMidPointXY(sNode, dNode, cNodePos); // source, dest,
controlled
  double journeyTime = GetTimeOfJourneyControlledNode(cNodePos, cNodeNewPos.x, cNodeNewPos.y,
cNodeSpeed); // 2.23 jogging speed.;
  double timeNow     = GetTimeNow('s');

  if (cNodePos.x == cNodeNewPos.x && cNodePos.y == cNodeNewPos.y)
  {
    cout << endl << timeNow << "s Controlled Node talks - I will not get to midpoint in
time!";
  }
  else
  {
    Vector sNodePos = PredictPosition(sNode, journeyTime);
    Vector dNodePos = PredictPosition(dNode, journeyTime);
    double distance = GetDistance(sNodePos, dNodePos);
    if (InRange(distance))
    {
      cout << endl << timeNow << "s NCC talks - Source and Destinitation will be in range
before you get there!";
      //journeyTime = -2;
    }
    else
    {
      ControlledNodeWalkTo(cNode, cNodeNewPos.x, cNodeNewPos.y, 0.8, cNodeSpeed);
    }
  }
  return journeyTime;
}




// TRAFFIC FLOWS

void
CogMan::CreateTrafficFlows(int n)
{
  if (n > numOfNodes)
  {
    cout << "Too many traffic flows for node count.";
  }
  else
  {
    for (int i=0; i<n; i++)
    {
      if (i != numOfNodes-1-i)
      {                         // sNode, dNode,   timeStart,   timeStop
      myTrafficFlows.push_back ({ i, numOfNodes-1-i, 0.01, simulationTime-10 });
      }
    }

    vector<traffic>::iterator vectorIndex;
    vectorIndex = myTrafficFlows.begin();

    while (vectorIndex < myTrafficFlows.end())
    {
      // from nodeId to nodeId starting at Time and stoping at Time
      CreateTrafficFlow(vectorIndex->sNode,
                        vectorIndex->dNode,
                        vectorIndex->timeStart,
                        vectorIndex->timeStop);
      vectorIndex++;
    }
  }
   //sourceNodesNumber = myTrafficFlows.size();
}

void
CogMan::CreateTrafficFlow(int sourceNodeId, int destinationNodeId, double startAt, double
stopAt)
{
  Print("CreateTrafficFlow",1);
  // Destination
  // First we have to determine if this is aodv or dsdv.
  cout << GetTimeNow('s') << "s Node " << sourceNodeId << " scheduled to transmit at " <<
startAt << "s " << "to node " <<  destinationNodeId << endl;


  CreateAodvTrafficFlow(sourceNodeId, destinationNodeId, startAt, stopAt);
  CreateDsdvTrafficFlow(sourceNodeId, destinationNodeId, startAt, stopAt);

  Print("CreateTrafficFlow",2);
}
```

```
void
CogMan::CreateAodvTrafficFlow(int sourceNodeId, int destinationNodeId, double startAt, double
stopAt)
{
  Print("CreateAodvTrafficFlow",1);
  // Destination
  Address remoteAddress(InetSocketAddress (aodvInterface.GetAddress(destinationNodeId),
port));
  PacketSinkHelper receiver ("ns3::UdpSocketFactory", remoteAddress);
  ApplicationContainer receiverapp =receiver.Install(aodvNodes.Get(destinationNodeId));
  // INSTALLING SINK-DESTINATION:
  Ptr<Socket> sink = SetupPacketReceive (aodvInterface.GetAddress
    (destinationNodeId), aodvNodes.Get (destinationNodeId));

  receiverapp.Start (Time(startAt));
  receiverapp.Stop (Time(stopAt));
  Print("Destination Created", 10);



       // lets create a server to receive packets
       //  UdpEchoServerHelper echoServer(9);
       //  ApplicationContainer serverApps = echoServer.Install(nodes.Get(dNode)); // last
node
       //  serverApps.Start (Seconds(10.0));
       //  serverApps.Stop(Seconds(20.0));

       // lets create the client to send packets
       //  UdpEchoClientHelper echoClient (interfaces.GetAddress(sNode), 9);
       //  echoClient.SetAttribute ("MaxPackets", UintegerValue(10000));
       //  echoClient.SetAttribute ("Interval", TimeValue(Seconds(1.0)));
       //  echoClient.SetAttribute ("PacketSize", UintegerValue(1024));

  // Source
  OnOffHelper sender ("ns3::UdpSocketFactory", remoteAddress);
  sender.SetConstantRate (DataRate(dataRate));
  sender.SetAttribute ("OnTime", StringValue ("ns3::ConstantRandomVariable[Constant=1.0]"));
  sender.SetAttribute ("OffTime", StringValue ("ns3::ConstantRandomVariable[Constant=0.0]"));
  ApplicationContainer senderapp;

  senderapp=sender.Install(aodvNodes.Get(sourceNodeId));

  senderapp.Start (Seconds (startAt));
  senderapp.Stop (Seconds (stopAt));

  Print("Source Created", 10);

  message << "At time: " << Simulator::Now().GetSeconds() << "s ";
  message << "setting traffic flow for node " << sourceNodeId << " (" <<
aodvInterface.GetAddress(sourceNodeId) << ")";
  message << " to send data to " << destinationNodeId << " (" <<
aodvInterface.GetAddress(destinationNodeId) << ")";
  message << " starting at " << startAt << " until " << stopAt;

  Print(message.str(), 10);
  Print("CreateAodvTrafficFlow",2);
}

void
CogMan::CreateDsdvTrafficFlow(int sourceNodeId, int destinationNodeId, double startAt, double
stopAt)
{
  Print("CreateDsdvTrafficFlow",1);
  // Destination
  // First we have to determine if this is aodv or dsdv.
  Address remoteAddress(InetSocketAddress (dsdvInterface.GetAddress(destinationNodeId),
port));
  PacketSinkHelper receiver ("ns3::UdpSocketFactory", remoteAddress);
  ApplicationContainer receiverapp =receiver.Install(dsdvNodes.Get(destinationNodeId));
  // INSTALLING SINK-DESTINATION:
  Ptr<Socket> sink = SetupPacketReceive (dsdvInterface.GetAddress
    (destinationNodeId), dsdvNodes.Get (destinationNodeId));

  receiverapp.Start (Time(startAt));
  receiverapp.Stop (Time(stopAt));
  Print("Destination Created", 10);
  // Source
  OnOffHelper sender ("ns3::UdpSocketFactory", remoteAddress);
  sender.SetConstantRate (DataRate(dataRate));
  sender.SetAttribute ("OnTime", StringValue ("ns3::ConstantRandomVariable[Constant=1.0]"));
  sender.SetAttribute ("OffTime", StringValue ("ns3::ConstantRandomVariable[Constant=0.0]"));
  ApplicationContainer senderapp;

  senderapp=sender.Install(dsdvNodes.Get(sourceNodeId));

  senderapp.Start (Seconds (startAt));
  senderapp.Stop (Seconds (stopAt));

  Print("Source Created", 10);

  message << "At time: " << Simulator::Now().GetSeconds() << "s ";
  message << "setting traffic flow for node " << sourceNodeId << " (" <<
dsdvInterface.GetAddress(sourceNodeId) << ")";
  message << " to send data to " << destinationNodeId << " (" <<
dsdvInterface.GetAddress(destinationNodeId) << ")";
  message << " starting at " << startAt << " until " << stopAt;

  Print(message.str(), 10);
  Print("CreateDsdvTrafficFlow",2);
}

bool
CogMan::StillTransmitting(int nodeId)
{
  vector<traffic>::iterator vectorIndex;
  vectorIndex = myTrafficFlows.begin();

  bool transmitting = false;
  double timeNow = GetTimeNow('s');

  while (vectorIndex != myTrafficFlows.end())
  {
    if ((vectorIndex->sNode == nodeId) && (vectorIndex->timeStart < timeNow) && (vectorIndex-
>timeStop > timeNow))
    {
      transmitting = true;
      break;
    }
    vectorIndex++;
  }
  return transmitting;
}


//  We are here!  ---  page 148 of document.
//  We are here!
//  We are here!
//  We are here!
//  We are here!
//  We are here!
//  We are here!
//  We are here!


Vector
CogMan::GetNewVectorUsingAngleAndDistance(Vector currentPosition, double distance, double
degree)
{
  double alpha = 3 * PI / 2 + PI * degree / 180;
  currentPosition.x = currentPosition.x + distance*cos(alpha);
  currentPosition.y = currentPosition.y + distance*sin(alpha);
  return currentPosition;
}

void
CogMan::PrintNodeInformation(Ptr<Node> node)
{
  message << "Node ID: " << node->GetId() << "   Memory Address: " << node;
  Print(message.str(), 18);

  message << "  Has " << node->GetNApplications() << " applications";
  Print(message.str(), 18);

  message << "  Has " << node->GetNDevices() << " network devices";
  Print(message.str(), 18);

  Ptr<Ipv4> ipv4 = node->GetObject<Ipv4>();
  message << "  Has " << ipv4->GetNInterfaces() << " interfaces";
  Print(message.str(),18);

  for (int i=0; i<ipv4->GetNInterfaces(); i++)
  {
    message << "    Interface " << i;
    Print(message.str(),18);
    for (int n=0; n<ipv4->GetNAddresses(i); n++)
    {
    Ipv4InterfaceAddress iaddr = ipv4->GetAddress (i,n);
    Ipv4Address addri = iaddr.GetLocal ();

      message << "        IP Address " << n << " is " << addri;
      Print(message.str(), 18);
    }
  }
  Print ("\n",18);
}



// PACKETS

void
CogMan::PhyRxDropReason(Ptr<const Packet> p, int nodeId)
{
  int reasonCount = 0;
  Print("PhyRxDropReason",1);
  string packetType = "undefined";

  string src_ip = "";
  string des_ip = "";

  if (IsIPv4Header(p))
  {
    src_ip = GetSourceIPAddress(p);
    des_ip = GetDestinationIPAddress(p);
  }

  if (IsCTL_ACKPacket(p))
  {
    packetType = "Clear to send"; // no ip header
    reasonCount++;
  }
  if (IsAODV_RREPPacket(p))
  {
    packetType = "AODV Route Reply";
    reasonCount++;
  }
  if (IsAODV_RREQPacket(p))
  {
    packetType = "AODV Route Request";
    reasonCount++;
  }
  if (IsARPPacketRequest(p))
  {
    packetType = "MAC Resolution Protocol Request";
    reasonCount++;
  }
  if (IsARPPacketReply(p))
  {
    packetType = "MAC Resolution Protocol Reply";
    reasonCount++;
  }
  if (IsDSDV_RREQPacket(p))
  {
    packetType = "DSDV Route Request";
    reasonCount++;
  }
  if (IsDSDV_RREPPacket(p))
  {
    packetType = "DSDV Route Reply";
    reasonCount++;
  }

  if (IsPayloadPacket(p))
  {
    string da = GetDestinationMACAddress(p);
    if (da == "ff")
    {
      reasonCount++;
      packetType = "broadcast";
    }
    else if (da == "0a")
    {
      reasonCount++;
      packetType = "unknown route";
    }
    //else if (nodeId != stoi(da))
    //{
    //  reasonCount++;
    //  packetType = "intended for " + da;
    //}
    else if (nodeId == 14)
    {
      reasonCount++;
      packetType = "payload for 14 ???";
    }
  }

  if (packetType == "undefined")
  {
       //cout << *p << endl << endl;
  }
  if (reasonCount > 1)
  {
  cout << "REASON COUNT IS: " << reasonCount;
  }

  UpdateDroppedPacketCounters(nodeId, packetType);
  //RecordDroppedPacket(src_ip, des_ip);
  Print("PhyRxDropReason",2);
}

bool
CogMan::IsPayloadPacket(Ptr<const Packet> p)
{
  string header;           // string to store packet header
  ostringstream convert;   // stream used for the conversion
  convert << *p;           // copy into ostringstream object
  header = convert.str();  // convert to a string

  if (header.find("Payload")==std::string::npos)
  {
    return false;
  }
  else
  {
    return true;
  }
}

string
CogMan::GetDestinationMACAddress(Ptr<const Packet> p)
{
  string header;           // string to store packet header
  ostringstream convert;   // stream used for the conversion
  convert << *p;           // copy into ostringstream object
  header = convert.str();  // convert to a string

  int myPointer = header.find("DA=");
  return header.substr(myPointer+18, 2);
}

string
CogMan::GetSourceIPAddress(Ptr<const Packet> p)
{
  string header;           // string to store packet header
  ostringstream convert;   // stream used for the conversion
  convert << *p;           // copy into ostringstream object
  header = convert.str();  // convert to a string

  int myStart   = header.find("10.");
  int myFinish  = header.find(">") - 1;
  return header.substr(myStart, myFinish-myStart);
}

string
CogMan::GetDestinationIPAddress(Ptr<const Packet> p)
{
  string header;           // string to store packet header
  ostringstream convert;   // stream used for the conversion
  convert << *p;           // copy into ostringstream object
  header = convert.str();  // convert to a string

  int myStart   = header.find("> 10.") + 2;
  int myFinish  = header.find(")", myStart);
  return header.substr(myStart, myFinish-myStart);
}

bool
CogMan::IsCTL_ACKPacket(Ptr<const ns3::Packet> p)
{
  string header;           // string to store packet header
  ostringstream convert;   // stream used for the conversion
  convert << *p;           // copy into ostringstream object
  header = convert.str();  // convert to a string

  if (header.find("CTL_ACK")==std::string::npos)
  {
    return false;
  }
  else
  {
    return true;
  }
}

bool
CogMan::IsAODV_RREPPacket(Ptr<const ns3::Packet> p)
{
  string header;           // string to store packet header
  ostringstream convert;   // stream used for the conversion
  convert << *p;           // copy into ostringstream object
  header = convert.str();  // convert to a string

  if (header.find("aodv::TypeHeader (RREP)")==std::string::npos)
  {
    return false;
  }
  else
  {
    return true;
  }
}

bool
CogMan::IsAODV_RREQPacket(Ptr<const ns3::Packet> p)
{
  string header;           // string to store packet header
  ostringstream convert;   // stream used for the conversion
  convert << *p;           // copy into ostringstream object
  header = convert.str();  // convert to a string

  if (header.find("aodv::TypeHeader (RREQ)")==std::string::npos)
  {
    return false;
  }
  else
  {
    return true;
  }
}

bool
```

```
CogMan::IsDSDV_RREQPacket(Ptr<const ns3::Packet> p)
{
  string header;            // string to store packet header
  ostringstream convert;    // stream used for the conversion
  convert << *p;            // copy into ostringstream object
  header = convert.str();   // convert to a string

  int occurances = 0;

  std::size_t found = header.find("dsdv::DsdvHeader (");
  if (found != std::string::npos)
  {
    occurances++;
  }

  found = header.find("dsdv::DsdvHeader (", found + 1);
  if (found != std::string::npos)
  {
    occurances++;
  }

  if (occurances == 1)
  {
    return true;
  }
  else
  {
    return false;
  }
}

bool
CogMan::IsDSDV_RREPPacket(Ptr<const ns3::Packet> p)
{
  string header;            // string to store packet header
  ostringstream convert;    // stream used for the conversion
  convert << *p;            // copy into ostringstream object
  header = convert.str();   // convert to a string

  int occurances = 0;

  std::size_t found = header.find("dsdv::DsdvHeader (");
  if (found != std::string::npos)
  {
    occurances++;
  }

  found = header.find("dsdv::DsdvHeader (", found + 1);
  if (found != std::string::npos)
  {
    occurances++;
  }

  if (occurances == 2)
  {
    return true;
  }
  else
  {
    return false;
  }
}

bool
CogMan::IsARPPacketRequest(Ptr<const ns3::Packet> p)
{
  string header;            // string to store packet header
  ostringstream convert;    // stream used for the conversion
  convert << *p;            // copy into ostringstream object
  header = convert.str();   // convert to a string

  if (header.find("ArpHeader (request")==std::string::npos)
  {
    return false;
  }
  else
  {
    return true;
  }
}

bool
CogMan::IsARPPacketReply(Ptr<const ns3::Packet> p)
{
  string header;           // string to store packet header
  ostringstream convert;   // stream used for the conversion
  convert << *p;           // copy into ostringstream object
  header = convert.str();  // convert to a string

  if (header.find("ArpHeader (reply")==std::string::npos)
  {
    return false;
  }
  else
  {
    return true;
  }
}

bool
CogMan::IsIPv4Header(Ptr<const ns3::Packet> p)
{
  string header;           // string to store packet header
  ostringstream convert;   // stream used for the conversion
  convert << *p;           // copy into ostringstream object
  header = convert.str();  // convert to a string

  if (header.find("Ipv4Header (")==std::string::npos)
  {
    return false;
  }
  else
  {
    return true;
  }
}

bool
CogMan::RecordDroppedPacketInformation(Ptr<const Packet> p)
{
  //droppedPacketTotal++;
  bool Payload = false;

  std::stringstream temp;
  p->Print(temp);
  string packetHeader = temp.str();

  if (packetHeader.find("Payload") != std::string::npos)
  { // found a payload
    Payload = true;
    if (packetHeader.find("10.1") != std::string::npos)
    {
      dropDsdvDataPacket++;
    }
    if (packetHeader.find("10.2") != std::string::npos)
    {
      dropAodvDataPacket++;
    }
  }
  else if (packetHeader.find("dsdv") != std::string::npos || packetHeader.find("10.1") !=
std::string::npos)
  {
    dropDsdvControlPacket++;
  }
  else if (packetHeader.find("aodv") != std::string::npos || packetHeader.find("10.2") !=
std::string::npos)
  {
    dropAodvControlPacket++;
  }
  else
  {
    //cout << "\nUnknown drop packet found\n";
    //cout << packetHeader;
  }
  return Payload;
}

void
CogMan::ForwardPacket(const Ipv4Header &header, Ptr<const Packet> packet, uint32_t interface)
{
  Print("ForwardPacket",1);
  //cout << "------------begin packet forwarded from node 0 ------------\n";
    //RecordSendPacketInformation(p);
  Ipv4Address src_ip = header.GetSource();
  Ipv4Address des_ip = header.GetDestination();
  uint16_t id = header.GetIdentification();
  uint8_t ttl = header.GetTtl();
  uint16_t fo = header.GetFragmentOffset();

  message << GetTimeNow('s') << " - Packet Forwarded (" << src_ip << " --> " << des_ip << ") - 
id:" << id << endl;
  Print (message.str(), 43);  // using 5 for now as forward packet has no group number 
assigned.
  // cout << "  time to live is    " << ttl << endl;
  // cout << "  fragment offset is " << fo << endl;
  //The following code is based from 
  //http://polythinking.wordpress.com/2012/05/30/ns-3-network-simulator-how-to-find-a-
specific-header-in-packet-in-ns-3/



    string headerIP;           // string to store IP Address
    ostringstream convertIP;      // stream used for the conversion
    convertIP << header;            // copy into ostringstream object
    headerIP = convertIP.str();  // convert to a string

    //cout << headerIP << endl;



    string header2;           // string to store IP Address
    ostringstream convert;      // stream used for the conversion
    convert << *packet;            // copy into ostringstream object
    header2 = convert.str();  // convert to a string

    //cout << header2 << endl;


  //cout << "------------end packet forwarded from node 0 ------------\n";
  Print("ForwardPacket",2);
}


// MANAGEMENT

int
CogMan::GetSourceNodeForControlledNode(int mNode)
{
  double timeNow = GetTimeNow('s');
  // first find source node that needs controlled node most.
  vector<cognitivity>::iterator vectorIndex0;
  vectorIndex0 = managementNodeMemory.end();

  bool   foundSource = false;
  double minRR = 9999999;
  int    sNode = -1;
  while (!foundSource && vectorIndex0 >= managementNodeMemory.begin() && minRR != 0)
  {
    vectorIndex0--;
    if (vectorIndex0->mNode == mNode && vectorIndex0->requestProgress == 'A' && vectorIndex0-
>request == 'M')
    {
      if (vectorIndex0->receiveRate < minRR)
      {
        sNode = vectorIndex0->sNode;
        minRR = vectorIndex0->receiveRate;
      }
    }
  }
  cout << endl << timeNow << "s Manager " << mNode << " talks - source " << sNode << " needs 
link fixed most.";
```

```
  return sNode;
}

int
CogMan::GetDestinationNodeForSourceNode(int sNode)
{
  int minPR = 9999999;
  int dNode = -1;
  int chosenDNode = -1;
  int numPR = 0;
  double timeNow = GetTimeNow('s');

  vector<traffic>::iterator vectorIndex;
  vectorIndex = myTrafficFlows.end();
  while (vectorIndex >= myTrafficFlows.begin() )
  {
    vectorIndex--;
    if (vectorIndex->timeStop > timeNow && vectorIndex->timeStart < timeNow && vectorIndex-
>sNode == sNode)
    {
      dNode = vectorIndex->dNode;
      cout << endl << timeNow << "s Source " << sNode << " talks - I'm trying to transmit to "
<< dNode;
      numPR = GetCurrentReceivedPacketsForDestinationFromSource(sNode, dNode, timeNow-5,
timeNow);
      cout << endl << timeNow << "s Destinitation " << dNode << " talks - I've received " <<
numPR << " packets for the last 5 seconds";
      if (numPR < minPR)
      {
        chosenDNode = dNode;
        minPR = numPR;
      }
    }
  }
  cout << endl << timeNow << "s Manager talks - I've chosen to attempt to fix breakage between
(" << sNode << "-->" << dNode << ")";
  return chosenDNode;
}

int
CogMan::GetClosestControlledNode(int sNode, int dNode)
{
  double timeNow = GetTimeNow('s');
  // get closest controlled node.
  int chosenCNode = -1;
  double minDist  = 99999999;
  for (int i=0; i<aodvControlledNodes.GetN(); i++)
  {
    if (IsControlledNodeMoving(i) == 0) // 0 then node is not moving
    {
      Vector cNodePos = GetControlledNodePosition(i);
      Vector dNodePos = GetNodePosition(dNode);
      Vector sNodePos = GetNodePosition(sNode);
      Vector middle;
      middle.x = (sNodePos.x + dNodePos.x) / 2;
      middle.y = (sNodePos.y + dNodePos.y) / 2;
      double dist = GetDistance(cNodePos,middle);
      if (dist < minDist)
      {
        minDist = dist;
        chosenCNode = i;
      }
    }
  }
  cout << endl << timeNow << "s Manager talks - Chosen node is " << chosenCNode;
  return chosenCNode;
}

Vector
CogMan::GetControlNodeMidPointXY(int sNode, int dNode, Vector cNode)
{
  // how long will it take to get to the middle point?
  double timeNow = GetTimeNow('s');
  double futureTime = timeNow;
  double journeyTime = simulationTime;
  bool   reachedMidPoint = false;
  while (!reachedMidPoint && simulationTime > futureTime)
  {
    Vector source = PredictPosition(sNode,futureTime);
    Vector destin = PredictPosition(dNode,futureTime);

    double x = (source.x + destin.x) / 2;
    double y = (source.y + destin.y) / 2;

    journeyTime   = GetTimeOfJourneyControlledNode(cNode, x, y, cNodeSpeed); // 2.23 jogging
speed.

    //cout << "\nCurrent Time: " << timeNow;
    //cout << "\nJourney Time: " << journeyTime;
    //cout << "\nFurture Time: " << futureTime;

    //cout << "\n x1,y1 is:" << source.x << "," << source.y;
    //cout << "\n x2,y2 is:" << destin.x << "," << destin.y;
    //cout << "\n xc,yc is:" << cNode.x << "," << cNode.y;
    //cout << "\n xm,ym is:" << x << "," << y;

    if (journeyTime + timeNow < futureTime)
    {
      cout << endl << timeNow << "s Controlled Node talks - I can get to the mid point at: "
<< futureTime;
      cNode.x = x;
      cNode.y = y;
      reachedMidPoint = true;
    }
    futureTime = futureTime + 1;
  }
  return cNode;
}



int
CogMan::GetManagerToMoveNodeFor()
{
  vector<request>::iterator vectorIndex;
  vectorIndex = requests.begin();
  double minRR = 999999;
  double mNode = 0;

  while (vectorIndex < requests.end())
  {
    if (vectorIndex->request == 'M')
    {
      if (vectorIndex->predictedReceivingRate < minRR)
      {
        mNode = vectorIndex->mNode;
        minRR = vectorIndex->predictedReceivingRate;
      }
    }
    vectorIndex++;
  }

  // show dealing with request
  vectorIndex = requests.begin();
  while (vectorIndex < requests.end())
  {
    if (vectorIndex->mNode == mNode)
    {
      vectorIndex->request == 'Z';
      break;
    }
    vectorIndex++;
  }

  cout << endl << GetTimeNow('s') << "s NCC talks - The manager authorised to move node is: "
<< mNode;
  return mNode;
}

void
CogMan::DenyAllRequestForControlledNode(double backOffUntil)
{
  double timeNow = GetTimeNow('s');
  vector<cognitivity>::iterator vectorIndex;
  vectorIndex = managementNodeMemory.end();

  while (vectorIndex >= managementNodeMemory.begin())
  {
    vectorIndex--;
    if (vectorIndex->request == 'M' && vectorIndex->requestProgress == 'A')
    {
      vectorIndex->requestProgress = 'D';
      vectorIndex->backOffTime = backOffUntil;
      cout << endl << timeNow << "s NCC talks - Request Denied to move controlled node for 
manager " << vectorIndex->mNode;
    }
  }
  //PrintVectorOfMemoryEntries();
}

void
CogMan::DenyARequestForControlledNode(double backOffUntil, int mNode)
{
  double timeNow = GetTimeNow('s');
  vector<cognitivity>::iterator vectorIndex;
  vectorIndex = managementNodeMemory.end();

  while (vectorIndex >= managementNodeMemory.begin())
  {
    vectorIndex--;
    if (vectorIndex->request == 'M' && vectorIndex->mNode == mNode)
    {
      vectorIndex->requestProgress = 'D';
      vectorIndex->backOffTime = backOffUntil;
      cout << endl << timeNow << "s NCC talks - Request Denied to move controlled node for 
manager " << vectorIndex->mNode;
      break;
    }
  }
  //PrintVectorOfMemoryEntries();
}

void
CogMan::GrantRequestForControlledNode(int manager, double backOffTime)
{
  double timeNow = GetTimeNow('s');
  vector<cognitivity>::iterator vectorIndex;
  vectorIndex = managementNodeMemory.end();

  while (vectorIndex >= managementNodeMemory.begin())
  {
    vectorIndex--;
    if (vectorIndex->request == 'M' && vectorIndex->requestProgress == 'A' && vectorIndex-
>mNode == manager)
    {
      vectorIndex->requestProgress = 'P';
      cout << endl << timeNow << "s NCC talks - Request Permitted to move controlled node for 
manager " << vectorIndex->mNode;
      vectorIndex->backOffTime = backOffTime; //timeNow + journeyTime;
      Simulator::Schedule(Seconds (backOffTime-timeNow), &CogMan::RecordActionOutcome,
                                     this, vectorIndex->atTime, manager, 'M');
    }
  }
  PrintVectorOfMemoryEntries();
}

void
CogMan::GrantRequestForSwitchProtocol(double backOffTime)
{
  double timeNow = GetTimeNow('s');
  vector<cognitivity>::iterator vectorIndex;
  vectorIndex = managementNodeMemory.end();

  while (vectorIndex >= managementNodeMemory.begin())
  {
    vectorIndex--;
    if (vectorIndex->request == 'P' && vectorIndex->requestProgress == 'A')
    {
      vectorIndex->requestProgress = 'P';
      cout << endl << timeNow << "s NCC talks - Request Permitted to Switch Protocol ";
      vectorIndex->backOffTime = backOffTime + timeNow;
      int manager = vectorIndex->mNode;
      Simulator::Schedule(Seconds (backOffTime), &CogMan::RecordActionOutcome,
                                        this, vectorIndex->atTime, manager, 'P');
    }
  }
  PrintVectorOfMemoryEntries();
}

void
CogMan::GrantRequestForChangeChannel(double backOffTime)
{
  double timeNow = GetTimeNow('s');
  vector<cognitivity>::iterator vectorIndex;
  vectorIndex = managementNodeMemory.end();

  while (vectorIndex >= managementNodeMemory.begin())
  {
    vectorIndex--;
    if (vectorIndex->request == 'C' && vectorIndex->requestProgress == 'A')
    {
      vectorIndex->requestProgress = 'P';
      cout << endl << timeNow << "s NCC talks - Request Permitted to Change Channel ";
      vectorIndex->backOffTime = backOffTime + timeNow;
      int manager = vectorIndex->mNode;
      Simulator::Schedule(Seconds (backOffTime), &CogMan::RecordActionOutcome,
                                        this, vectorIndex->atTime, manager, 'C');
    }
  }
  PrintVectorOfMemoryEntries();
}

void
CogMan::RecordActionOutcome(double atTime, int mNode, char action)
{

  cout << "\nperformance update....\n";
  cout << "\nTime Now is: " << GetTimeNow('s');
  cout << "\nTime of request was: " << atTime;
  cout << "\nFor manager: " << mNode;
  cout << "\nAction authorised was: " << action << "...";

  double timeNow = GetTimeNow('s');
  int outcome = 0;
  vector<cognitivity>::iterator vectorIndex;
  vectorIndex = managementNodeMemory.end();

  while (vectorIndex > managementNodeMemory.begin() )
  {
    vectorIndex--;
    if (vectorIndex->atTime == atTime && vectorIndex->mNode == mNode &&
        vectorIndex->request == action && vectorIndex->requestProgress == 'P')
    {
      int sNode = vectorIndex->sNode;
      int packetsReceived = GetCurrentReceivedPacketsForSource(sNode, atTime, timeNow);
      int packetsSent     = GetCurrentSentPacketsForSource(sNode, atTime, timeNow);
      int packetsDropped  = GetCurrentDroppedPacketsForSource(sNode, atTime, timeNow);
      double recPerSec    = packetsReceived / (timeNow - atTime);
      double senPerSec    = packetsSent / (timeNow - atTime);
      double droPerSec    = packetsDropped / (timeNow - atTime);

      if (recPerSec > vectorIndex->receiveRate)
      {
        outcome++;
      }
      if (recPerSec < vectorIndex->receiveRate)
      {
        outcome--;
      }

      if (droPerSec < vectorIndex->dropRate)
      {
        outcome++;
      }
      if (droPerSec > vectorIndex->dropRate)
      {
        outcome--;
      }

      if (senPerSec > vectorIndex->sendRate)
      {
        outcome++;
      }
      if (senPerSec < vectorIndex->sendRate)
      {
        outcome--;
      }
      vectorIndex->performance = outcome;
    }
  }
  //PrintVectorOfMemoryEntries();
}



double
CogMan::TimeAllowedToMoveControlledNode()
{
  double timeNow = GetTimeNow('s');
  double nodeMovingUntil = simulationTime;
  double backOffUntil    = simulationTime;
  for (int i=0; i<aodvControlledNodes.GetN(); i++)
  {
    nodeMovingUntil = IsControlledNodeMoving(i); // 0 then node is not moving

    if (nodeMovingUntil < backOffUntil && nodeMovingUntil != 0)
    {
      backOffUntil = nodeMovingUntil;
      cout << endl << timeNow << "s Controlled Node " << i << " talks - I busy for another " 
<< backOffUntil << "s";
    }
    else if (nodeMovingUntil == 0)
    {
      cout << endl << timeNow << "s Controlled Node " << i << " talks - I'm available";
      backOffUntil = 0;
    }
  }
  return backOffUntil;
}

void
CogMan::NetworkCommandCentre()// cognitivity!!!
{
  NCCScheduled = false;
  double timeNow = GetTimeNow('s');

  timeToStabilize = timeNow + 2;
  requests = newRequests;
  newRequests.clear();

  cout << endl << timeNow << "s NCC talks - The number of requests is: " << requests.size();

  vector<request>::iterator vectorIndex;
  vectorIndex = requests.begin();

  // we will go with most popular request
  int M = 0, P = 0, C = 0; // move, protocol, channel.

  while (vectorIndex < requests.end())
  {
    cout << endl << timeNow << "s NCC talks - Wait! Considering requests";
    cout << endl << timeNow << "s  Manager " << vectorIndex->mNode << " requests " << 
vectorIndex->request << 
    " if no action predicted receive rate is " << vectorIndex->predictedReceivingRate;

    if (vectorIndex->request == 'M')
      M++;
    if (vectorIndex->request == 'C')
      C++;
    if (vectorIndex->request == 'P')
      P++;

    vectorIndex++;
  }
  cout << endl << timeNow << "s  Total requests to move node      :" << M;
  cout << endl << timeNow << "s  Total requests to change channel :" << C;
  cout << endl << timeNow << "s  Total requests to switch protocol:" << P;

  // MOVE NODE
  bool didMoveNode = false;
  if (M > C && M > P)
  {
    cout << endl << timeNow << "s NCC talks move node most popular - controlled node count is
" << aodvControlledNodes.GetN();

    double backOffUntil = TimeAllowedToMoveControlledNode(); // if > 0 returned all controlled
nodes are busy

    if (backOffUntil > 0)
    {
      M = 0; // can't move node.
      // deny requests with back off time.
      cout << endl << timeNow << "s NCC talks - No controllable nodes available for moving!";
      DenyAllRequestForControlledNode(backOffUntil + timeNow);
    }
    else
    {
      cout << endl << timeNow << "s NCC talks - Moving is most popular and we can accomodate";
      int mCount = M;
      while (mCount > 0 && TimeAllowedToMoveControlledNode() <= 0)
      {
        mCount--;
        int        manager = GetManagerToMoveNodeFor();
        double journeyTime = MoveControlledNode(manager);

        if (journeyTime < 0)
        {
          DenyARequestForControlledNode(timeNow + backOffUntil, manager);    //  takes longer
than permitted time to get there.
        }
        else
        {
          GrantRequestForControlledNode(manager, timeNow + journeyTime);
          didMoveNode = true;
        }
      }
    }
  }
  //   END OF MOVE NODE REQUEST
  if (!didMoveNode)
    M = 0;  // could not meet any managers expectations.


  if (C > M && C > P)
  {
    int currentChannel = GetChannel(0);
    cout << endl << timeNow << "s NCC talks - Change channel is most popular - we are
currently on channel: " << currentChannel;
    if (currentChannel == 1 || currentChannel == 4)
    {
      for (int i=0; i<numOfNodes; i++)
        SetChannel(i,currentChannel+1);
      for (int i=0; i<numOfControlledNodes; i++)
        SetChannelControlledNodes(i,currentChannel+1);
    }
    else
    {
      for (int i=0; i<numOfNodes; i++)
        SetChannel(i,currentChannel-1);
      for (int i=0; i<numOfControlledNodes; i++)
        SetChannelControlledNodes(i,currentChannel-1);
    }
    double backOffTime = GetPeriodToStablise("changeChannel");
    GrantRequestForChangeChannel(backOffTime);
    cout << endl << timeNow << "s NCC talks - We have switched channel to: " << GetChannel(0);

    PrintAllChannels();
  }

  if (P > M && P > C)
  {
    cout << endl << timeNow << "s NCC talks - Switch protocol is most popular - we are
currently on protocol: " << currentProtocol;
    double backOffTime = GetPeriodToStablise("switchProtocol");
    GrantRequestForSwitchProtocol(backOffTime);
    SwitchProtocol();
  }

  if (P == M && M == C)
  {
    cout << "\n we have a three way tie!";
  }
}

void
CogMan::AutonomousControlledNode(int n)
{
  // if node not requested to move.
  double timeNow = GetTimeNow('s');
  double movingUntil = IsControlledNodeMoving(n);
  if (movingUntil <= 0) // not moving currently
  {
    int sumOfX = 0;
    int sumOfY = 0;
    int count = 0;
    vector<currentPosition>::iterator vectorIndex;
    vectorIndex = myPositions.end();
    if (myPositions.size() > 0)
    {
      vectorIndex--;
      double recordedTime = vectorIndex->time;

      while (vectorIndex >= myPositions.begin() && vectorIndex->time == recordedTime)
      {
        count++;
        sumOfX = sumOfX + vectorIndex->x;
        sumOfY = sumOfY + vectorIndex->y;
        vectorIndex--;
      }

      // center of world
      Vector currentPosition = GetControlledNodePosition(n);
      Vector goalPosition;
         goalPosition.x = sumOfX / count;
         goalPosition.y = sumOfY / count;
      double distance  = GetDistance(currentPosition, goalPosition);
      double stepCount = distance / 0.8; // average step size (stride)
      double xStep     = ((goalPosition.x - currentPosition.x) / stepCount) +
currentPosition.x;
      double yStep     = ((goalPosition.y - currentPosition.y) / stepCount) +
currentPosition.y;

      MoveControlledNodeTo(n, xStep, yStep);
      MoveControlledNodeTo(n + numOfControlledNodes, xStep, yStep); // corrosponding DSDV node
      Simulator::Schedule (Seconds (1), &CogMan::AutonomousControlledNode, this, n);
    }
  }
  else
  {
    Simulator::Schedule (Seconds (movingUntil), &CogMan::AutonomousControlledNode, this, n);
  }
}





void
CogMan::TestRadius()
{
  cout << "\nTesting Transmission Radius";
  Set100mRadius();
  numOfNodes = 2;
  CreateAodvNodes(2);
  allNodes = NodeContainer::GetGlobal(); // allNodes used by some methods.
  CreateTrafficFlows(1);
  packetRecdInfo = true;
  MoveNodeTo(0,0,0);
  MoveNodeTo(1,0,0);
  for (int i=1; i<simulationTime; i++)
  {
    Simulator::Schedule (Seconds (i), &CogMan::MoveNodeTo, this, 1,i,0);
  }
}

void
CogMan::Set21mRadius()
{
  EnergyDetectionThreshold = -61.76;  // original value was -96.0
  CcaMode1Threshold = -61.76;         // original value was -99.0
  TxGain = 4.5;                      // original value was 4.5
  RxGain = 4.5;                      // original value was 4.5
  TxPowerLevels = 1;                 // attributes are actually from
  TxPowerEnd = 16;                   // YansWifiPhy class
  TxPowerStart = 16;                 // has full list of attributes
  RxNoiseFigure = 4;
  myRadius = 21;
}

void
CogMan::Set100mRadius()
{
  EnergyDetectionThreshold = -61.76;  // original value was -96.0
  CcaMode1Threshold = -61.76;         // original value was -99.0
  TxGain = 14.5;                      // original value was 4.5
  RxGain = 14.5;                      // original value was 4.5
  TxPowerLevels = 1;                 // attributes are actually from
  TxPowerEnd = 16;                   // YansWifiPhy class
  TxPowerStart = 16;                 // has full list of attributes
  RxNoiseFigure = 4;
  myRadius = 100;
}




void
CogMan::Callbacks()
{
  if (mobilityModel == 'R')
  {
    for (int n=0; n<numOfNodes; n++)
    {
    string test = "/NodeList/" + to_string(n) + "/$ns3::MobilityModel/CourseChange";
    Config::ConnectWithoutContext (test, MakeCallback (&CogMan::CourseChange, this));
    }
  }
  // dropped packets
  if (numOfNodes >= 0)
  {
  Config::ConnectWithoutContext("/NodeList/0/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop0,  this));
  Config::ConnectWithoutContext("/NodeList/1/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop1,  this));
  Config::ConnectWithoutContext("/NodeList/2/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop2,  this));
  Config::ConnectWithoutContext("/NodeList/3/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop3,  this));
  Config::ConnectWithoutContext("/NodeList/4/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop4,  this));
  Config::ConnectWithoutContext("/NodeList/5/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop5,  this));
  Config::ConnectWithoutContext("/NodeList/6/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop6,  this));
  Config::ConnectWithoutContext("/NodeList/7/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop7,  this));
  Config::ConnectWithoutContext("/NodeList/8/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop8,  this));
  Config::ConnectWithoutContext("/NodeList/9/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop9,  this));
```

```
  Config::ConnectWithoutContext("/NodeList/10/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop10, this));
  Config::ConnectWithoutContext("/NodeList/11/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop11, this));
  Config::ConnectWithoutContext("/NodeList/12/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop12, this));
  Config::ConnectWithoutContext("/NodeList/13/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop13, this));
  Config::ConnectWithoutContext("/NodeList/14/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop14, this));
  Config::ConnectWithoutContext("/NodeList/15/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop15, this));
  Config::ConnectWithoutContext("/NodeList/16/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop16, this));
  Config::ConnectWithoutContext("/NodeList/17/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop17, this));
  Config::ConnectWithoutContext("/NodeList/18/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop18, this));
  Config::ConnectWithoutContext("/NodeList/19/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop19, this));
  }

  if (numOfNodes >= 20)
  {
  Config::ConnectWithoutContext("/NodeList/20/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop20,  this));
  Config::ConnectWithoutContext("/NodeList/21/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop21,  this));
  Config::ConnectWithoutContext("/NodeList/22/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop22,  this));
  Config::ConnectWithoutContext("/NodeList/23/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop23,  this));
  Config::ConnectWithoutContext("/NodeList/24/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop24,  this));
  Config::ConnectWithoutContext("/NodeList/25/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop25,  this));
  Config::ConnectWithoutContext("/NodeList/26/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop26,  this));
  Config::ConnectWithoutContext("/NodeList/27/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop27,  this));
  Config::ConnectWithoutContext("/NodeList/28/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop28,  this));
  Config::ConnectWithoutContext("/NodeList/29/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop29,  this));
  Config::ConnectWithoutContext("/NodeList/30/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop30, this));
  Config::ConnectWithoutContext("/NodeList/31/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop31, this));
  Config::ConnectWithoutContext("/NodeList/32/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop32, this));
  Config::ConnectWithoutContext("/NodeList/33/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop33, this));
  Config::ConnectWithoutContext("/NodeList/34/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop34, this));
  Config::ConnectWithoutContext("/NodeList/35/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop35, this));
  Config::ConnectWithoutContext("/NodeList/36/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop36, this));
  Config::ConnectWithoutContext("/NodeList/37/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop37, this));
  Config::ConnectWithoutContext("/NodeList/38/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop38, this));
  Config::ConnectWithoutContext("/NodeList/39/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop39, this));
  }

  if (numOfNodes >= 40)
  {
  Config::ConnectWithoutContext("/NodeList/40/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop40, this));
  Config::ConnectWithoutContext("/NodeList/41/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop41, this));
  Config::ConnectWithoutContext("/NodeList/42/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop42, this));
  Config::ConnectWithoutContext("/NodeList/43/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop43, this));
```

```
  Config::ConnectWithoutContext("/NodeList/44/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop44, this));
  Config::ConnectWithoutContext("/NodeList/45/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop45, this));
  Config::ConnectWithoutContext("/NodeList/46/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop46, this));
  Config::ConnectWithoutContext("/NodeList/47/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop47, this));
  Config::ConnectWithoutContext("/NodeList/48/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop48, this));
  Config::ConnectWithoutContext("/NodeList/49/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop49, this));
  Config::ConnectWithoutContext("/NodeList/50/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop50, this));
  Config::ConnectWithoutContext("/NodeList/51/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop51, this));
  Config::ConnectWithoutContext("/NodeList/52/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop52, this));
  Config::ConnectWithoutContext("/NodeList/53/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop53, this));
  Config::ConnectWithoutContext("/NodeList/54/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop54, this));
  Config::ConnectWithoutContext("/NodeList/55/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop55, this));
  Config::ConnectWithoutContext("/NodeList/56/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop56, this));
  Config::ConnectWithoutContext("/NodeList/57/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop57, this));
  Config::ConnectWithoutContext("/NodeList/58/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop58, this));
  Config::ConnectWithoutContext("/NodeList/59/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop59, this));
  Config::ConnectWithoutContext("/NodeList/60/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop60, this));
  Config::ConnectWithoutContext("/NodeList/61/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop61, this));
  Config::ConnectWithoutContext("/NodeList/62/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop62, this));
  Config::ConnectWithoutContext("/NodeList/63/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop63, this));
  Config::ConnectWithoutContext("/NodeList/64/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop64, this));
  Config::ConnectWithoutContext("/NodeList/65/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop65, this));
  Config::ConnectWithoutContext("/NodeList/66/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop66, this));
  Config::ConnectWithoutContext("/NodeList/67/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop67, this));
  Config::ConnectWithoutContext("/NodeList/68/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop68, this));
  Config::ConnectWithoutContext("/NodeList/69/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop69, this));
  Config::ConnectWithoutContext("/NodeList/70/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop70, this));
  Config::ConnectWithoutContext("/NodeList/71/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop71, this));
  Config::ConnectWithoutContext("/NodeList/72/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop72, this));
  Config::ConnectWithoutContext("/NodeList/73/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop73, this));
  Config::ConnectWithoutContext("/NodeList/74/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop74, this));
  Config::ConnectWithoutContext("/NodeList/75/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop75, this));
  Config::ConnectWithoutContext("/NodeList/76/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop76, this));
  Config::ConnectWithoutContext("/NodeList/77/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop77, this));
  Config::ConnectWithoutContext("/NodeList/78/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop78, this));
  Config::ConnectWithoutContext("/NodeList/79/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop79, this));
  }

  if (numOfNodes >= 80)
  {
```

```
  Config::ConnectWithoutContext("/NodeList/80/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop80, this));
  Config::ConnectWithoutContext("/NodeList/81/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop81, this));
  Config::ConnectWithoutContext("/NodeList/82/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop82, this));
  Config::ConnectWithoutContext("/NodeList/83/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop83, this));
  Config::ConnectWithoutContext("/NodeList/84/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop84, this));
  Config::ConnectWithoutContext("/NodeList/85/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop85, this));
  Config::ConnectWithoutContext("/NodeList/86/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop86, this));
  Config::ConnectWithoutContext("/NodeList/87/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop87, this));
  Config::ConnectWithoutContext("/NodeList/88/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop88, this));
  Config::ConnectWithoutContext("/NodeList/89/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop89, this));
  Config::ConnectWithoutContext("/NodeList/90/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop90, this));
  Config::ConnectWithoutContext("/NodeList/91/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop91, this));
  Config::ConnectWithoutContext("/NodeList/92/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop92, this));
  Config::ConnectWithoutContext("/NodeList/93/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop93, this));
  Config::ConnectWithoutContext("/NodeList/94/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop94, this));
  Config::ConnectWithoutContext("/NodeList/95/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop95, this));
  Config::ConnectWithoutContext("/NodeList/96/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop96, this));
  Config::ConnectWithoutContext("/NodeList/97/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop97, this));
  Config::ConnectWithoutContext("/NodeList/98/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop98, this));
  Config::ConnectWithoutContext("/NodeList/99/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop99, this));

Config::ConnectWithoutContext("/NodeList/100/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop100, this));

Config::ConnectWithoutContext("/NodeList/101/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop101, this));

Config::ConnectWithoutContext("/NodeList/102/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop102, this));

Config::ConnectWithoutContext("/NodeList/103/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop103, this));

Config::ConnectWithoutContext("/NodeList/104/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop104, this));

Config::ConnectWithoutContext("/NodeList/105/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop105, this));

Config::ConnectWithoutContext("/NodeList/106/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop106, this));

Config::ConnectWithoutContext("/NodeList/107/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop107, this));

Config::ConnectWithoutContext("/NodeList/108/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop108, this));

Config::ConnectWithoutContext("/NodeList/109/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop109, this));

Config::ConnectWithoutContext("/NodeList/110/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop110, this));

Config::ConnectWithoutContext("/NodeList/111/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop111, this));
```

```
Config::ConnectWithoutContext("/NodeList/112/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop112, this));

Config::ConnectWithoutContext("/NodeList/113/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop113, this));

Config::ConnectWithoutContext("/NodeList/114/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop114, this));

Config::ConnectWithoutContext("/NodeList/115/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop115, this));

Config::ConnectWithoutContext("/NodeList/116/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop116, this));

Config::ConnectWithoutContext("/NodeList/117/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop117, this));

Config::ConnectWithoutContext("/NodeList/118/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop118, this));

Config::ConnectWithoutContext("/NodeList/119/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop119, this));

Config::ConnectWithoutContext("/NodeList/120/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop120, this));

Config::ConnectWithoutContext("/NodeList/121/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop121, this));

Config::ConnectWithoutContext("/NodeList/122/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop122, this));

Config::ConnectWithoutContext("/NodeList/123/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop123, this));

Config::ConnectWithoutContext("/NodeList/124/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop124, this));

Config::ConnectWithoutContext("/NodeList/125/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop125, this));

Config::ConnectWithoutContext("/NodeList/126/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop126, this));

Config::ConnectWithoutContext("/NodeList/127/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop127, this));

Config::ConnectWithoutContext("/NodeList/128/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop128, this));

Config::ConnectWithoutContext("/NodeList/129/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop129, this));

Config::ConnectWithoutContext("/NodeList/130/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop130, this));

Config::ConnectWithoutContext("/NodeList/131/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop131, this));

Config::ConnectWithoutContext("/NodeList/132/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop132, this));

Config::ConnectWithoutContext("/NodeList/133/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop133, this));

Config::ConnectWithoutContext("/NodeList/134/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop134, this));

Config::ConnectWithoutContext("/NodeList/135/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop135, this));

Config::ConnectWithoutContext("/NodeList/136/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop136, this));
```

```
Config::ConnectWithoutContext("/NodeList/137/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop137, this));

Config::ConnectWithoutContext("/NodeList/138/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop138, this));

Config::ConnectWithoutContext("/NodeList/139/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop139, this));

Config::ConnectWithoutContext("/NodeList/140/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop140, this));

Config::ConnectWithoutContext("/NodeList/141/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop141, this));

Config::ConnectWithoutContext("/NodeList/142/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop142, this));

Config::ConnectWithoutContext("/NodeList/143/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop143, this));

Config::ConnectWithoutContext("/NodeList/144/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop144, this));

Config::ConnectWithoutContext("/NodeList/145/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop145, this));

Config::ConnectWithoutContext("/NodeList/146/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop146, this));

Config::ConnectWithoutContext("/NodeList/147/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop147, this));

Config::ConnectWithoutContext("/NodeList/148/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop148, this));

Config::ConnectWithoutContext("/NodeList/149/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop149, this));

Config::ConnectWithoutContext("/NodeList/150/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop150, this));

Config::ConnectWithoutContext("/NodeList/151/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop151, this));

Config::ConnectWithoutContext("/NodeList/152/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop152, this));

Config::ConnectWithoutContext("/NodeList/153/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop153, this));

Config::ConnectWithoutContext("/NodeList/154/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop154, this));

Config::ConnectWithoutContext("/NodeList/155/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop155, this));

Config::ConnectWithoutContext("/NodeList/156/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop156, this));

Config::ConnectWithoutContext("/NodeList/157/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop157, this));

Config::ConnectWithoutContext("/NodeList/158/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop158, this));

Config::ConnectWithoutContext("/NodeList/159/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
MakeCallback(&CogMan::PhyRxDrop159, this));
  }


  // sent packets
  Config::ConnectWithoutContext("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyTxBegin",
MakeCallback(&CogMan::SendPacket, this));
  // forwarded packets
  //    NodeContainer::Iterator nodesBegin = allNodes.Begin();
  //NodeContainer::Iterator nodesEnd   = allNodes.End();
  //while (nodesBegin < nodesEnd)
  //{
  //    nodesBegin++;
  //    Ptr<Ipv4L3Protocol> m_protocol =  allNodes.Get(0)->GetObject<Ipv4L3Protocol>();
  //    m_protocol->TraceConnectWithoutContext("UnicastForward",
MakeCallback(&CogMan::ForwardPacket, this));

    Ptr<Ipv4L3Protocol> m_protocol =  allNodes.Get(1)->GetObject<Ipv4L3Protocol>();
    m_protocol->TraceConnectWithoutContext("UnicastForward",
MakeCallback(&CogMan::ForwardPacket, this));

    /*  This is callback stuff - kept just in case
    Packet::EnableMetadata();
    Config::Connect("/NodeListDevices//Queue/Enqueue",
MakeCallback(&CognitiveMANET::LogDevQueueEnqueue,this));
    Config::Connect("/NodeList/Devices//Queue/Dequeue",
MakeCallback(&CognitiveMANET::LogDevQueueEnqueue,this));
    Config::Connect("/NodeList/Devices//Queue/Drop",
MakeCallback(&CognitiveMANET::LogDevQueueEnqueue,this));
    */

//Config::ConnectWithoutContext("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/MacTxDrop",
    //                                                  MakeCallback(&CogMan::MacTxDrop,
this));

//Config::ConnectWithoutContext("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDrop",
    //                                                  MakeCallback(&CogMan::PhyRxDrop,
this));

//Config::ConnectWithoutContext("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyTxDrop",
    //                                                  MakeCallback(&CogMan::PhyTxDrop,
this));


  //  std::string context2= "/NodeList/*/ApplicationList/*/$ns3::OnOffApplication/Tx";
  //  Config::Connect (context2, MakeCallback(&CogMan::SendPacket,this));
}


//  COUNTERS

void
CogMan::RecordSendPacketInformation(Ptr<const Packet> p)
{
  sendPacketTotal++;

  std::stringstream temp;
  p->Print(temp);
  string packetHeader = temp.str();

  if (packetHeader.find("Payload") != std::string::npos)
  { // found a payload
    if (packetHeader.find("10.1") != std::string::npos)
    {
      sendDsdvDataPacket++;
    }
    if (packetHeader.find("10.2") != std::string::npos)
    {
      sendAodvDataPacket++;
    }
  }
  else if (packetHeader.find("dsdv") != std::string::npos)
  {
    sendDsdvControlPacket++;
  }
  else if (packetHeader.find("aodv") != std::string::npos)
  {
    sendAodvControlPacket++;
  }
  else
  {
    //cout << "\nUnknown send packet found\n";
    //cout << packetHeader;
  }
}
```

```
void  // do NOT update sent counters anywhere other than here!
CogMan::UpdateSentPacketCounters(Ptr<const Packet> p)
{
  sendPacketTotal++;

  std::stringstream temp;
  p->Print(temp);
  string packetHeader = temp.str();

  string packetType = "";
  if (packetHeader.find("Payload") != std::string::npos)
  { // found a payload
    packetType = "Payload";
  }
  else
  {
    packetType = "control";
  }

  string protocol = "";
  if (packetHeader.find("10.1") != std::string::npos)
  {
    protocol = "dsdv";
  }
  if (packetHeader.find("10.2") != std::string::npos)
  {
    protocol = "aodv";
  }



  if (packetType == "Payload")
  { // found a payload
    if (protocol == "aodv")
    {
      sendAodvDataPacket++;
    }
    else if (protocol == "dsdv")
    {
      sendDsdvDataPacket++;
    }
    else
    {
      cout << "STOP! --- We have no idea what protocol SENT this DATA packet.";
    }
  }
  else if (packetType == "control")
  {
    if (protocol == "aodv")
    {
      sendAodvControlPacket++;
    }
    else if (protocol == "dsdv")
    {
      sendDsdvControlPacket++;
    }
    else
    {
      cout << "STOP! --- We have no idea what protocol SENT this CONTROL packet.";
    }
  }
  else
  {
    cout << "STOP! --- We have no idea what type of SENT packet this is.";
  }
}

void  // do NOT update dropped counters anywhere other than here!
CogMan::UpdateDroppedPacketCounters(int nodeId, string packetType)
{
  string protocol = "";

  // deal with totals
  dropPacketTotal++;
  if (nodeId < numOfNodes)
  {
    protocol = "aodv";
    dropAodvTotal++;
  }
  else
  {
    protocol = "dsdv";
    dropDsdvTotal++;
  }

  if (currentProtocol == protocol)
  {
    dropCurrentProtocol++;
  }

  if (protocol == "aodv")
  {
    if (packetType == "Clear to send")
    {
      dropAodvControlPacket++;
      dropAodvCTS++;
    }
    if (packetType == "AODV Route Request")
    {
      dropAodvControlPacket++;
      dropAodvRQ++;
    }
    if (packetType == "AODV Route Reply")
    {
      dropAodvControlPacket++;
      dropAodvMAC_RR++;
    }
    if (packetType == "MAC Resolution Protocol Request")
    {
      dropAodvControlPacket++;
      dropAodvRQ++;
    }
    if (packetType == "MAC Resolution Protocol Reply")
    {
      dropAodvControlPacket++;
      dropAodvRR++;
    }
    if (packetType == "broadcast")
    {
      dropAodvDataPacket++;
      dropAodvFF++;
    }
    if (packetType == "unknown route")
    {
      dropAodvDataPacket++;
      dropAodvUR++;
    }
    if (packetType.find("intended for")!=std::string::npos)
    {
      dropAodvDataPacket++;
      dropAodvNFM++; //Not For Me
    }
    if (packetType == "payload for 14 ???")
    {
      dropAodvDataPacket++;
      dropAodvOther++;
    }
  }
  else
  {
    if (packetType == "Clear to send")
    {
      dropDsdvControlPacket++;
      dropDsdvCTS++;
    }
    if (packetType == "DSDV Route Request")
    {
      dropDsdvControlPacket++;
      dropDsdvRQ++;
    }
    if (packetType == "DSDV Route Reply")
    {
      dropDsdvControlPacket++;
      dropDsdvMAC_RR++;
    }
    if (packetType == "MAC Resolution Protocol Request")
    {
      dropDsdvControlPacket++;
      dropDsdvRQ++;
    }
    if (packetType == "MAC Resolution Protocol Reply")
    {
      dropDsdvControlPacket++;
      dropDsdvRR++;
    }
    if (packetType == "broadcast")
    {
      dropDsdvDataPacket++;
      dropDsdvFF++;
    }
    if (packetType == "unknown route")
    {
      dropDsdvDataPacket++;
      dropDsdvUR++;
    }
    if (packetType.find("intended for")!=std::string::npos)
    {
      dropDsdvDataPacket++;
      dropDsdvNFM++; //Not For Me
    }
    if (packetType == "payload for 14 ???")
    {
      dropDsdvDataPacket++;
      dropDsdvOther++;
    }
  }
}

//  PRINT INFORMATION OR STATISTICS


//  Output Data Files

void
CogMan::WriteVectorOfMemorySystemStateToDisk(string theFilename)
{
  Print("WriteVectorOfMemorySystemStateToDisk",1);
  ofstream systemStateFile;                             //------------------------------------//
  systemStateFile.open (theFilename + ".csv");        //  A vector called theDistances     //

  string header = "sNode, mNode, time, denisty,";
  header +=       "protocol, AODV D, DSDV D,";
  header +=       "AODV R, DSDV R, channel, Switching per 5s\n";
  systemStateFile << header;
                                                      //                                   //
  vector<memorySystemState>::iterator vectorIndex;    //  The structure has variables      //
  vectorIndex = systemState.begin();                  //    nodeId      int                //
                                                      //    time        double             //
  while (vectorIndex < systemState.end())             //    x           double             //
  {                                                   //    y           double             //
    systemStateFile << vectorIndex->sNode  << ",";    //-----------------------------------//
    systemStateFile << vectorIndex->mNode  << ",";
    systemStateFile << vectorIndex->time           << ",";
    systemStateFile << vectorIndex->density        << ",";
    systemStateFile << vectorIndex->protocol       << ",";
    systemStateFile << vectorIndex->droppedPacketsAodv << ",";
    systemStateFile << vectorIndex->droppedPacketsDsdv << ",";
    systemStateFile << vectorIndex->receivedPacketsAodv << ",";
    systemStateFile << vectorIndex->receivedPacketsDsdv << ",";
    systemStateFile << vectorIndex->channel        << ",";
    systemStateFile << vectorIndex->receivedPacketsSwitching << "\n";
    vectorIndex++;
  }
  systemStateFile.close();
  Print("WriteVectorOfMemorySystemStateToDisk",2);
}

void
CogMan::WriteVectorOfManagementNodeMemoryToDisk(string theFilename)
{
  Print("WriteVectorOfManagementNodeMemoryToDisk",1);
  ofstream mNodeMemoryFile;                                          //-----------------------
-----------//
  mNodeMemoryFile.open (theFilename + ".csv");                       //  A vector called
theDistances    //

  string header = "Time,Source,Manager,Drop Rate,";
  header +=       "Receive Rate, Send Rate, Density,";
  header +=       "Channel, Routing Protocol, Action, ";
  header +=       "Progress, Backoff Time, Performance\n";
  mNodeMemoryFile << header;
                                                                          //
//
  vector<cognitivity>::iterator vectorIndex;                   //  The structure has variables
//
  vectorIndex = managementNodeMemory.begin();                                     //    nodeId
int               //
                                                                       //    time
double            //
  while (vectorIndex < managementNodeMemory.end())                                  //    x
double            //
  {                                                                       //    y
double            //
    mNodeMemoryFile << vectorIndex->atTime          << ",";                         //---------
-------------------------//
    mNodeMemoryFile << vectorIndex->sNode           << ",";
    mNodeMemoryFile << vectorIndex->mNode           << ",";
    mNodeMemoryFile << vectorIndex->dropRate        << ",";
    mNodeMemoryFile << vectorIndex->receiveRate     << ",";
    mNodeMemoryFile << vectorIndex->sendRate        << ",";
    mNodeMemoryFile << vectorIndex->density         << ",";
    mNodeMemoryFile << vectorIndex->channel         << ",";
    mNodeMemoryFile << vectorIndex->routingProtocol << ",";
    mNodeMemoryFile << vectorIndex->request         << ",";
    mNodeMemoryFile << vectorIndex->requestProgress << ",";
    mNodeMemoryFile << vectorIndex->backOffTime     << ",";
    mNodeMemoryFile << vectorIndex->performance     << "\n";
    vectorIndex++;
  }
  mNodeMemoryFile.close();
  Print("WriteVectorOfManagementNodeMemoryToDisk",2);
}

void
CogMan::WriteAODVvsDSDVStats(double interval)  // the interval to split packets into
{
  ofstream aodvDsdvStats;                                  //----------------------------------
//
  aodvDsdvStats.open (fn + "dsdvAodv.csv");               //  A vector called packetReceived
// (fn + "netAnim.xml")

  vector<AllPackets>::iterator vectorIndex;
  vectorIndex = allPackets.begin();

  vector<IpAddresses>::iterator indexCurrentProtocolS;
  vector<IpAddresses>::iterator indexCurrentProtocolR;
  vector<IpAddresses>::iterator indexCurrentProtocolD;

  indexCurrentProtocolS = packetSentByCurrentProtocol.begin();
  indexCurrentProtocolR = packetReceivedByCurrentProtocol.begin();
  indexCurrentProtocolD = packetDroppedByCurrentProtocol.begin();

  double timeUntil = interval;

  int aodvS  = 0;  int aodvR  = 0;
  int aodvD  = 0;  int dsdvS  = 0;
  int dsdvR  = 0;  int dsdvD  = 0;
  int totalA = 0;  int totalD = 0;
  int totalS = 0;  int goodA  = 0;
  int goodD  = 0;  int bad    = 0;

  cout << " WriteAODVvsDSDVStats has been called and will check everything what happened every
"<< interval<<" seconds" <<endl;

  string header = "time, AODV S, AODV R, AODV D, DSDV S, DSDV R, DSDV D, Swi S, Swi R, Swi D,
\n";
  aodvDsdvStats << header;
  while (vectorIndex < allPackets.end())
  {
    string protocol = GetProtocol(vectorIndex->destination);
    if (protocol == "aodv")
    {
      if (vectorIndex->action == 'S')
      {
        aodvS++;
      }
      else if (vectorIndex->action == 'R')
      {
        aodvR++;
      }
      else if (vectorIndex->action == 'D')
      {
        aodvD++;
      }
    }
    else if (protocol == "dsdv")
    {
      if (vectorIndex->action == 'S')
      {
        dsdvS++;
      }
      else if (vectorIndex->action == 'R')
      {
        dsdvR++;
      }
      else if (vectorIndex->action == 'D')
      {
        dsdvD++;
      }
    }
    if (vectorIndex->time >= timeUntil)
    {
      int routeProtocolS = 0;
      int routeProtocolR = 0;
      int routeProtocolD = 0;

                       while (indexCurrentProtocolS->time < timeUntil && indexCurrentProtocolS
< packetSentByCurrentProtocol.end())
                       {
                              routeProtocolS++;
                              indexCurrentProtocolS++;
                       }

                       while (indexCurrentProtocolR->time < timeUntil && indexCurrentProtocolR
< packetReceivedByCurrentProtocol.end())
                       {
                              routeProtocolR++;
                              indexCurrentProtocolR++;
                       }

                       while (indexCurrentProtocolD->time < timeUntil && indexCurrentProtocolD
< packetDroppedByCurrentProtocol.end())
                       {
                              routeProtocolD++;
                              indexCurrentProtocolD++;
                       }

      if (aodvR > dsdvR)
      {
        goodA++;
      }
      else if (dsdvR > aodvR)
      {
        goodD++;
      }
      else
      {
        bad++;
      }

      aodvDsdvStats << timeUntil << "," << aodvS << "," << aodvR << "," << aodvD << ","
                                        << dsdvS << "," << dsdvR << "," << dsdvD << ","
                                        << routeProtocolS << "," << routeProtocolR << ","
                                        << routeProtocolD << "," << "\n";

      timeUntil = timeUntil + interval;

      totalA = totalA + aodvR;
      totalD = totalD + dsdvR;
      totalS = totalS + routeProtocolR;
      aodvS = 0;  aodvR = 0;  aodvD = 0;
      dsdvS = 0;  dsdvR = 0;  dsdvD = 0;
    }
    vectorIndex++;
  }

  aodvDsdvStats.close();

  cout << " finally, Aodv Received " << totalA << " Dsdv received " << totalD << " and 
Switching Protocols received " <<
  totalS <<endl;
  if (totalS > totalA && totalS > totalD)
  {
    cout << " WIN" <<endl;
  }
  else if (totalS >= totalA && totalS >= totalD)
  {
    cout << " Not bad, we have picked up best protocol" <<endl;
  }
  else if ((totalS >= totalA && totalA < totalD) || (totalS >= totalD && totalD < totalA))
  {
    cout << "at least we have not spoiled much :-( as we were doing better than the worst 
protocol " << endl;
  }
  else
  {
    cout << "opps! we are the worst creatures spoiling good things " <<endl;
  }


  cout << "Aodv was good " << goodA << " times, and since each time is "<< interval << " 
seconds, we have been good " <<
  goodA*interval << " seconds out of " << simulationTime << "seconds. Or, " << (goodA*interval 
/simulationTime) * 100 << " % of simulation time. " << endl;
  cout << "Dsdv was good " << goodD << " times, and since each time is " << interval << " 
seconds, we have been good " <<
  goodD*interval << " seconds out of " << simulationTime << "seconds.Or, " << (goodD*interval 
/simulationTime) * 100 << " % of simulation time. " << endl;
}

void
CogMan::WriteVectorOfNodePositionsToDisk(string theFilename)
{
  Print("WriteVectorOfNodePositionsToDisk",1);
  ofstream distancesFile;                             //----------------------------------//
  distancesFile.open (theFilename + ".csv");          //  A vector called theDistances    //
                                                      //                                  //
  vector<currentPosition>::iterator vectorIndex;      //  The structure has variables     //
  vectorIndex = myPositions.begin();                  //    nodeId      int               //
                                                      //    time        double            //
  while (vectorIndex < myPositions.end())             //    x           double            //
  {                                                   //    y           double            //
    distancesFile << vectorIndex->nodeId  << ",";     //----------------------------------//
    distancesFile << vectorIndex->time    << ",";
    distancesFile << vectorIndex->x       << ",";
    distancesFile << vectorIndex->y       << "\n";
    vectorIndex++;
  }
  distancesFile.close();
  Print("WriteVectorOfNodePositionsToDisk",2);
}

void
CogMan::WriteVectorOfAllPacketsToDisk(string theFilename)
{
  Print("WriteVectorOfReceivedPacketsToDisk",1);
  ofstream allPacketsFile;                                  //---------------------------------
-//
  allPacketsFile.open (theFilename + ".csv");               //  A vector called packetReceived 
//
```

```
                                                           //
//
  vector<AllPackets>::iterator vectorIndex;               //  The structure has variables
//
  vectorIndex = allPackets.begin();                        //    source      Ipv4Address
//

  string header = "Source, Destination, Time, Action\n";     //    destination Ipv4Address
//
  allPacketsFile << header;
  while (vectorIndex < allPackets.end())               //    time        double            //
  {                                                        //---------------------------------
-//
    allPacketsFile << vectorIndex->source      << ",";
    allPacketsFile << vectorIndex->destination << ",";
    allPacketsFile << vectorIndex->time        << ",";
    allPacketsFile << vectorIndex->action       << "\n";
    vectorIndex++;
  }
  allPacketsFile.close();
  Print("WriteVectorOfReceivedPacketsToDisk",2);
}

void
CogMan::WriteVectorOfReceivedPacketsToDisk(string theFilename)
{
  Print("WriteVectorOfReceivedPacketsToDisk",1);
  ofstream receivedPacketsFile;                            //---------------------------------
-//
  receivedPacketsFile.open (theFilename + ".csv");         //  A vector called packetReceived
//
                                                           //
//
  vector<IpAddresses>::iterator vectorIndex;               //  The structure has variables
//
  vectorIndex = packetReceived.begin();                    //    source      Ipv4Address
//
                                                           //    destination Ipv4Address
//
  while (vectorIndex < packetReceived.end())               //    time        double
//
  {                                                        //---------------------------------
-//
    receivedPacketsFile << vectorIndex->source      << ",";
    receivedPacketsFile << vectorIndex->destination << ",";
    receivedPacketsFile << vectorIndex->time       << "\n";
    vectorIndex++;
  }
  receivedPacketsFile.close();
  Print("WriteVectorOfReceivedPacketsToDisk",2);
}


void
CogMan::WriteVectorOfDroppedPacketsToDisk(string theFilename)
{
  Print("WriteVectorOfDroppedPacketsToDisk",1);
  ofstream droppedPacketsFile;                             //---------------------------------
-//
  droppedPacketsFile.open (theFilename + ".csv");          //  A vector called packetDropped
//
                                                           //
//
  vector<IpAddresses>::iterator vectorIndex;               //  The structure has variables
//
  vectorIndex = packetDropped.begin();                     //    source      Ipv4Address
//
                                                           //    destination Ipv4Address
//
  while (vectorIndex < packetDropped.end())                //    time        double
//
  {                                                        //---------------------------------
-//
    droppedPacketsFile << vectorIndex->source      << ",";
    droppedPacketsFile << vectorIndex->destination << ",";
    droppedPacketsFile << vectorIndex->time       << "\n";
    vectorIndex++;
  }
  droppedPacketsFile.close();
  Print("WriteVectorOfDroppedPacketsToDisk",2);
}

void
CogMan::WriteVectorOfSentPacketsToDisk(string theFilename)
{
  Print("WriteVectorOfSentPacketsToDisk",1);
  ofstream sentPacketsFile;                       //-----------------------------------//
  sentPacketsFile.open (theFilename + ".csv");  //  A vector called packetSent       //
                                                //                                 //
  vector<IpAddresses>::iterator vectorIndex;    //  The structure has variables      //
  vectorIndex = packetSent.begin();             //    source      Ipv4Address        //
                                                //    destination Ipv4Address        //
  while (vectorIndex < packetSent.end())        //    time        double             //
  {                                             //-----------------------------------//
    sentPacketsFile << vectorIndex->source      << ",";
    sentPacketsFile << vectorIndex->destination << ",";
    sentPacketsFile << vectorIndex->time       << "\n";
    vectorIndex++;
  }
  sentPacketsFile.close();
  Print("WriteVectorOfSentPacketsToDisk",2);
}

void
CogMan::WriteVectorOfSentPacketsBCPToDisk(string theFilename)
{
  Print("WriteVectorOfSentPacketsToDisk",1);
  ofstream sentPacketsBCPFile;                            //----------------------------------
//
  sentPacketsBCPFile.open (theFilename + ".csv");         //  A vector called packetSent
//
                                                          //
//
  vector<IpAddresses>::iterator vectorIndex;              //  The structure has variables
//
  vectorIndex = packetSentByCurrentProtocol.begin();      //    source      Ipv4Address
//
                                                          //    destination Ipv4Address
//
  while (vectorIndex < packetSentByCurrentProtocol.end()) //    time        double
//
  {                                                       //----------------------------------
//
    sentPacketsBCPFile << vectorIndex->source      << ",";
    sentPacketsBCPFile << vectorIndex->destination << ",";
    sentPacketsBCPFile << vectorIndex->time       << "\n";
    vectorIndex++;
  }
  sentPacketsBCPFile.close();
  Print("WriteVectorOfSentPacketsToDisk",2);
}

void
CogMan::WriteVectorOfPacketRatesToDisk(string theFilename)
{
  Print("WriteVectorOfPacketRatesToDisk",1);
  ofstream sentPacketRateFile;                             //---------------------------------
--//
  sentPacketRateFile.open (theFilename + ".csv");          //  A vector called packetRates
//
                                                           //  to store the current packetRate
//
  vector<packetRate>::iterator vectorIndex;                //
//
  vectorIndex = packetRates.begin();                       //  The structure has variables
//
                                                           //    time              double
//
  string header = "Time,Source, Sent,Received,Dropped\n";  //    sent              double
//
  sentPacketRateFile << header;                            //    received          double
//
                                                           //    dropped           double
//
```

```
  while (vectorIndex < packetRates.end())                  //---------------------------------
--//
  {
    sentPacketRateFile << vectorIndex->time     << ",";
    sentPacketRateFile << vectorIndex->source   << ",";
    sentPacketRateFile << vectorIndex->sent     << ",";
    sentPacketRateFile << vectorIndex->received << ",";
    sentPacketRateFile << vectorIndex->dropped  << "\n";
    vectorIndex++;
  }
  sentPacketRateFile.close();
  Print("WriteVectorOfPacketRatesToDisk",2);
}

void
CogMan::WriteVectorFlowMonitorToDisk(string theFilename)
{
  Print("WriteVectorFlowMonitorToDisk",1);
  ofstream flowMonFile;                                    //---------------------------------
---//
  flowMonFile.open (theFilename + ".csv");                 //  A vector called myDataFlows
//
                                                           //  to store many flowInformation
//
  vector<flowInformation>::iterator vectorIndex;           //
//
  vectorIndex = myDataFlows.begin();                       //  The structure has variables
//
  while (vectorIndex < myDataFlows.end())                  //    flowId          unsigned int
//
  {                                                        //    sourceAddress   Ipv4Address
//
    flowMonFile << vectorIndex->flowId             << ","; //    destinationNode Ipv4Address
//
    flowMonFile << vectorIndex->sourceAddress      << ","; //    txPackets       uint32_t
//
    flowMonFile << vectorIndex->destinationAddress << ","; //    txBytes         uint64_t
//
    flowMonFile << vectorIndex->txPackets          << ","; //    rxOffered       double
//
    flowMonFile << vectorIndex->txBytes            << ","; //    rxPackets       uint64_t
//
    flowMonFile << vectorIndex->rxOffered          << ","; //    rxBytes         uint64_t
//
    flowMonFile << vectorIndex->rxPackets          << ","; //    throughput      double
//
    flowMonFile << vectorIndex->rxBytes            << ","; //---------------------------------
---//
    flowMonFile << vectorIndex->throughput         << "\n";
    vectorIndex++;
  }
  flowMonFile.close();
  Print("WriteVectorFlowMonitorToDisk",2);
}

// End of Output Data Files




void
CogMan::SendPacket(Ptr<const Packet> p)  // this is my function which is been call by connect
function in the code.
{
  Print("SendPacket",1);

  Ipv4Address src_ip;
  Ipv4Address des_ip;
  //The following code is based from
  //http://polythinking.wordpress.com/2012/05/30/ns-3-network-simulator-how-to-find-a-
specific-header-in-packet-in-ns-3/

  // To get a header from Ptr<Packet> packet first, copy the packet
  Ptr<Packet> q = p->Copy();

  // Use indicator to search the packet
  PacketMetadata::ItemIterator metadataIterator = q->BeginItem();
  PacketMetadata::Item item;
  while (metadataIterator.HasNext())
  {
    item = metadataIterator.Next();
    //cout << "header name: " << item.tid.GetName() << endl;

    // If we want to have an ip header
    if(item.tid.GetName() == "ns3::Ipv4Header")
    {
      Callback<ObjectBase *> constr = item.tid.GetConstructor();
      NS_ASSERT(!constr.IsNull());

      // Ptr<> and DynamicCast<> won't work here as all headers are from ObjectBase, not
Object
      ObjectBase *instance = constr();
      NS_ASSERT(instance != 0);

      Ipv4Header* ipv4Header = dynamic_cast<Ipv4Header*> (instance);
      NS_ASSERT(ipv4Header != 0);

      ipv4Header->Deserialize(item.current);

      // The ipv4Header can now obtain the source of the packet
      src_ip = ipv4Header->GetSource();
      des_ip = ipv4Header->GetDestination();

      // Finished, clear the ip header and go back
      message << "Packet Sent at " << Simulator::Now().GetSeconds() << "s\n   source      IP
Address is: " << src_ip;
      int packetInfo = 5;
      string tmp = ConvertIPAddressToString(des_ip);
      if (tmp.find("255")!=std::string::npos)
      {
        packetInfo = 42; // route discovery packet
      }
      else
      {
        if (!reportedSend)
          {
            reportedSend = true;
            cout << GetTimeNow('s') << " First data packet sent\n";
          }
      }

      Print (message.str(), packetInfo);
      message << "  destination IP Address is: " << des_ip;
      Print (message.str(), packetInfo);

      RecordSentPacket(src_ip, des_ip);
      UpdateSentPacketCounters(p);

      break;
    }
  }
  Print("SendPacket",2);
}

void
CogMan::Print(string theOutput, int type)
{
  if (functionMessageStart && type==1)
  {
    cout << "At " << Simulator::Now().GetSeconds() << "s Entered " << theOutput << " method."
<< endl;
  }
  if (functionMessageEnd && type==2)
  {
    cout << "At " <<  Simulator::Now().GetSeconds() << "s Left " << theOutput << " method." <<
endl;
  }
  if (outputVariableValue && type==3)
  {
    cout << " " << theOutput << endl;
  }

  if (packetDropInfo && type==4)
  {
    cout << " " << theOutput << endl;
  }

  if (packetRouteInfo && type==42)
  {
    cout << " " << theOutput << endl;
  }

  if (packetSentInfo && type==5)
  {
    cout << " " << theOutput << endl;
  }

  if (packetRecdInfo && type==6)
  {
    cout << " " << theOutput << endl;
  }

  if (distanceInfo && type==7)
  {
    cout << " " << theOutput << endl;
  }

  if (deviceInfo && type==8)
  {
    cout << " " << theOutput << endl;
  }

  if (addressInfo && type==9)
  {
    cout << " " << theOutput << endl;
  }

  if (trafficFlowInfo && type==10)
  {
    cout << " " << theOutput << endl;
  }

  if (electionInfo && type==11)
  {
    cout << " " << theOutput << endl;
  }

  if (destinationInfo && type==12)
  {
    cout << " " << theOutput << endl;
  }

  if (recordPositionInfo && type==13)
  {
    cout << " " << theOutput << endl;
  }

  if (remainingEnergyInfo && type==14)
  {
    cout << " " << theOutput << endl;
  }

  if (allDistancesFromSourceInfo && type==15)
  {
    cout << " " << theOutput << endl;
  }

  if (neigbourDistancesFromSourceInfo && type==16)
  {
    cout << " " << theOutput << endl;
  }



  if (nodeInfo && type==18)
  {
    cout << " " << theOutput << endl;
  }


  if (debugInSteps && numberOfSteps <= debugStepCount)
  {
    debugStepCount=0;
    //cout << "...paused <ENTER> to continue...";
   // cin.get();
  }
  debugStepCount++;
  message.str( std::string() );
  message.clear();
}

void
CogMan::LogStuff()
{
  Print("LogStuff",1);
  LogComponentEnable("RegularWifiMac", LOG_LEVEL_ALL);
  LogComponentEnable("WifiPhy", LOG_LEVEL_ALL);
  LogComponentEnable("WifiRemoteStationManager", LOG_LEVEL_ALL);
  LogComponentEnable("WifiPhyStateHelper", LOG_LEVEL_ALL);
  LogComponentEnable("WifiChannel", LOG_LEVEL_ALL);
  LogComponentEnable("SupportedRates", LOG_LEVEL_ALL);
  LogComponentEnable("StaWifiMac", LOG_LEVEL_ALL);
  LogComponentEnable("RraaWifiManager", LOG_LEVEL_ALL);
  LogComponentEnable("PropagationLossModel", LOG_LEVEL_ALL);
  LogComponentEnable("RegularWifiMac", LOG_LEVEL_ALL);
  // To see entire list of what can be logged uncomment next line
  // LogComponentEnable("ShowMeAll", LOG_LEVEL_ALL);
  Print("LogStuff",2);
}

void
CogMan::PhyTxDrop(Ptr<const Packet> p)
{
  Print("PhyTxDrop",1);

  bool Payload = RecordDroppedPacketInformation(p);

  if (Payload)
  {
    Ipv4Address src_ip;
    Ipv4Address des_ip;
    //The following code is based from
    //http://polythinking.wordpress.com/2012/05/30/ns-3-network-simulator-how-to-find-a-
specific-header-in-packet-in-ns-3/

    // To get a header from Ptr<Packet> packet first, copy the packet
    Ptr<Packet> q = p->Copy();

    // Use indicator to search the packet
    PacketMetadata::ItemIterator metadataIterator = q->BeginItem();
    PacketMetadata::Item item;
    while (metadataIterator.HasNext())
    {
      item = metadataIterator.Next();
      //cout << "header name: " << item.tid.GetName() << endl;

      // If we want to have an ip header
      if(item.tid.GetName() == "ns3::Ipv4Header")
      {
        Callback<ObjectBase *> constr = item.tid.GetConstructor();
        NS_ASSERT(!constr.IsNull());

        // Ptr<> and DynamicCast<> won't work here as all headers are from ObjectBase, not
Object
        ObjectBase *instance = constr();
        NS_ASSERT(instance != 0);

        Ipv4Header* ipv4Header = dynamic_cast<Ipv4Header*> (instance);
        NS_ASSERT(ipv4Header != 0);

        ipv4Header->Deserialize(item.current);

        // The ipv4Header can now obtain the source of the packet
        src_ip = ipv4Header->GetSource();
        des_ip = ipv4Header->GetDestination();

        // Finished, clear the ip header and go back
        message << "Packet Dropped at " << Simulator::Now().GetSeconds() << "s\n   source
IP Address is: " << src_ip;

        int packetInfo = 4;

        string tmp = ConvertIPAddressToString(des_ip);
        if (tmp.find("255")!=std::string::npos)
        {
          packetInfo = 42; // route discovery packet
        }

        Print (message.str(), packetInfo);
        message << "  destination IP Address is: " << des_ip;
        Print (message.str(), packetInfo);
        RecordDroppedPacket(src_ip, des_ip);
        break;
      }
    }
  }
  Print("PhyTxDrop",2);
}

void
CogMan::MacTxDrop(Ptr<const Packet> p)
{
  Print("MacTxDrop",1);

  bool Payload = RecordDroppedPacketInformation(p);

  if (Payload)
  {

    Ipv4Address src_ip;
    Ipv4Address des_ip;

    // To get a header from Ptr<Packet> packet first, copy the packet
    Ptr<Packet> q = p->Copy();

    // Use indicator to search the packet
    PacketMetadata::ItemIterator metadataIterator = q->BeginItem();
    PacketMetadata::Item item;
    while (metadataIterator.HasNext())
    {

      item = metadataIterator.Next();

      // If we want to have an ip header
      if(item.tid.GetName() == "ns3::Ipv4Header")
      {
        Callback<ObjectBase *> constr = item.tid.GetConstructor();
        NS_ASSERT(!constr.IsNull());

        // Ptr<> and DynamicCast<> won't work here as all headers are from ObjectBase, not
Object
        ObjectBase *instance = constr();
        NS_ASSERT(instance != 0);

        Ipv4Header* ipv4Header = dynamic_cast<Ipv4Header*> (instance);
        NS_ASSERT(ipv4Header != 0);

        ipv4Header->Deserialize(item.current);

        // The ipv4Header can now obtain the source of the packet
        src_ip = ipv4Header->GetSource();
        des_ip = ipv4Header->GetDestination();

        // Finished, clear the ip header and go back
        message << "Packet Dropped at " << Simulator::Now().GetSeconds() << "s\n   source
IP Address is: " << src_ip;

        int packetInfo = 4;

        string tmp = ConvertIPAddressToString(des_ip);
        if (tmp.find("255")!=std::string::npos)
        {
          packetInfo = 42; // route discovery packet
        }
        Print (message.str(), packetInfo);
        message << "  destination IP Address is: " << des_ip;
        Print (message.str(), packetInfo);

        RecordDroppedPacket(src_ip, des_ip);
        break;
      }
    }
  }
  Print("MacTxDrop",2);
}

void
CogMan::PhyRxDrop(Ptr<const Packet> p)
{
  Print("PhyRxDrop",1);

  bool Payload = RecordDroppedPacketInformation(p);

  if (Payload)
  {
    message << "PhyRxDrop";
    Print(message.str(),1);

    Ipv4Address src_ip;
    Ipv4Address des_ip;

    // To get a header from Ptr<Packet> packet first, copy the packet
    Ptr<Packet> q = p->Copy();

    // Use indicator to search the packet
    PacketMetadata::ItemIterator metadataIterator = q->BeginItem();
    PacketMetadata::Item item;
    while (metadataIterator.HasNext())
    {

      item = metadataIterator.Next();
      //    cout << "header name: " << item.tid.GetName() << endl;

      // If we want to have an ip header
      if(item.tid.GetName() == "ns3::Ipv4Header")
      {
        Callback<ObjectBase *> constr = item.tid.GetConstructor();
        NS_ASSERT(!constr.IsNull());

        // Ptr<> and DynamicCast<> won't work here as all headers are from ObjectBase, not 
Object
        ObjectBase *instance = constr();
        NS_ASSERT(instance != 0);

        Ipv4Header* ipv4Header = dynamic_cast<Ipv4Header*> (instance);
        NS_ASSERT(ipv4Header != 0);

        ipv4Header->Deserialize(item.current);

        // The ipv4Header can now obtain the source of the packet
        src_ip = ipv4Header->GetSource();
        des_ip = ipv4Header->GetDestination();

        // Finished, clear the ip header and go back
        message << "Packet Dropped at " << Simulator::Now().GetSeconds() << "s\n   source 
IP Address is: " << src_ip;

        int packetInfo = 4;

        string tmp = ConvertIPAddressToString(des_ip);
        if (tmp.find("255")!=std::string::npos)
        {
          packetInfo = 42; // route discovery packet
        }

        Print (message.str(), packetInfo);
        message << "  destination IP Address is: " << des_ip;
        Print (message.str(), packetInfo);
        RecordDroppedPacket(src_ip, des_ip);
        break;
      }
    }
  }



  /*  uint8_t *buffer = new uint8_t[p->GetSize()];
  p->CopyData (buffer, p->GetSize());
  string data = string((char*)buffer);

  stringstream os;
  p->PrintPacketTags(os);
  cout << os;

  uint8_t *buffer = new uint8_t (p->GetSize ());
  p->CopyData (buffer, p->GetSize ());

  cout << buffer; */
  Print("PhyRxDrop",2);
}

void
CogMan::ConfigureNetAmin(AnimationInterface& anim)
{

  vector<controlledNode>::iterator vectorIndex;
  vector<controlledNode>::iterator vectorEnd;

  vectorIndex = AodvSharksNode.begin();
  vectorEnd = AodvSharksNode.end();
  while (vectorIndex < vectorEnd)
  {
    anim.UpdateNodeColor (aodvNodes.Get (vectorIndex->nodeId), 100, 50, 204);
    vectorIndex++;
  }

  vectorIndex = DsdvSharksNode.begin();
  vectorEnd = DsdvSharksNode.end();
  while (vectorIndex < vectorEnd)
  {
    anim.UpdateNodeColor (dsdvNodes.Get (vectorIndex->nodeId), 50, 50, 50);
    vectorIndex++;
  }
  vectorIndex = AodvEaglesNode.begin();
  vectorEnd = AodvEaglesNode.end();
  while (vectorIndex < vectorEnd)
  {
    anim.UpdateNodeColor (aodvNodes.Get (vectorIndex->nodeId), 100, 0, 20);
    vectorIndex++;
  }
  vectorIndex = DsdvEaglesNode.begin();
  vectorEnd = DsdvEaglesNode.end();
  while (vectorIndex < vectorEnd)
  {
    anim.UpdateNodeColor (dsdvNodes.Get (vectorIndex->nodeId), 10, 250, 20);
    vectorIndex++;
  }


  for (int i=0; i<aodvControlledNodes.GetN(); i++)
  {
    anim.UpdateNodeColor (aodvControlledNodes.Get (i), 222, 64, 182);
  }


  anim.EnablePacketMetadata(true);
  // anim.SetMobilityPollInterval (Seconds (0.25));
  //  anim.EnableIpv4RouteTracking (fn + "routingTable.xml",
  //                               Seconds(0), Seconds(simulationTime),
  //                               Seconds(0.25));
}

void
CogMan::WriteInformationToDisk(string &fn)
{
  WriteVectorOfReceivedPacketsToDisk(fn + "packetsReceived");
  WriteVectorOfDroppedPacketsToDisk(fn + "packetsDropped");
  WriteVectorOfSentPacketsToDisk(fn + "packetsSent");
  WriteVectorOfSentPacketsBCPToDisk(fn + "packetsSentBCP");
  WriteVectorOfAllPacketsToDisk(fn + "packetsAll");
  //WriteVectorOfMemorySystemStateToDisk(fn + "systemStateAK");
  WriteVectorOfPacketRatesToDisk(fn + "packetRates");
  WriteVectorOfManagementNodeMemoryToDisk(fn + "managementMemory");
  //WriteVectorOfNodePositionsToDisk(fn + "nodePositions");

  WriteAODVvsDSDVStats(5.0);
}

void
CogMan::ScheduleStuff()
{
  Print("ScheduleStuff",1);

  //  NodeWalkTo (int nodeId, double xGoal, double yGoal, double stepSize (per meter), double
speed (mps))
  //  Average step size (stide) is 0.8 meters per stride
  //  Average walking speed is 1.79 meters per second

  //Simulator::Schedule (Seconds (49), &CogMan::PrintRemainingEnergy, this, 1);
  //Simulator::Schedule (Seconds (3), &CogMan::RemoveDepleatedNodes, this);


  /*  This is callback stuff - kept just in case
  Packet::EnableMetadata();
  Config::Connect("/NodeListDevices//Queue/Enqueue",
MakeCallback(&CognitiveMANET::LogDevQueueEnqueue,this));
  Config::Connect("/NodeList/Devices//Queue/Dequeue",
MakeCallback(&CognitiveMANET::LogDevQueueEnqueue,this));
  Config::Connect("/NodeList/Devices//Queue/Drop",
MakeCallback(&CognitiveMANET::LogDevQueueEnqueue,this));
  */

  //Config::ConnectWithoutContext
("/NodeList/*/DeviceList/*/$ns3::TcpL4Protocol/UnicastForward",
  //  MakeCallback (&forwardedhmm));

  //Ipv4RoutingProtocol::UnicastForwardCallback ucb = MakeCallback (&forwardedhmm);

  //flowmon->SerializeToXmlFile ("flowmonFile", false, false);
  //monitor->CheckForLostPackets ();

  //test.PrintFlowMonitorInformation(monitor);
}

void
CogMan::CreateTrafficFlowsForCongestion()
{
  Print("CreateTrafficFlowsForCongestion",1);
  // sNode, dNode, timeStart, timeStop
  int midPoint = congestionNodes.GetN() / 2;
  for (int i=0 ; i <  midPoint; i++)
  {
    myTrafficCongestionFlows.push_back ({ i, i+midPoint , 0.01, simulationTime-10 });
  }

  vector<traffic>::iterator vectorIndex;
  vectorIndex = myTrafficCongestionFlows.begin();

  while (vectorIndex < myTrafficCongestionFlows.end())
  {
    // from nodeId to nodeId starting at Time and stoping at Time
    CreateTrafficFlowForCongestion(vectorIndex->sNode,
                                   vectorIndex->dNode,
                                   vectorIndex->timeStart,
                                   vectorIndex->timeStop);
    vectorIndex++;
  }
  Print("CreateTrafficFlowsForCongestion",2);
}

void
CogMan::RemoveDepleatedNodes()
```

```
{
  Print("RemoveDepleatedNodes",1);
  // create a dead container
  NodeContainer deadNodes;
  NodeContainer liveNodes;

  NodeContainer::Iterator allNodesBegin = allNodes.Begin();
  NodeContainer::Iterator allNodesEnd = allNodes.End() - 1;

  while (allNodesBegin < allNodesEnd)
  {
    message << "Returned energy was: " << GetRemainingEnergy((*allNodesBegin)->GetId());
    Print(message.str(), 14);
    allNodesBegin++;

    if (GetRemainingEnergy((*allNodesBegin)->GetId() > 0))
    {
      liveNodes.Add(allNodes.Get((*allNodesBegin)->GetId()));
    }
    else
    {
      deadNodes.Add(allNodes.Get((*allNodesBegin)->GetId()));
    }
  }

  message << "There are " << deadNodes.GetN() << " dead nodes";
  message << "\nThere are " << liveNodes.GetN() << " live nodes";
  Print(message.str(), 14);
  Print("RemoveDepleatedNodes",2);
}

void
CogMan::PrintAodvFrequency()
{
  PointerValue tmpPhy;
  aodvDevices.Get(0)->GetAttribute("Phy", tmpPhy);
  Ptr<Object> wifiPhyLayer = tmpPhy.GetObject();
  Ptr<YansWifiPhy> yansWifiPhyLayer = wifiPhyLayer->GetObject<YansWifiPhy>();
  double frequency = yansWifiPhyLayer->GetChannelFrequencyMhz();
  cout << "\nThe frequency is: " << frequency << "Hz";
  cout << "\nThe channel number is: " << yansWifiPhyLayer->GetChannelNumber();

  //  Ptr<PropagationLossModel> current;
  //  vector<ObjectFactory>::const_iterator i = m_propagationLoss.begin;

  //  while (i != m_propagationLoss.end)
  //  {
  //    cout << "prop loss model";
  //    i++;
  //  }
}

void
CogMan::RecordPosition (int myNodeId)  // original at 191 of all2.cc
{
  Print("RecordPosition",1);
  Vector pos = GetNodePosition(myNodeId);                 //----------------------------------
//
                                                          //  A vector called myPositions
//
  myPositions.push_back({ myNodeId,                       //  to store many currentPositions
//
                          Simulator::Now().GetSeconds(),  //
//
                          pos.x,                          //  The structure has variables
//
                          pos.y                           //    nodeId      int
//
                        });                               //    time        double
//
                                                          //    x           double
//
  message << "At " << Simulator::Now().GetSeconds()       //    y           double
//
    << "s node " << myNodeId << " is at\tx: " << pos.x    //----------------------------------
//
    << "\ty: " << pos.y ;
  Print(message.str(), 13);
  message << "Vector myPositions now has " << myPositions.size() << " currentPositions.";
  Print(message.str(), 13);
  Print("RecordPosition",2);

  Simulator::Schedule (Seconds (recordPositionInterval), &CogMan::RecordPosition, this,
myNodeId);
}



void
CogMan::PrintNodePosition(int nodeId)
{
  Print("PrintNodePosition",1);
  Vector pos = GetNodePosition(nodeId);
  cout << ns3::Simulator::Now().GetMicroSeconds () <<
  " Node: " << nodeId <<
  " x=" << pos.x <<
  ", y=" << pos.y <<
  std::endl;
  Print("PrintNodePosition",2);
}







void
CogMan::PrintVectorOfDistances(vector<nodeDistance> theDistances)
{
  Print("PrintVectorOfDistances",1);
  vector<nodeDistance>::iterator vectorIndex;       //----------------------------------//
  vectorIndex = theDistances.begin();               //  A vector called theDistances    //
  while (vectorIndex < theDistances.end())          //  to store local nodeDistance     //
  {                                                 //                                  //
    cout << vectorIndex->sourceNode    << ", ";     //  The structure has variables     //
    cout << vectorIndex->neighbourNode    << ", ";  //    sourceNode      int           //
    cout << vectorIndex->distance    << ", ";       //    neighbourNode   int           //
    cout << vectorIndex->myTime    << "\n";         //    distance        double        //
    vectorIndex++;                                  //    myTime          double        //
  }                                                 //----------------------------------//
  Print("PrintVectorOfDistances",2);
}



void
CogMan::PrintNodesContainerInfo(NodeContainer myNodes, string containerName)
{
  Print("PrintNodesContainerInfo",1);
  std::cout << "Node Container Information for " << containerName;
  struct winsize w;                      // the following lines simply
  ioctl(0, TIOCGWINSZ, &w);              // count the height and width
  std::cout << "\nRows: " << w.ws_row;  // of the terminal window so
  std::cout << "\nColumns " << w.ws_col;// that we can adust column count

  w.ws_col = 94; // so that it fits in sublime window

  std::cout << std::endl << std::endl << containerName << std::endl;

  string ipAddress;                                 // string to store IP Address
  ostringstream convert;                            // stream used for the conversion
  convert << aodvInterface.GetAddress(allNodes.GetN()-1);   // get last IP address - we could
have a memory address greater but smaller in length i.e. 10.1.1.5 comes after 10.1.0.254
  ipAddress = convert.str();                        // convert to a string

  string nodeCount;                                 // string to store max node number
  ostringstream convert2;                           // stream used for the conversion
  convert2 << myNodes.GetN();                       // get total number of nodes
  nodeCount = convert2.str();                       // convert to string

  string nodeMemoryAddress;                         // string which will contain the Last Node
Pointer
  ostringstream convert3;                           // stream used for the conversion
  convert3 << myNodes.Get(myNodes.GetN()-1);        // get memory address of last node
  nodeMemoryAddress = convert3.str();               // convert to string

  uint32_t longestElementLength = nodeCount.length()+5;  // 5 is for "Node "

  if (ipAddress.length() > longestElementLength)
  {
    longestElementLength=ipAddress.length();
  }
  if (nodeMemoryAddress.length() > longestElementLength)
  {
    longestElementLength=nodeMemoryAddress.length();
  }

  uint32_t ourColumnWidth=longestElementLength+4+1; // 4 is for " |  "
  uint32_t numberOfColumns=floor(w.ws_col/ourColumnWidth);

  uint32_t nodeCounter = 0;
  while (nodeCounter < myNodes.GetN())
  {
    std::cout << endl;
    uint32_t columnCounter = 0;
    while (columnCounter < numberOfColumns && nodeCounter < myNodes.GetN())
    {
      uint32_t myOutputLength = GetLengthOfInteger(nodeCounter) + 5 + 4;
      std::cout << "Node " << nodeCounter;
      std::cout << SpacesAndTail(ourColumnWidth - myOutputLength);
      nodeCounter++;
      columnCounter++;
    }

    nodeCounter=nodeCounter-columnCounter;
    columnCounter = 0;
    std::cout << endl;
    while (columnCounter < numberOfColumns && nodeCounter < myNodes.GetN())
    {
      string memoryAddress;
      ostringstream convert;
      convert << myNodes.Get(nodeCounter);
      memoryAddress = convert.str();

      uint32_t myOutputLength = memoryAddress.length() + 4;
      std::cout << myNodes.Get(nodeCounter);
      std::cout << SpacesAndTail(ourColumnWidth - myOutputLength);
      nodeCounter++;
      columnCounter++;
    }

    nodeCounter=nodeCounter-columnCounter;
    columnCounter = 0;
    std::cout << endl;
    while (columnCounter < numberOfColumns && nodeCounter < myNodes.GetN())
    {
      string indexLength;          // string which will contain the result
      ostringstream convert;   // stream used for the conversion
      convert << aodvInterface.GetAddress(nodeCounter);      // insert the textual
representation of 'Number' in the characters in the stream
      indexLength = convert.str(); // set 'Result' to the contents of the stream

      uint32_t myOutputLength = indexLength.length() + 4;
      std::cout<< aodvInterface.GetAddress(nodeCounter);
      std::cout << SpacesAndTail(ourColumnWidth - myOutputLength);
      nodeCounter++;
      columnCounter++;
    }
  std::cout << std::endl;
  }
  std::cout << std::endl << std::endl;
  Print("PrintNodesContainerInfo",2);
}

void
CogMan::CreateVectorOfDistancesAllNodes()
{
  Print("CreateVectorOfDistancesAllNodes",1);
  for (int fromNode=0; fromNode<allNodes.GetN(); fromNode++) // represents the node of
interest
```

```
  {
    Vector sourcePosition = GetNodePosition(fromNode);
    for (int toNode=0; toNode<allNodes.GetN(); toNode++) // represents distance to other node
    {
      Vector destinationPosition = GetNodePosition(toNode);
      double distance = GetDistance(sourcePosition, destinationPosition);

      cout << "Time: " << Simulator::Now() << " Distance from " <<
        fromNode << " to " << toNode << " is " << distance ;

                                                                 //---------------------------
---------//
                                                                 //  A vector called
allNodesTheWorld  //
      allNodesTheWorld.push_back({ fromNode,                     //  to store many
nodeDistance        //
                                   toNode,                       //
//
                                   distance,                     //  The structure has
variables       //
                                   Simulator::Now().GetSeconds() //    sourceNode      int
//
                              });                                //    destinationNode int
//
                                                                 //    distance        double
//
    }                                                            //    myTime          double
//
  }                                                              //---------------------------
---------//
  Print("CreateVectorOfDistancesAllNodes",2);
}

void
CogMan::PrintVectorOfNodePositions(vector<currentPosition> theDistances, string theFilename)
{
  Print("PrintVectorOfNodePositions",1);
  cout << "Nodes in vector are: \n";                 //----------------------------------//
                                                     //  A vector called theDistances    //
                                                     //                                  //
  vector<currentPosition>::iterator vectorIndex;     //  The structure has variables     //
  vectorIndex = myPositions.begin();                 //    nodeId      int               //
                                                     //    time        double            //
  while (vectorIndex < myPositions.end())            //    x           double            //
  {                                                  //    y           double            //
    cout << vectorIndex->nodeId   << ",";            //----------------------------------//
    cout << vectorIndex->time     << ",";
    cout << vectorIndex->x        << ",";
    cout << vectorIndex->y        << "\n";
    vectorIndex++;
  }
  Print("PrintVectorOfNodePositions",2);
}

void
CogMan::PrintVectorOfMemoryEntries()
{
  Print("PrintVectorOfMemoryEntries",1);
  cout << "Entries in cognitive memory vector are: \n";

  vector<cognitivity>::iterator vectorIndex;            //----------------------------------//
  vectorIndex = managementNodeMemory.begin();           //  A vector of type <cognitivity>  //
  cout << "Time\tsNode\tdNode\tdR\t\trR\t\tsR\t\t";     //  called managementNodeMemory     //
  cout << "density\t\t\tchannel\tprotocol\trequest\t";  //                                  //
  cout << "progress\tbackoff\tperformance\n";           //  The structure has variables     //
  while (vectorIndex < managementNodeMemory.end())      //    double      atTime            //
  {                                                     //    int         sNode             //
    cout << vectorIndex->atTime          << "\t\t";     //    int         mNode             //
    cout << vectorIndex->sNode           << "\t\t";     //    double      dropRate          //
    cout << vectorIndex->mNode           << "\t\t";     //    double      receiveRate       //
    cout << vectorIndex->dropRate        << "\t\t";     //    double      sendRate          //
    cout << vectorIndex->receiveRate     << "\t\t";     //    double      density           //
    cout << vectorIndex->sendRate        << "\t\t";     //    int         channel           //
    cout << vectorIndex->density         << "\t\t";     //    string      routingProtocol   //
    cout << vectorIndex->channel         << "\t\t";     //    char        request           //
    cout << vectorIndex->routingProtocol << "\t\t";     //    char        requestProgress   //
    cout << vectorIndex->request         << "\t\t";     //    double      backOffTime       //
    cout << vectorIndex->requestProgress << "\t\t";     //    int         performance       //
    cout << vectorIndex->backOffTime     << "\t\t";     //----------------------------------//
    cout << vectorIndex->performance     << "\n";
    vectorIndex++;
  }
  Print("PrintVectorOfMemoryEntries",2);
}

void
CogMan::CreateVectorFlowMonitorInformation(Ptr<FlowMonitor> monitor)
{
  Print("CreateVectorFlowMonitorInformation",1);
  Ptr<Ipv4FlowClassifier> classifier = DynamicCast<Ipv4FlowClassifier> (flowmon.GetClassifier
());
  std::map<FlowId, FlowMonitor::FlowStats> stats = monitor->GetFlowStats ();
  for (std::map<FlowId, FlowMonitor::FlowStats>::const_iterator i = stats.begin (); i !=
stats.end (); ++i)
  {
    Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow (i->first);       //----------------
--------------------//
                                                                             //  A vector
called myDataFlows       //
    myDataFlows.push_back({ i->first,                                        //  to store many
flowInfomation      //
                            t.sourceAddress,                                 //
//
                            t.destinationAddress,                            //  The structure
has variables       //
                            i->second.txPackets,                             //    flowId
unsigned int    //
                            i->second.txBytes,                               //
sourceAddress   Ipv4Address     //
                            i->second.rxBytes * 8.0 / 9.0 / 1000 / 1000,     //
destinationNode Ipv4Address     //
                            i->second.rxPackets,                             //    txPackets
uint32_t        //
                            i->second.rxBytes,                               //    txBytes
uint64_t        //
                            i->second.rxBytes * 8.0 / 9.0 / 1000 / 1000      //    rxOffered
double          //
                            });                                              //    rxPackets
uint64_t        //
                                                                             //    rxBytes
uint64_t        //
                                                                             //    throughput
double          //
                                                                             //----------------
--------------------//
  }
  Print("CreateVectorFlowMonitorInformation",2);
}



void
CogMan::PrintFlowMonitorInformation(Ptr<FlowMonitor> monitor)
{
  Print("PrintFlowMonitorInformation",1);
  Ptr<Ipv4FlowClassifier> classifier = DynamicCast<Ipv4FlowClassifier> (flowmon.GetClassifier
());
  std::map<FlowId, FlowMonitor::FlowStats> stats = monitor->GetFlowStats ();
  for (std::map<FlowId, FlowMonitor::FlowStats>::const_iterator i = stats.begin (); i !=
stats.end (); ++i)
  {
    Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow (i->first);
    std::cout << "Flow " << i->first - 2 << " (" << t.sourceAddress << " -> " <<
t.destinationAddress << ")\n";
    std::cout << "  Tx Packets: " << i->second.txPackets << "\n";
    std::cout << "  Tx Bytes:   " << i->second.txBytes << "\n";
    std::cout << "  TxOffered:  " << i->second.txBytes * 8.0 / 9.0 / 1000 / 1000  << "
Mbps\n";
    std::cout << "  Rx Packets: " << i->second.rxPackets << "\n";
    std::cout << "  Rx Bytes:   " << i->second.rxBytes << "\n";
    std::cout << "  Throughput: " << i->second.rxBytes * 8.0 / 9.0 / 1000 / 1000  << "
Mbps\n";
  }
  Print("PrintFlowMonitorInformation",2);
```

```
}



// PLACEMENT STRATEGIES









// BATTERIES AND POWER MANAGEMENT
void
RemainingEnergy (double oldValue, double remainingEnergy)
{
  //Print("RemainingEnergy",1);
  //cout << Simulator::Now ().GetSeconds () << "s Current remaining energy = "
  //  << remainingEnergy << "J" << endl;
  //Print("RemainingEnergy",2);
}

void
TotalEnergy (double oldValue, double totalEnergy)
{
  //Print("TotalEnergy",1);
  //cout << Simulator::Now ().GetSeconds () << "s Total energy consumed by radio = " <<
  //totalEnergy << "J" << endl;
  //Print("TotalEnergy",2);
}

void
CogMan::PrintRemainingEnergy(int nodeId)
{
  Print("PrintRemainingEnergy",1);
  // get the nth element from sources container (warning, this is not for a particular nodeId
  // but for a particular sourceId).  Use a loop later to get the correct source should there
  // be more than one node container.
  Ptr<EnergySource> myEnergySource = DynamicCast<EnergySource>(sources.Get(nodeId));

  // this gets the node pointer for the nth battery that is in the container above.
  Ptr<Node> myNode = myEnergySource->GetNode();

  message << "node " << nodeId << " energy remaining is " << myEnergySource-
>GetRemainingEnergy() << "J at " << Simulator::Now().GetSeconds() << endl;
  Print(message.str(), 14);

  /*FindDeviceEnergyModels ("ns3::WifiRadioEnergyModel").Get (0)
  cout << "Node Ptr\tEnergy Ptr\t\n";
  cout << myNode << "\t" << myEnergySource << endl;
  cout << "Remaining Energy for node " << nodeId << " is " <<
  myEnergySource->GetRemainingEnergy() << " J\n";   */
  Print("PrintRemainingEnergy",2);
}


double
CogMan::GetRemainingEnergy(int nodeId)
{
  Ptr<EnergySource> source = allNodes.Get(nodeId)->GetObject<EnergySourceContainer>()->Get(0);
 // cout << "EnergySource is " << source;

  double x;
  x = source->GetRemainingEnergy();
  //cout << "Remaining Energy is: " << x <<endl;
  return x;
}



void
CogMan::SetAodvRemainingEnergy(int nodeId)
{
  //Ptr<Node> node = aodvNodes.Get(nodeId);
  Ptr<EnergySource> source = aodvNodes.Get(nodeId)->GetObject<EnergySourceContainer>()-
>Get(0);
  //cout << "EnergySource is " << source;

  double x;
  x = source->GetRemainingEnergy();
  //cout << "Remaining Energy is: " << x <<endl;
}

void
CogMan::SetDsdvRemainingEnergy(int nodeId)
{

}

void
CogMan::SetRemainingEnergy(int nodeId)
{
  Print ("SetRemainingEnergy",1);
  if (currentProtocol == "aodv")
  {
    SetAodvRemainingEnergy(nodeId);
  }
  else
  {
    SetDsdvRemainingEnergy(nodeId);
  }
  Print ("SetRemainingEnergy",2);
}

double
CogMan::GetRemainingEnergyES(int nodeId)
{
  Print("GetRemainingEnergy",1);
  /* Get a pointer to the first element of the sources container.
     Get a pointer to the last + 1 element of the sources container.
     While not checked every source in the sources container.
     Create a pointer to an energy source.
     Set this equal to the address that the sourceBegin pointer is pointing to.
     Get the node pointer of the node that is using the energy source currently pointed to.
     Get the node pointer of the node that was passed to the method (nodeId).
     If this node pointer is the same as the node pointer for this energy source we have found
       the correct node using this energy source (battery).
       Return the remaining energy of this source.
     Else move sourceBegin pointer to the next source in the container and do again.
  */


  //message << "The nodeId is " << nodeId;
  //Print(message, 14)
  //message << "The node address is " << allNodes.Get(nodeId);
  //Print(message,14);

  EnergySourceContainer::Iterator sourceBegin = sources.Begin();
  EnergySourceContainer::Iterator sourceEnd   = sources.End();

  while (sourceBegin < sourceEnd)
  {
    Ptr<EnergySource> myEnergySource = *(sourceBegin);
    //cout << "\npointer to the energy source is: " << myEnergySource << endl;

    Ptr<Node> myNode = myEnergySource->GetNode();
    //cout << "\npointer to node address is: " << myNode;

    if (myNode == aodvNodes.Get(nodeId))
    {
      //cout << "\nfound a match";
      //cout << "At time: " << Simulator::Now().GetSeconds()
      //   << "s the remaining energy at node " << nodeId << " is "
      //   << myEnergySource->GetRemainingEnergy() << "J";
      Print(message.str(),14);
      return myEnergySource->GetRemainingEnergy();
    }
    sourceBegin++;
  }
  Print("GetRemainingEnergy",2);
}
```

```
void
CogMan::ChangeEnergy()
{

  //   BasicEnergySourceHelper     aodvBasicSource;
  //   WifiRadioEnergyModelHelper  aodvRadioEnergy;
  //   DeviceEnergyModelContainer  aodvDeviceEnergyModel;
  //   EnergySourceContainer       aodvEnergySources;

  //cout << "ChangeEnergy\n";
  Ptr<Node> node = aodvNodes.Get(1);
  //cout << "Node Pointer: " << node << "\n";

  Ptr<EnergySource>       source;
  //cout << "EnergySource Pointer: " << source << "\n";

  Ptr<BasicEnergySource>  bes;
  //cout << "BasicEnergySource Pointer: " << bes << "\n";

  //cout << node->GetObject<EnergySource>();

  double x;

  source = node->GetObject<EnergySource>();

  //if (!source)
  //  NS_FATAL_ERROR ("Error! no find ns3::EnergySource.");
  //  bes = node->GetObject<BasicEnergySource> ();

  bes = DynamicCast<BasicEnergySource> (aodvEnergySources.Get (1));

  //Ptr<EnergySource> source = m_node->GetObject<EnergySourceContainer> ()->Get (0);

  if (!bes)
    NS_FATAL_ERROR ("Error! no find ns3::BasicEnergySource.");

  x = bes->GetRemainingEnergy();
  //cout << "Remaining Energy is: " << x <<endl;

  bes->SetInitialEnergy(2 * x);
  x = bes->GetRemainingEnergy();
  //cout << "Remaining Energy is: " << x <<endl;
}



void
CogMan::SetBatteries()
{
  Print("SetBatteries",1);
  // Delcared Globally.

  // BasicEnergySourceHelper basicSourceHelper;
  // WifiRadioEnergyModelHelper radioEnergyHelper;
  // DeviceEnergyModelContainer deviceModels;
  // EnergySourceContainer sources;

  //   BasicEnergySourceHelper     aodvBasicSource;
  //   WifiRadioEnergyModelHelper  aodvRadioEnergy;
  //   DeviceEnergyModelContainer  aodvDeviceEnergyModel;
  //   EnergySourceContainer       aodvEnergySources;

  message << "Installing batteries.";
  Print (message.str(), 14);
  // configure energy source
  basicSourceHelper.Set ("BasicEnergySourceInitialEnergyJ", DoubleValue (initialEnergy));  //
average battery
  aodvEnergySources = basicSourceHelper.Install (aodvNodes);

  // configure radio energy model
  radioEnergyHelper.Set ("TxCurrentA", DoubleValue (0.0174));
  // install device model
  aodvDeviceEnergyModel = radioEnergyHelper.Install (aodvDevices, aodvEnergySources);

  /** connect trace sources **/
  /***********************************************************************/
  // energy source
  //for (uint32_t i = 0; i<nodes.GetN()-1; i++)
  //{

  Ptr<BasicEnergySource> basicSourcePtr = DynamicCast<BasicEnergySource>
(aodvEnergySources.Get (1));
  basicSourcePtr->TraceConnectWithoutContext ("RemainingEnergy", MakeCallback
(&RemainingEnergy));

  // device energy model
  Ptr<DeviceEnergyModel> basicRadioModelPtr =
    basicSourcePtr->FindDeviceEnergyModels ("ns3::WifiRadioEnergyModel").Get (0);
  //NS_ASSERT (basicRadioModelPtr != NULL);
  basicRadioModelPtr->TraceConnectWithoutContext ("TotalEnergyConsumption", MakeCallback
(&TotalEnergy));
  /***********************************************************************/
  //}
  Print("SetBatteries",2);
}

// PACKETS








Ptr<Socket>
CogMan::SetupPacketReceive (Ipv4Address addr, Ptr<Node> node)
{
  Print("SetupPacketReceive",1);
  TypeId tid = TypeId::LookupByName ("ns3::UdpSocketFactory");
  Ptr<Socket> sink = Socket::CreateSocket (node, tid);
  InetSocketAddress local = InetSocketAddress (addr, port);
  sink->Bind (local);
  sink->SetRecvCallback (MakeCallback (&CogMan::ReceivedPacket, this));
  return sink;
  Print("SetupPacketReceive",2);
}

void
CogMan::CreateTrafficFlowForCongestion(int sourceNodeId, int destinationNodeId, double
startAt, double stopAt)
{
  Print("CreateTrafficFlowForCongestion",1);
  // Destination
  // First we have to determine if this is aodv or dsdv.
  // cout << "Protocol is " << currentProtocol;
  //CreateAodvTrafficFlow(sourceNodeId, destinationNodeId, startAt, stopAt);
  //sourceNodeId = numOfNodes + sourceNodeId;
  //destinationNodeId = numOfNodes + destinationNodeId;
  //CreateDsdvTrafficFlow(sourceNodeId, destinationNodeId, startAt, stopAt);
  CreateCongestionTrafficFlow(sourceNodeId, destinationNodeId, startAt, stopAt);

  Print("CreateTrafficFlowForCongestion",2);
}

void
CogMan::CreateCongestionTrafficFlow(int sourceNodeId, int destinationNodeId, double startAt,
double stopAt)
{
  Print("CreateCongestionTrafficFlow",1);
  // Destination
  // cout << "**********";

  Address remoteAddress(InetSocketAddress (congestionInterface.GetAddress(destinationNodeId),
port));
  PacketSinkHelper receiver ("ns3::UdpSocketFactory", remoteAddress);
  ApplicationContainer receiverapp =receiver.Install(congestionNodes.Get(destinationNodeId));
  // INSTALLING SINK-DESTINATION:
  Ptr<Socket> sink = SetupPacketReceive (congestionInterface.GetAddress
    (destinationNodeId), congestionNodes.Get (destinationNodeId));
  receiverapp.Start (Time(startAt));
  receiverapp.Stop (Time(stopAt));
  Print("Destination Created", 10);

  // lets create a server to receive packets
  //  UdpEchoServerHelper echoServer(9);
  //  ApplicationContainer serverApps = echoServer.Install(nodes.Get(dNode)); // last node
  //  serverApps.Start (Seconds(10.0));
  //  serverApps.Stop(Seconds(20.0));

  // lets create the client to send packets
  //  UdpEchoClientHelper echoClient (interfaces.GetAddress(sNode), 9);
  //  echoClient.SetAttribute ("MaxPackets", UintegerValue(10000));
  //  echoClient.SetAttribute ("Interval", TimeValue(Seconds(1.0)));
  //  echoClient.SetAttribute ("PacketSize", UintegerValue(1024));

  // Source
  OnOffHelper sender ("ns3::UdpSocketFactory", remoteAddress);
  sender.SetConstantRate (DataRate("128kb/s"));
  sender.SetAttribute ("OnTime", StringValue ("ns3::ConstantRandomVariable[Constant=1.0]"));
  sender.SetAttribute ("OffTime", StringValue ("ns3::ConstantRandomVariable[Constant=0.0]"));
  ApplicationContainer senderapp;

  senderapp=sender.Install(congestionNodes.Get(sourceNodeId));

  senderapp.Start (Seconds (startAt));
  senderapp.Stop (Seconds (stopAt));

  Print("Source Created", 10);

  message << "At time: " << Simulator::Now().GetSeconds() << "s ";
  message << "setting traffic flow for node " << sourceNodeId << " (" <<
congestionInterface.GetAddress(sourceNodeId) << ")";
  message << " to send data to " << destinationNodeId << " (" <<
congestionInterface.GetAddress(destinationNodeId) << ")";
  message << " starting at " << startAt << " until " << stopAt;

  Print(message.str(), 10);

  //Ptr<UniformRandomVariable> var = CreateObject<UniformRandomVariable> ();
  //int i=floor(6*rand())
  //ApplicationContainer temp = onoff1.Install (nodes.Get(floor(rand()%(nNodes-dNode-
1))+dNode));

  //Config::ConnectWithoutContext("/NodeList/1/ApplicationList/0/$ns3::OnOffApplication/Rx",
MakeCallback (&SetTagTid));
  //  Ptr<OnOffApplication> onoffappBE;
  //  onoffappBE = DynamicCast<OnOffApplication>(senderapps1.Get(0));
  //  onoffappBE->TraceConnectWithoutContext("Tx",  MakeBoundCallback (&SetTagTid, AC_BE));
  Print("CreateCongestionTrafficFlow",2);
}




std::string
CogMan::PrintReceivedPacket (Ptr<Socket> socket, Ptr<Packet> packet)
{
  Print("PrintReceivedPacket",1);

  SocketAddressTag tag;
  bool found;
  found = packet->PeekPacketTag (tag);

  Ptr<Node> myNode = allNodes.Get(socket->GetNode ()->GetId ());
  Ptr<Ipv4> ipv4 = myNode->GetObject<Ipv4>();

  Ipv4InterfaceAddress iaddr = ipv4->GetAddress (1,0);

  // std::cout << "\niaddr " << iaddr;
  // socket->GetNode()->GetId ();

  Ipv4Address addri = iaddr.GetLocal ();
  if (found)
    {
      InetSocketAddress addr = InetSocketAddress::ConvertFrom (tag.GetAddress ());

      message << " Packet Received at " << Simulator::Now().GetSeconds ()
              << " s\n   source\tIP Address is: " << addr.GetIpv4();
      Print(message.str(),6);

      message << "   destination IP Address is: " << addri
              << "\n   distance is " << GetDistance(GetNodePosition(1),GetNodePosition(0));
      Print(message.str(),6);
      RecordReceivedPacket(addr.GetIpv4(),addri);
      if (!reportedReceived)
      {
        reportedReceived = true;
        cout << GetTimeNow('s') << " First data packet received\n";
      }
    }
  else
    {
      message << " NOT received one packet!";
      Print(message.str(),6);
    }

  return message.str ();
  Print("PrintReceivedPacket",2);   inRangeUntil
}

void
CogMan::RecordReceivedPacket(Ipv4Address src, Ipv4Address dst)
{
  string protocol = GetProtocol(dst);
  if (protocol == "aodv")
  {
    totalAodvPacketsReceived++;
    // cout << " Current total AODV received is " << totalAodvPacketsReceived<<endl;
  }
  else if (protocol == "dsdv")
  {
    totalDsdvPacketsReceived++;
    // cout << " Current total DSDV received is " << totalDsdvPacketsReceived<<endl;
  }
                                            //-----------------------------------//
                                            //  A vector called packetReceived   //
  packetReceived.push_back({src,            //  to store many IpAddresses        //
                            dst,            //                                   //
                            GetTimeNow('s') //    source      Ipv4Address        //
                          });               //    destination Ipv4Address        //
                                            //    myTime      double             //
                                            //-----------------------------------//
  allPackets.push_back({src,dst,GetTimeNow('s'),'R'});
  // cout << " PACKET RECEIVED !!!!";
  // cout << " cp is : " << currentProtocol << " and protocol is : " << protocol;
  if (protocol == currentProtocol)
  {
      //cout << " PACKET RECEIVED IN HERE TOOOOOOOOOO  !!!!";
    totalSwitchingReceived++;
    // cout << "Current total switching received is " << totalSwitchingReceived<<endl;
    packetReceivedByCurrentProtocol.push_back({src,
                                               dst,
                                               GetTimeNow('s')
                                              });
  }
}



void
CogMan::RecordSentPacket(Ipv4Address src, Ipv4Address dst)
{
  double timeNow = GetTimeNow('s');
  string protocolThatSent = GetProtocol(dst);


  if (protocolThatSent != "routeDiscovery")     //-----------------------------------//
  {                                             //  A vector called packetReceived   //
  packetSent.push_back({ src,                   //  to store many IpAddresses        //
                         dst,                   //                                   //
                         timeNow                //    source      Ipv4Address        //
                      });                       //    destination Ipv4Address        //
  allPackets.push_back({src,dst,timeNow,'S'});  //    myTime      double             //
  }                                             //-----------------------------------//

  if (protocolThatSent == currentProtocol)
  {
    packetSentByCurrentProtocol.push_back({src,
                                           dst,
                                           timeNow
                                         });
  }
}

void
CogMan::RecordDroppedPacket(Ipv4Address src, Ipv4Address dst)
{
  double timeNow = GetTimeNow('s');
  string protocolThatDropped = GetProtocol(dst);

  if (protocolThatDropped != "routeDiscovery")  //-----------------------------------//
  {                                             //  A vector called packetDropped    //
  packetDropped.push_back({ src,                //  to store many IpAddresses        //
                            dst,                //                                   //
                            timeNow             //    source      Ipv4Address        //
                          });                   //    destination Ipv4Address        //
  allPackets.push_back({src,dst,timeNow,'D'});  //    myTime      double             //
  }                                             //-----------------------------------//

  if (protocolThatDropped == currentProtocol)
  {
    packetDroppedByCurrentProtocol.push_back({src,
                                           dst,
                                           timeNow
                                         });
  }
}

string
CogMan::ConvertIPAddressToString(Ipv4Address addr)
{
  string ipAddress;            // string to store IP Address
  ostringstream convert;       // stream used for the conversion
  convert << addr;             // copy into ostringstream object
  ipAddress = convert.str();   // convert to a string
  return ipAddress;
}

string
CogMan::ConvertDoubleToString(double doubleObject)
{
  string stringObject;           // string to converted double value
  ostringstream convert;         // stream used for the conversion
  convert << doubleObject;       // copy double value into ostringstream object
  stringObject = convert.str(); // convert to a string
  return stringObject;
}

string
CogMan::GetProtocol(Ipv4Address addr)
{
  string protocol;
  string ipAddress = ConvertIPAddressToString(addr);

  if (ipAddress[3] == '2')
  {
    protocol = "aodv";
  }
  if (ipAddress[3] == '1')
  {
    protocol = "dsdv";
  }
  if (ipAddress.find("255")!=std::string::npos)
  {
    protocol = "routeDiscovery"; // route discovery packet
  }
  return protocol;
}



void
CogMan::ReceivedPacket(Ptr<Socket> skt)
{
  Print("ReceivedPacket",1);
  Ptr<Packet> packet;
  while ((packet = skt->Recv ()))
    {
      bytesTotal += packet->GetSize ();
      packetsReceived += 1;
      PrintReceivedPacket (skt, packet);
    }
  //std::cout << " one packet received and socket is at " <<  skt << std::endl;
  Print("ReceivedPacket",2);
}


// ELECTION

/*
step3: checking their battery lifes, here come your weightings
5*distance+3*predicted time+2*battery life
and i guess in the future the coefficients should be adjusted (as part of cognitivity)
according to previous decisions
*/





void
CogMan::RecordSystemState(int sNode, int mNode)
{
  int switchingReceived;
  if (currentProtocol == "aodv" && aodvBool) // if now and before it was
  {
     totalswitchingA = totalAodvPacketsReceived;
     switchingReceived = totalAodvPacketsReceived - totalPreviousSwitchingA;
     totalPreviousSwitchingA = totalAodvPacketsReceived;
     totalPreviousSwitchingD = totalDsdvPacketsReceived;
     aodvBool = true;
     dsdvBool = false;
  }

  if (currentProtocol == "aodv" && dsdvBool) // if now and before it was
  {
     totalswitchingA = totalAodvPacketsReceived;
     switchingReceived = totalAodvPacketsReceived - totalPreviousSwitchingA;
     totalPreviousSwitchingA = totalAodvPacketsReceived;
     totalPreviousSwitchingD = totalDsdvPacketsReceived;
     aodvBool = true;
     dsdvBool = false;
  }

  if (currentProtocol == "dsdv" && dsdvBool)
  {
     totalswitchingD = totalDsdvPacketsReceived;
     switchingReceived = totalDsdvPacketsReceived - totalPreviousSwitchingD ;
     totalPreviousSwitchingD = totalDsdvPacketsReceived;
     totalPreviousSwitchingA = totalAodvPacketsReceived;
     dsdvBool = true;
     aodvBool = false;
  }
  if (currentProtocol == "dsdv" && aodvBool)
  {
     totalswitchingD = totalDsdvPacketsReceived;
     switchingReceived = totalDsdvPacketsReceived - totalPreviousSwitchingD ;
     totalPreviousSwitchingD = totalDsdvPacketsReceived;
     totalPreviousSwitchingA = totalAodvPacketsReceived;
     dsdvBool = true;
     aodvBool = false;
  }
  double density = GetDensityFromNode(mNode);
  int channel    = GetChannel(mNode);

  systemState.push_back(
    {  sNode,
       mNode,
       GetTimeNow('s'),
       density,
       currentProtocol,
       totalAodvPacketsDropped,
       totalDsdvPacketsDropped,
       totalAodvPacketsReceived,
       totalDsdvPacketsReceived,
       channel,
       switchingReceived
    });
}

double
CogMan::GetChannel(int mNode)
{
  // cout << "I am in getchannel and the current protocol is " << currentProtocol <<endl;
  PointerValue tmpPhy;
  if (currentProtocol == "aodv")
  {
    aodvDevices.Get(mNode)->GetAttribute("Phy", tmpPhy);
  }
  else
  {
    dsdvDevices.Get(mNode)->GetAttribute("Phy", tmpPhy);
  }

  Ptr<Object> wifiPhyLayer = tmpPhy.GetObject();
  Ptr<YansWifiPhy> yansWifiPhyLayer = wifiPhyLayer->GetObject<YansWifiPhy>();
  double frequency = yansWifiPhyLayer->GetChannelFrequencyMhz();
  message << "\nThe frequency is: " << frequency << "Hz";
  message << "\nThe channel number is: " << yansWifiPhyLayer->GetChannelNumber();
  return yansWifiPhyLayer->GetChannelNumber();
  Print(message.str(),8);
}

void
CogMan::PrintAllChannels()
{
  // cout << "I am in getchannel and the current protocol is " << currentProtocol <<endl;
  PointerValue tmpPhy;

  for (int i=0; i<numOfNodes; i++)
  {
    cout << "\n Number of devices is: " << aodvDevices.GetN();

    aodvDevices.Get(i)->GetAttribute("Phy", tmpPhy);
    Ptr<Object> wifiPhyLayer = tmpPhy.GetObject();
    Ptr<YansWifiPhy> yansWifiPhyLayer = wifiPhyLayer->GetObject<YansWifiPhy>();
    double frequency = yansWifiPhyLayer->GetChannelFrequencyMhz();
    cout << "\nAODV NODE: " << i;
    cout << "  frequency is: " << frequency << "Hz";
    cout << "  The channel number is: " << yansWifiPhyLayer->GetChannelNumber();
    // cout << message.str();
  }
  for (int i=0; i<numOfNodes; i++)
  {
    dsdvDevices.Get(i)->GetAttribute("Phy", tmpPhy);
    Ptr<Object> wifiPhyLayer = tmpPhy.GetObject();
    Ptr<YansWifiPhy> yansWifiPhyLayer = wifiPhyLayer->GetObject<YansWifiPhy>();
    double frequency = yansWifiPhyLayer->GetChannelFrequencyMhz();
    cout << "\nDSDV NODE: " << i;
    cout << "  frequency is: " << frequency << "Hz";
    cout << "  The channel number is: " << yansWifiPhyLayer->GetChannelNumber();
    // cout << message.str();
  }
  for (int i=0; i<numOfControlledNodes; i++)
  {
    aodvControlledDevices.Get(i)->GetAttribute("Phy", tmpPhy);
    Ptr<Object> wifiPhyLayer = tmpPhy.GetObject();
    Ptr<YansWifiPhy> yansWifiPhyLayer = wifiPhyLayer->GetObject<YansWifiPhy>();
    double frequency = yansWifiPhyLayer->GetChannelFrequencyMhz();
    cout << "\nAODV CONTROLLED NODE: " << i;
    cout << "  frequency is: " << frequency << "Hz";
    cout << "  The channel number is: " << yansWifiPhyLayer->GetChannelNumber();
    // cout << message.str();
  }
  for (int i=0; i<numOfControlledNodes; i++)
  {
    dsdvControlledDevices.Get(i)->GetAttribute("Phy", tmpPhy);
    Ptr<Object> wifiPhyLayer = tmpPhy.GetObject();
    Ptr<YansWifiPhy> yansWifiPhyLayer = wifiPhyLayer->GetObject<YansWifiPhy>();
    double frequency = yansWifiPhyLayer->GetChannelFrequencyMhz();
    cout << "\nDSDV CONTROLLED NODE: " << i;
    cout << "  frequency is: " << frequency << "Hz";
    cout << "  The channel number is: " << yansWifiPhyLayer->GetChannelNumber();
    // cout << message.str();
  }
}


void
CogMan::SetChannel(int mNode, int channelNumber)
{

  // cout << "I am in  SET channel and the current protocol is " << currentProtocol <<endl;
  PointerValue tmpPhy;
  if (currentProtocol == "aodv")
  {
    aodvDevices.Get(mNode)->GetAttribute("Phy", tmpPhy);
  }
  else
  {
    dsdvDevices.Get(mNode)->GetAttribute("Phy", tmpPhy);
  }

  Ptr<Object> wifiPhyLayer = tmpPhy.GetObject();
  Ptr<YansWifiPhy> yansWifiPhyLayer = wifiPhyLayer->GetObject<YansWifiPhy>();
  double frequency = yansWifiPhyLayer->GetChannelFrequencyMhz();
  message << "\nThe frequency is: " << frequency << "Hz";
  message << "\nThe channel number is: " << yansWifiPhyLayer->GetChannelNumber();

  //wifiPhy.Set("ChannelNumber", UintegerValue(11));
  //yansWifiPhyLayer->SetChannel(&channelNumber);


  YansWifiChannelHelper wifiChannel;
  wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");
  wifiChannel.AddPropagationLoss ("ns3::FriisPropagationLossModel"); // we can create channel
later

  //wifiPhy.Set("ChannelNumber", UintegerValue(3));
  yansWifiPhyLayer->SetChannelNumber (channelNumber);





  wifiPhyLayer = tmpPhy.GetObject();
  yansWifiPhyLayer = wifiPhyLayer->GetObject<YansWifiPhy>();
  frequency = yansWifiPhyLayer->GetChannelFrequencyMhz();
  message << "\nThe frequency is: " << frequency << "Hz";
  message << "\nThe channel number is: " << yansWifiPhyLayer->GetChannelNumber();



  //return yansWifiPhyLayer->GetChannelNumber();
  Print(message.str(),8);



  //void ns3::YansWifiPhy::SetChannelNumber ( uint16_t  id  )
  //yansWifiPhyLayer->SetChannel(&wifiChannel);
  //yansWifiPhyLayer->SetFrequency(5);
  //cout << "\nthe frequency is: " << frequency << "Hz";
  //cout << "\n The channel number is: " << yansWifiPhyLayer->GetChannelNumber();

  // get center frequency
```

```
 // cout << "\nThe center frequency is: " << yansWifiPhyLayer->GetCenterFrequencyMhz();



  //uint32_t ns3::YansWifiPhy::GetFrequency ( void    ) const
  //Ptr<YansWifiPhy> myWifi = GetObject<YansWifiPhy>();
  //cout << wifiPhy.GetChannelFrequencyMhz();

  //double ns3::FriisPropagationLossModel::GetFrequency ( void    ) const
  //cout << "frequency: " << wifiChannel.GetFrequency();



  /*  Ptr<WifiPhy> ns3::YansWifiPhyHelper::Create ( Ptr<Node> node,
                                                Ptr<WifiNetDevice> device ) const

 Returns a pointer to WifiPhy
 Create method (pointer to node, pointer to network device)
 void ns3::YansWifiPhyHelper::EnableAsciiInternal ( Ptr< OutputStreamWrapper >  stream,
                                                    std::string   prefix,
                                                    Ptr< NetDevice >  nd,
                                                    bool  explicitFilename )
 void ns3::YansWifiPhyHelper::EnablePcapInternal (  std::string   prefix,
                                                    Ptr< NetDevice >  nd,
                                                    bool  promiscuous,
                                                    bool  explicitFilename ) */
}


void
CogMan::SetChannelControlledNodes (int mNode, int channelNumber)
{

  // cout << "I am in  SET channel and the current protocol is " << currentProtocol <<endl;
  PointerValue tmpPhy;
  if (currentProtocol == "aodv")
  {
    aodvControlledDevices.Get(mNode)->GetAttribute("Phy", tmpPhy);
  }
  else
  {
    dsdvControlledDevices.Get(mNode)->GetAttribute("Phy", tmpPhy);
  }

  Ptr<Object> wifiPhyLayer = tmpPhy.GetObject();
  Ptr<YansWifiPhy> yansWifiPhyLayer = wifiPhyLayer->GetObject<YansWifiPhy>();
  double frequency = yansWifiPhyLayer->GetChannelFrequencyMhz();
  message << "\nThe frequency is: " << frequency << "Hz";
  message << "\nThe channel number is: " << yansWifiPhyLayer->GetChannelNumber();

  //wifiPhy.Set("ChannelNumber", UintegerValue(11));
  //yansWifiPhyLayer->SetChannel(&channelNumber);


  YansWifiChannelHelper wifiChannel;
  wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");
  wifiChannel.AddPropagationLoss ("ns3::FriisPropagationLossModel"); // we can create channel
later

  //wifiPhy.Set("ChannelNumber", UintegerValue(3));
  yansWifiPhyLayer->SetChannelNumber (channelNumber);





  wifiPhyLayer = tmpPhy.GetObject();
  yansWifiPhyLayer = wifiPhyLayer->GetObject<YansWifiPhy>();
  frequency = yansWifiPhyLayer->GetChannelFrequencyMhz();
  message << "\nThe frequency is: " << frequency << "Hz";
  message << "\nThe channel number is: " << yansWifiPhyLayer->GetChannelNumber();



  //return yansWifiPhyLayer->GetChannelNumber();
  Print(message.str(),8);
  //void ns3::YansWifiPhy::SetChannelNumber ( uint16_t  id  )
  //yansWifiPhyLayer->SetChannel(&wifiChannel);
  //yansWifiPhyLayer->SetFrequency(5);
  //cout << "\nthe frequency is: " << frequency << "Hz";
  //cout << "\n The channel number is: " << yansWifiPhyLayer->GetChannelNumber();

  // get center frequency

 // cout << "\nThe center frequency is: " << yansWifiPhyLayer->GetCenterFrequencyMhz();




  //uint32_t ns3::YansWifiPhy::GetFrequency ( void    ) const
  //Ptr<YansWifiPhy> myWifi = GetObject<YansWifiPhy>();
  //cout << wifiPhy.GetChannelFrequencyMhz();

  //double ns3::FriisPropagationLossModel::GetFrequency ( void    ) const
  //cout << "frequency: " << wifiChannel.GetFrequency();



  /*  Ptr<WifiPhy> ns3::YansWifiPhyHelper::Create ( Ptr<Node> node,
                                              Ptr<WifiNetDevice> device ) const

 Returns a pointer to WifiPhy
 Create method (pointer to node, pointer to network device)
 void ns3::YansWifiPhyHelper::EnableAsciiInternal ( Ptr< OutputStreamWrapper >  stream,
                                                  std::string   prefix,
                                                  Ptr< NetDevice >  nd,
                                                  bool  explicitFilename )
 void ns3::YansWifiPhyHelper::EnablePcapInternal (  std::string   prefix,
                                                  Ptr< NetDevice >  nd,
                                                  bool  promiscuous,
                                                  bool  explicitFilename ) */
}




bool
CogMan::CriticalDensity(double currentDensity, double refDef)
{
  if (currentProtocol == "aodv")
  {
    if (currentDensity > refDef * 1.3)
    {
      return true;
    }
    else
    {
      return false;
    }
  }
  else
  {
    if (currentDensity < refDef * 0.8)
    {
      return true;
    }
    else
    {
      return false;
    }
  }
}

double
CogMan::GetCurrentDropRate()
{
  vector<memorySystemState>::iterator vectorIndex;
  vectorIndex = systemState.end()-1;

  int currentDroppedPackets = 0;
  int previousDroppedPackets = 0;
  double currentTime = 0;
  double previousTime = 0;

  int correctLines = 0;
  while (correctLines < 2 && vectorIndex >= systemState.begin())
  {
    if (vectorIndex->protocol == currentProtocol)
    {
      correctLines++;
      if (correctLines == 1 && currentProtocol == "aodv")
      {
        currentDroppedPackets = vectorIndex->droppedPacketsAodv;
        currentTime = vectorIndex->time;
      }
      else if (correctLines == 1 && currentProtocol == "dsdv")
      {
        currentDroppedPackets = vectorIndex->droppedPacketsDsdv;
        currentTime = vectorIndex->time;
      }
      else if (correctLines == 2 && currentProtocol == "aodv")
      {
        previousDroppedPackets = vectorIndex->droppedPacketsAodv;
        previousTime = vectorIndex->time;
      }
      else if (correctLines == 2 && currentProtocol == "dsdv")
      {
        previousDroppedPackets = vectorIndex->droppedPacketsDsdv;
        previousTime = vectorIndex->time;
      }
      else
      {
        // cout << "Something went terribly wrong working out which protocol is in use!";
      }
    }
    vectorIndex--;
  }

  if (correctLines < 2)
  {
    return -1;
  }
  else
  {
    // cout << "hahaha"<< (currentDroppedPackets - previousDroppedPackets) / (currentTime -
previousTime) <<endl;
    return (currentDroppedPackets - previousDroppedPackets) / (currentTime - previousTime);
  }
}

double
CogMan::GetCurrentReceiveRate()
{
  vector<memorySystemState>::iterator vectorIndex;
  vectorIndex = systemState.end()-1;

  int currentReceivedPackets = 0;
  int previousReceivedPackets = 0;
  double currentTime = 0;
  double previousTime = 0;

  int correctLines = 0;
  while (correctLines < 2 && vectorIndex >= systemState.begin())
  {
    if (vectorIndex->protocol == currentProtocol)
    {
      correctLines++;
      if (correctLines == 1 && currentProtocol == "aodv") //need to check sNode as well
      {
        currentReceivedPackets = vectorIndex->receivedPacketsAodv;
        currentTime = vectorIndex->time;
      }
      else if (correctLines == 1 && currentProtocol == "dsdv")
      {
        currentReceivedPackets = vectorIndex->receivedPacketsDsdv;
        currentTime = vectorIndex->time;
      }
      else if (correctLines == 2 && currentProtocol == "aodv")
      {
        previousReceivedPackets = vectorIndex->receivedPacketsAodv;
        previousTime = vectorIndex->time;
      }
      else if (correctLines == 2 && currentProtocol == "dsdv")
      {
        previousReceivedPackets = vectorIndex->receivedPacketsDsdv;
        previousTime = vectorIndex->time;
      }
      else
      {
        // cout << "Something went terribly wrong working out which protocol is in use!";
      }
    }
    vectorIndex--;
  }

  if (correctLines < 2)
  {
    return -1;
  }
  else
  {
    return (currentReceivedPackets - previousReceivedPackets) / (currentTime - previousTime);
  }
}

bool
CogMan::WillBeInRange(int sNode, int mNode, double atTime)
{
  Vector sNodePredictedPosition;
  Vector mNodePredictedPosition;
  sNodePredictedPosition = PredictPosition(sNode, atTime);
  mNodePredictedPosition = PredictPosition(mNode, atTime);
  double distance = GetDistance(sNodePredictedPosition, mNodePredictedPosition);
  //cout << "\ndistance should be " << distance << "m";

  return InRange(distance);
}

bool
CogMan::WillBeAlive(int nodeId, double atTime)
{
  double remainingEnergy = GetRemainingEnergy(nodeId);

  Print("WillBeAlive",1);
  //double remainingEnergy      = GetRemainingEnergy(nodeId);
  // cout << "remaining energy is : " << remainingEnergy;
  double elapsedTime          = GetTimeNow('s');
  double usedEnergy           = initialEnergy - remainingEnergy;
  // cout << " used energy : " << usedEnergy;
  double energyUsedPerSecond  = usedEnergy / elapsedTime;
  // cout << " energy per s : " << energyUsedPerSecond;
  double powerNeeded = energyUsedPerSecond * (atTime - elapsedTime);
  // cout << " powerNeeded : " << powerNeeded;

  Print(message.str(),14);
  return remainingEnergy > powerNeeded;
  Print("GetPredictedTimePowerUntil",2);
}

bool
CogMan::RecordRelationshipState(int sNode, int mNode)
{
  Print("RecordRelationshipState",1);
  double energyLevel  = GetRemainingEnergy(sNode);
  double timeNow      = GetTimeNow('s');
  Vector locSNode     = GetNodePosition(sNode);
  Vector locMNode     = GetNodePosition(mNode);
  double range        = GetDistance(locSNode, locMNode);

  relationshipStates.push_back( //------------------------------------//
   {  sNode,                    //  A vector called relationshipState //
      mNode,                    //  to store the relationshipState(s) //
      timeNow,                  //                                    //
      range,                    //  The structure has variables       //
      energyLevel,              //    sourceNode        int           //
   });                          //    managerNodeId     int           //
                                //    time              double        //
                                //    range             double        //
                                //    energyLevel       double        //
                                //------------------------------------//
  Print("RecordRelationshipState",2);
}

void
CogMan::RecordPacketRate(double timeNow, int source, double sent, double received, double
dropped)
{
  Print("RecordPacketRate",1);

  packetRates.push_back(       //------------------------------------//
   {                           //  A vector called packetRates       //
       timeNow,                //  to store the current packetRate   //
       source,                 //                                    //
       sent,                   //  The structure has variables       //
       received,               //    time              double        //
       dropped,                //    source            int           //
   });                         //    sent              double        //
                               //    received          double        //
                               //    dropped           double        //
  Print("RecordPacketRate",2); //------------------------------------//
}



int
CogMan::GetCurrentReceivedPacketsForSource(int nodeId, double timeFrom, double timeUntil)
{
  Print("GetCurrentReceivedPacketsForSource",1);
  int pr = 0;
  Ipv4Address nodeIP = GetIPAddressFromNodeId(nodeId);

  vector<IpAddresses>::iterator vectorIndex;
  vectorIndex = packetReceivedByCurrentProtocol.end()-1;

  while (packetReceivedByCurrentProtocol.size() > 0 && vectorIndex->time >= timeFrom)
  {
    if (vectorIndex->source == nodeIP && vectorIndex->time <= timeUntil)
    {
      pr++;
    }
    vectorIndex--;
  }

  Print("GetCurrentReceivedPacketsForSource",2);
  return pr;
}

int
CogMan::GetCurrentReceivedPacketsForDestinationFromSource(int sNode, int dNode, double
timeFrom, double timeUntil)
{
  Print("GetCurrentReceivedPacketsForSource",1);
  int pr = 0;
  Ipv4Address sNodeIP = GetIPAddressFromNodeId(sNode);
  Ipv4Address dNodeIP = GetIPAddressFromNodeId(dNode);

  vector<IpAddresses>::iterator vectorIndex;
  vectorIndex = packetReceivedByCurrentProtocol.end()-1;

  while (packetReceivedByCurrentProtocol.size() > 0 && vectorIndex->time >= timeFrom)
  {
    if (vectorIndex->destination == dNodeIP &&
        vectorIndex->source == sNodeIP &&
        vectorIndex->time <= timeUntil)
    {
      pr++;
    }
    vectorIndex--;
  }
  Print("GetCurrentReceivedPacketsForSource",2);
  return pr;
}


int
CogMan::GetCurrentSentPacketsForSource(int nodeId, double timeFrom, double timeUntil)
{
  Print("GetCurrentSentPacketsForSource",1);
  int ps = 0;
  Ipv4Address nodeIP = GetIPAddressFromNodeId(nodeId);

  //cout << "you are getting here at least!";

  vector<IpAddresses>::iterator vectorIndex;
  vectorIndex = packetSentByCurrentProtocol.end()-1;

  //cout << "\n vector size is : " << packetSentByCurrentProtocol.size();

  while (packetSentByCurrentProtocol.size() > 0 && vectorIndex->time >= timeFrom)
  {
    if (vectorIndex->source == nodeIP && vectorIndex->time <= timeUntil)
    {
      ps++;
    }
    vectorIndex--;
  }

  Print("GetCurrentSentPacketsForSource",2);
  return ps;
    //return (currentReceivedPackets - previousReceivedPackets) / (currentTime -
previousTime);
}

int
CogMan::GetCurrentDroppedPacketsForSource(int nodeId, double timeFrom, double timeUntil)
{
  Print("GetCurrentDroppedPacketsForSource",1);
  int pd = 0;
  Ipv4Address nodeIP = GetIPAddressFromNodeId(nodeId);

  //cout << "you are getting here at least!";

  vector<IpAddresses>::iterator vectorIndex;
  vectorIndex = packetDroppedByCurrentProtocol.end()-1;

  //cout << "\n vector size is : " << packetDroppedByCurrentProtocol.size();

  while (packetDroppedByCurrentProtocol.size() > 0 && vectorIndex->time >= timeFrom)
  {
    if (vectorIndex->source == nodeIP && vectorIndex->time <= timeUntil)
    {
      pd++;
    }
    vectorIndex--;
  }

  Print("GetCurrentDroppedPacketsForSource",2);
  return pd;
    //return (currentReceivedPackets - previousReceivedPackets) / (currentTime -
previousTime);
}

string
CogMan::GetAction(char gutFeeling, int mNode, double droPerSec, double recPerSec, double
senPerSec, double density, int channel, string currentProtocol)
{
  cout << "\n-->";
  vector<cognitivity>::iterator vectorIndex;
  vectorIndex = managementNodeMemory.end();
  double timeNow = GetTimeNow('s');
  double backOffUntil = simulationTime;

  double closestDistance = 1000;
  int pastCount = 6;
  int strikeCount = 0;
  bool switchingProtocolAllowed = true;
  bool changeChannelAllowed = true;
  bool moveNodeAllowed = true;

  int MCount = 0;
  int PCount = 0;
  int CCount = 0;

  string action = "";

  while ( (vectorIndex > managementNodeMemory.begin()) &&
          (strikeCount < 3) &&
          (pastCount > 0) )
  {
    vectorIndex--;
    if (vectorIndex->mNode == mNode)
    {
      if (vectorIndex->backOffTime > timeNow)
      {
        cout << "STRIKE " << strikeCount;
        strikeCount++;
        if (vectorIndex->request == 'P')
          switchingProtocolAllowed = false;
        if (vectorIndex->request == 'C')
          changeChannelAllowed = false;
        if (vectorIndex->request == 'M')
          moveNodeAllowed = false;

        if (backOffUntil < vectorIndex->backOffTime)
        {
          backOffUntil = vectorIndex->backOffTime;
        }
      }
      else if (vectorIndex->performance > 0)
      {
        if ((vectorIndex->request == 'P' && switchingProtocolAllowed == true) ||
            (vectorIndex->request == 'C' && changeChannelAllowed == true)     ||
            (vectorIndex->request == 'M' && moveNodeAllowed == true) )
        {
          double MemoryDroPerSec = vectorIndex->dropRate;
          double MemoryRecPerSec = vectorIndex->receiveRate;
          double MemorySenPerSec = vectorIndex->sendRate;
          double MemoryDensity   = vectorIndex->density;
          int    MemoryChannel   = vectorIndex->channel;
          string MemoryProtocol  = vectorIndex->routingProtocol;

          double droPerSecPercent = (droPerSec - MemoryDroPerSec) * (droPerSec - 
MemoryDroPerSec) * 0.1;
          double recPerSecPercent = (recPerSec - MemoryRecPerSec) * (recPerSec - 
MemoryRecPerSec) * 0.5;
          double senPerSecPercent = (senPerSec - MemorySenPerSec) * (senPerSec - 
MemorySenPerSec) * 0.4;

          double distance = (droPerSecPercent + recPerSecPercent + senPerSecPercent);

          if (distance < closestDistance)
          {
            closestDistance = distance;
            action = vectorIndex->request;
            cout << "action updated to: " << action;
          }
        }
      }
      else
      {
        if (vectorIndex->request == 'M') MCount = MCount + vectorIndex->performance;
        if (vectorIndex->request == 'P') PCount = PCount + vectorIndex->performance;
        if (vectorIndex->request == 'C') CCount = CCount + vectorIndex->performance;
        pastCount++;
      }
    }
  }

  if (strikeCount > 2)
  {
    cout << endl << timeNow << "s Manager " << mNode << " talks - told to back off until " << 
backOffUntil;
    action = ConvertDoubleToString(backOffUntil);
  }
  else if (action == "")
  {
    if (gutFeeling == 'P' && switchingProtocolAllowed == true)
        action = "switch protocol";
    if (gutFeeling == 'M' && moveNodeAllowed == true)
        action = "move node";
    if (gutFeeling == 'C' && changeChannelAllowed == true)
        action = "change channel";

    if (PCount > CCount && PCount > MCount && switchingProtocolAllowed == true) action =
"switch protocol";
    if (CCount > PCount && CCount > MCount && changeChannelAllowed     == true) action =
"change channel";
    if (MCount > PCount && MCount > CCount && moveNodeAllowed          == true) action = "move
node";

    while (action == "")
    {
      int randomAction = rand() % 3 + 1;
      cout << "random action" << randomAction;
      if (randomAction == 1 && switchingProtocolAllowed == true)
        action = "switch protocol";
      if (randomAction == 2 && changeChannelAllowed == true)
        action = "change channel";
      if (randomAction == 3 && moveNodeAllowed == true)
        action = "move node";
    }
  }
  else
  {
    // we have an action.
  }

  if (action == "C") action = "change channel";
  if (action == "P") action = "switch protocol";
  if (action == "M") action = "move node";

  cout << endl << timeNow << "s Manager " << mNode << " talks - Request: " << action;


  //PrintVectorOfMemoryEntries();
  return action;
}


void
CogMan::RecordActionRequest(double timeNow, int sNode, int mNode, double droPerSec,
                            double recPerSec, double senPerSec, double density,
                            int channel, string currentProtocol, char request)
{
  Print("RecordActionRequest",1);
                                      //----------------------------------------//
  managementNodeMemory.push_back({    //  A vector called managementNodeMemory  //
                                      //  to store the cognitivity data         //
      timeNow,                        //                                        //
      sNode,                          //  The structure has variables           //
      mNode,                          //    time             double             //
      droPerSec,                      //    sNode            int                //
      recPerSec,                      //    mNode            double             //
      senPerSec,                      //    dropRate         double             //
      density,                        //    receiveRate      double             //
      channel,                        //    sendRate         double             //
      currentProtocol,                //    density          double             //
      request                   });   //    channel          int                //
                                      //    routingProtocol  string             //
                                      //    request          char               //
  Print("RecordActionRequest",2);     //----------------------------------------//

  //PrintVectorOfMemoryEntries();
}



double
CogMan::GetPeriodToStablise(string action)
{
  double waitTime = 0;
  if (action == "switchProtocolToDSDV")
  {
    waitTime = 3;
  }
  if (action == "switchProtocol")
  {
    waitTime = 3;
  }
  if (action == "changeChannel")
  {
    waitTime = 3;
  }
  if (action == "moveControlNode")
  {
    waitTime = 3;
  }
  timeToStabilize = waitTime + GetTimeNow('s');
  return waitTime;
}

void
CogMan::RequestAction(double atTime, int mNode, char request, double prr)
{
  double timeNow = GetTimeNow('s');
  // add request to list
  newRequests.push_back({
    atTime,
    mNode,
    request,
    prr
    });

  // acknowledge request
  vector<cognitivity>::iterator vectorIndex;
  vectorIndex = managementNodeMemory.end();

  bool found = false;
  while (!found && vectorIndex > managementNodeMemory.begin())
  {
    vectorIndex--;
    if ((vectorIndex->atTime  == atTime ) &&
        (vectorIndex->mNode   == mNode  ) &&
        (vectorIndex->request == request))
        {
          vectorIndex->requestProgress = 'A';
          cout << endl << timeNow << "s NCC talks - Request Acknowledged from node " << mNode;
          found = true;
        }
  }
  //PrintVectorOfMemoryEntries();
}

void
CogMan::PrintAllPacketStats(double fromTime, double untilTime)
{
  vector<AllPackets>::iterator vectorIndex;
  vectorIndex = allPackets.end();

  int dsdvReceived  = 0;
  int dsdvSent      = 0;
  int dsdvDropped   = 0;

  int aodvReceived  = 0;
  int aodvSent      = 0;
  int aodvDropped   = 0;


  while (vectorIndex > allPackets.begin() )
  {
    vectorIndex--;
    while (vectorIndex->time > untilTime)
    {
      vectorIndex--;
    }
    if (vectorIndex->time < fromTime)
    {
      break; // leave loop
    }

    string protocol = GetProtocol(vectorIndex->destination);

    if (protocol == "aodv")
    {
      if (vectorIndex->action == 'S')
      {
        aodvSent++;
      }
      else if (vectorIndex->action == 'D')
      {
        aodvDropped++;
      }
      else if (vectorIndex->action == 'R')
      {
        aodvReceived++;
      }
    }
    else if (protocol == "dsdv")
    {
      if (vectorIndex->action == 'S')
      {
        dsdvSent++;
      }
      else if (vectorIndex->action == 'D')
      {
        dsdvDropped++;
      }
      else if (vectorIndex->action == 'R')
      {
        dsdvReceived++;
      }
    }
  }
  // cout << "\nFor the period " << fromTime << " until " << untilTime << endl;
  // cout << "     AODV     \n";
  // cout << "==============\n";
  // cout << " Received : " << aodvReceived << endl;
  // cout << " Sent     : " << aodvSent << endl;
  // cout << " Dropped  : " << aodvDropped << endl << endl;
  // cout << "     DSDV     \n";
  // cout << "==============\n";
  // cout << " Received : " << dsdvReceived << endl;
  // cout << " Sent     : " << dsdvSent << endl;
  // cout << " Dropped  : " << dsdvDropped << endl;
}

void
CogMan::Management(int sNode, int mNode)
{
  Print("Management",1);

  // do below if source is still transmitting

  double timeNow      = GetTimeNow('s');
  //cout << "\n\nAt " << timeNow << " you are in management!\n";

  bool InRange      = WillBeInRange(sNode, mNode, timeNow + checkingIntervalTime);
  bool EnoughPower  = WillBeAlive(mNode, timeNow + checkingIntervalTime);
  //bool Transmitting = StillTransmitting(sNode);

  if (StillTransmitting(sNode))
  {
    if (InRange && EnoughPower)
    {
      RecordRelationshipState(sNode, mNode);

      int packetsReceived = GetCurrentReceivedPacketsForSource(sNode, (timeNow-
checkingIntervalTime), timeNow);
      int packetsSent     = GetCurrentSentPacketsForSource    (sNode, (timeNow-
checkingIntervalTime), timeNow);
      int packetsDropped  = GetCurrentDroppedPacketsForSource (sNode, (timeNow-
checkingIntervalTime), timeNow);
      double recPerSec    = packetsReceived / checkingIntervalTime;
      double senPerSec    = packetsSent     / checkingIntervalTime;
      double droPerSec    = packetsDropped  / checkingIntervalTime;

      double previousRecPerSec = 0;
      double previousSenPerSec = 0;
      double previousDroPerSec = 0;
      double previousTime      = 0;

      double currentFlowRate   = 0;
      double previousFlowRate  = 0;

      double checkTime = 0;  // used to allow network to stabalise.

      // get the previous rate
      vector<packetRate>::iterator vectorIndex;
      vectorIndex = packetRates.end();

      while (vectorIndex > packetRates.begin() && packetRates.size() > 0)
      {
        vectorIndex--;
        if (vectorIndex->source == sNode)
        {
          previousRecPerSec = vectorIndex->received;
          previousSenPerSec = vectorIndex->sent;
          previousDroPerSec = vectorIndex->dropped;
          previousTime      = vectorIndex->time;
          break; // break out of while loop when found
        }
      }

      RecordPacketRate(timeNow, sNode, senPerSec, recPerSec, droPerSec);

      if (recPerSec > maxReceivingRate)
      {
        maxReceivingRate = recPerSec;
      }
      if (senPerSec != 0)
      {
      currentFlowRate  = (recPerSec / senPerSec) * 100;
      }

      if (previousSenPerSec !=0)
      previousFlowRate = (previousRecPerSec / previousSenPerSec) * 100;

      /*
      cout << "\n the period is from " << timeNow-checkingIntervalTime << " until " <<
timeNow;
      cout << "\n packets received : " << packetsReceived;
      cout << "\n packets sent     : " << packetsSent;
      cout << "\n packets dropped  : " << packetsDropped << endl << endl;
      cout << "Rates Per Second";
      cout << "\n packets received : " << recPerSec;
      cout << "\n packets sent     : " << senPerSec;
      cout << "\n packets dropped  : " << droPerSec;
      cout << "\n flow rate        : " << currentFlowRate << endl << endl;
      cout << "Previous Rates Per Second";
      cout << "\n packets received : " << previousRecPerSec;
      cout << "\n packets sent     : " << previousSenPerSec;
      cout << "\n packets dropped  : " << previousDroPerSec;
      cout << "\n flow rate        : " << previousFlowRate;
      cout << endl << endl;
      */

      PrintAllPacketStats(timeNow-checkingIntervalTime, timeNow);

      if (timeToStabilize < timeNow)
      {
        if (currentFlowRate == 0 || currentFlowRate < previousFlowRate)
        {
          double previousPreviousRecPerSec = 0;
          double previousPreviousSenPerSec = 0;
          double previousPreviousDroPerSec = 0;
          double previousPreviousTime      = 0;
          double previousPreviousFlowRate  = 0;

          while (vectorIndex > packetRates.begin() && packetRates.size() > 0)
          {
            vectorIndex--;
            if (vectorIndex->source == sNode)
            {
              previousPreviousRecPerSec = vectorIndex->received;
              previousPreviousSenPerSec = vectorIndex->sent;
              previousPreviousDroPerSec = vectorIndex->dropped;
              previousPreviousTime      = vectorIndex->time;
              break; // break out of while loop when found
            }
          }

          if (previousPreviousSenPerSec !=0)
          previousPreviousFlowRate = (previousPreviousRecPerSec / previousPreviousSenPerSec) *
100;

          // cout << "Previous Previous Rates Per Second";
          // cout << "\n packets received : " << previousRecPerSec;
          // cout << "\n packets sent     : " << previousSenPerSec;
          // cout << "\n packets dropped  : " << previousDroPerSec;
          // cout << "\n flow rate        : " << previousFlowRate;
          // cout << endl << endl;

          if (previousPreviousFlowRate == 0 || previousFlowRate < previousPreviousFlowRate)
          {
            //cout << "\n performance has dropped twice!!!\n";

            /// Least squares approximation
            // Get predicted packet per second rate if trend continues.
            double a=0;
            double b=0;
            double c=0;
            double d=0;
            double o=0;
            double m=0;

            a=previousPreviousTime+previousTime+timeNow;
            b=previousPreviousRecPerSec+previousRecPerSec+recPerSec;

c=previousPreviousTime*previousPreviousRecPerSec+previousTime*previousRecPerSec+timeNow*recPer
Sec;

d=previousPreviousTime*previousPreviousTime+previousTime*previousTime+timeNow*timeNow;

            o=(b*d-a*c)/(3*d-a*a);
            m=(3*c-a*b)/(3*d-a*a);

            double nextTime=0;
            double estimatedRecPerSec=0;

            nextTime=timeNow-previousTime;
            estimatedRecPerSec=m*(timeNow+nextTime)+o;

            if (estimatedRecPerSec > maxReceivingRate)
            {
              estimatedRecPerSec = maxReceivingRate;
            }
            if (estimatedRecPerSec < 0)
            {
              estimatedRecPerSec = 0;
            }


            double density = GetDensityFromNode(mNode);
            double channel = GetChannel(mNode);

            char gutFeeling = 'C';
            if (currentFlowRate == 0 && previousFlowRate == 0 && previousPreviousFlowRate ==
0)
            {
              gutFeeling = 'M'; // move controlled node
            }
            else if (currentFlowRate > 0  && GetNumberOfNeighbours(mNode, aodvNodes) < 10)
            {
              gutFeeling = 'P'; // switch protocol
            }

            string action = GetAction(gutFeeling, mNode, droPerSec, recPerSec, senPerSec,
density, channel, currentProtocol);
```

```
            cout << "\n1.  Action returned from GetAction : " << action;

            char request;
            if (action == "switch protocol")
              request = 'P';
            if (action == "change channel")
              request = 'C';
            if (action == "move node")
              request = 'M';

            if (request == 'P' || request == 'C' || request == 'M')
            {
              RecordActionRequest(timeNow, sNode, mNode, droPerSec, recPerSec, senPerSec,
density, channel, currentProtocol, request);
              RequestAction(timeNow, mNode, request, estimatedRecPerSec);
                // check when we can deal with request
              if (timeToStabilize < timeNow && !NCCScheduled)
              {
                NetworkCommandCentre();
              }
              else
              {
                NCCScheduled = true;
                Simulator::Schedule(Seconds (timeToStabilize+4),
&CogMan::NetworkCommandCentre, this);
              }
            }
            else
            {
              cout << "\nYOU HAVE BEEN TOLD TO BACKOFF! " << action << " +-+-+-+" ;
            }

            cout << "\n The action returned is: " << action;
          }
          else
          {
            // cout << "\n performance has dropped\n";
          }
        }
        else if (currentFlowRate > previousFlowRate)
        {
          if (!reportedIncrease)
          {
            cout << GetTimeNow('s') << "s Performance increases\n";
            reportedIncrease = true;
            reportedDrop = false;
            reportedStabalise = false;
          }
        }
        else
        {
          if (!reportedStabalise)
          {
            cout << GetTimeNow('s') << "s Performance has stabalised\n";
            reportedStabalise = true;
            reportedIncrease = false;
            reportedDrop = false;
          }
        }


        if (checkTime==0)
        {
          Simulator::Schedule(Seconds (checkingIntervalTime), &CogMan::Management, this,
sNode, mNode);
        }
        else
        {
          Simulator::Schedule(Seconds (checkTime+1), &CogMan::Management, this, sNode, mNode);
        }
      }
      else
      {
        double checkTime = GetTimeNow('s') - timeToStabilize;
        Simulator::Schedule(Seconds (1), &CogMan::Management, this, sNode, mNode);
      }
```

```
    }
    else
    {
      // PERFORM RE-ELECTION
    }
  }
  Print("Management",2);
}

void
CogMan::TotalNumberOfPackets()
{
 vector<memorySystemState>::iterator vectorIndex;
 vectorIndex = systemState.end()-1;

 totalAodvPacketsReceived=vectorIndex->receivedPacketsAodv;
 totalDsdvPacketsReceived=vectorIndex->receivedPacketsDsdv;

 /*
 vector<memorySystemState>::iterator vectorIndex3;
 vectorIndex3 = systemState.begin();

 int receivedAodv=0;
 int receivedDsdv=0;
 int receivedPreviousAodv=0;
 int receivedPreviousDsdv=0;
 int totalAodv=0;
 int totalDsdv=0;


 while (vectorIndex3<=systemState.end()-1)
 {
  if (vectorIndex3->protocol=="aodv")
    {
    if (vectorIndex3!=systemState.begin())
      {
      //receivedPreviousAodv=vectorIndex3->receivedPacketsAodv;
        cout << "VectorIndex3 is " << vectorIndex3 << " and receivedPacketsAodv is "
         << vectorIndex3->receivedPacketsAodv <<endl;
      }
 //   if (vectorIndex==systemState.begin())
   //   {

     // }


     vectorIndex3++;
     receivedAodv=vectorIndex3->receivedPacketsAodv;
     totalAodv=totalAodv+(receivedAodv-receivedPreviousAodv);
    }
 if (vectorIndex3->protocol=="dsdv")
    {
    if (vectorIndex3!=systemState.begin())
      {
      receivedPreviousDsdv=vectorIndex3->receivedPacketsDsdv;
      }
 //   if (vectorIndex==systemState.begin())
   //   {

     // }


     vectorIndex3++;
     receivedDsdv=vectorIndex3->receivedPacketsDsdv;
     totalDsdv=totalDsdv+(receivedDsdv-receivedPreviousDsdv);
    }

 }

*/
cout << "Received by switching is " << totalSwitchingReceived << endl;
cout << "Received by AODV is " << totalAodvPacketsReceived << endl;
cout << "Received by DSDV is " << totalDsdvPacketsReceived << endl;

PrintNodePosition(0);
PrintNodePosition(numOfNodes-1);
```

```
cout << "Received by switching on this interval " << totalSwitchingReceived -
SpreviousReceived << endl;
cout << "Received by AODV on this interval is " << totalAodvPacketsReceived -
ApreviousReceived<< endl;
cout << "Received by DSDV on this interval  is " << totalDsdvPacketsReceived -
DpreviousReceived << endl;
cout << " Current protocol is " << currentProtocol << endl;

if ((totalAodvPacketsReceived - ApreviousReceived) > (totalDsdvPacketsReceived -
DpreviousReceived)
        && currentProtocol=="aodv")
        {
                cout << " HURRAY we are on the best protocol (aodv)!!!" << endl;
                counterGoodJob++;
                counterGoodAodv++;
        }

if ((totalDsdvPacketsReceived - DpreviousReceived) > (totalAodvPacketsReceived -
ApreviousReceived)
        && currentProtocol=="dsdv")
        {
                cout << " HURRAY we are on the best protocol (dsdv)!!!" << endl;
                counterGoodJob++;
                counterGoodDsdv++;
        }

        if ((totalAodvPacketsReceived - ApreviousReceived) < (totalDsdvPacketsReceived -
DpreviousReceived)
        && currentProtocol=="aodv")
        {
                cout << " HEY, what's wrong??? we are NOT on the best protocol (dsdv)!!!" <<
endl;
                 counterGoodDsdv++;
        }

if ((totalDsdvPacketsReceived - DpreviousReceived) < (totalAodvPacketsReceived -
ApreviousReceived)
        && currentProtocol=="dsdv")
        {
                cout << " HEY, what's wrong??? we are NOT on  the best protocol (aodv)!!!" <<
endl;
                counterGoodAodv++;
        }

        if ((totalAodvPacketsReceived - ApreviousReceived) == (totalDsdvPacketsReceived -
DpreviousReceived)
        )
        {
                cout << " HEY, we are fine, we are on the best protocol since they are working
equally good!!!"
                << endl;
                counterGoodJob++;
        }

 if ( totalAodvPacketsReceived > totalDsdvPacketsReceived  && totalAodvPacketsReceived >
totalSwitchingReceived)
 {

  cout << " Aodv was working better! " << endl;


   //if (totalDsdvPacketsReceived > totalSwitchingReceived) // Switching < Dsdv <Aodv
   //{
        // double percBetter = ((double)totalSwitchingReceived -
(double)totalDsdvPacketsReceived)/((double)totalDsdvPacketsReceived -
          //   (double)totalAodvPacketsReceived );

          double k = 100*(totalAodvPacketsReceived -
            totalDsdvPacketsReceived)/(totalDsdvPacketsReceived) ;
          double kk = 100*(totalAodvPacketsReceived -
            totalSwitchingReceived)/(totalDsdvPacketsReceived);
          double kkk = 100*(totalAodvPacketsReceived -
            totalDsdvPacketsReceived)/(totalSwitchingReceived) ;
          double kkkk = 100*(totalAodvPacketsReceived -
            totalSwitchingReceived)/(totalSwitchingReceived) ;
          double kkkkk = 100*(totalAodvPacketsReceived -
            totalDsdvPacketsReceived)/(totalDsdvPacketsReceived - totalSwitchingReceived) ;
          double kkkkkk = 100*(totalAodvPacketsReceived -
            totalSwitchingReceived)/(totalDsdvPacketsReceived - totalSwitchingReceived);

          cout << "Aodv was working " << k
          << " percent better than Dsdv w.r.t DSDV"<<endl;

          cout << "Aodv was working " << kk
          << " percent better than Switching w.r.t DSDV"<<endl;

          cout << "Aodv was working " << kkk
          << " percent better than Dsdv w.r.t Switching"<<endl;

          cout << "Aodv was working " << kkkk
          << " percent better than Switching w.r.t Switching"<<endl;

          cout << "Aodv was working " << kkkkk
          << " percent better than Dsdv Relative to First Improvement"<<endl;

          cout << "Aodv was working " << kkkkkk
          << " percent better than Switching Relative to First Improvement"<<endl;



//   }

 }
 if ( totalDsdvPacketsReceived > totalAodvPacketsReceived && totalDsdvPacketsReceived >
totalSwitchingReceived)
 {
  cout << " Dsdv was working better! " << endl;


          double k = 100*(totalDsdvPacketsReceived -
            totalAodvPacketsReceived)/(totalAodvPacketsReceived) ;
          double kk = 100*(totalDsdvPacketsReceived -
            totalSwitchingReceived)/(totalAodvPacketsReceived) ;
          double kkk = 100*(totalDsdvPacketsReceived -
            totalAodvPacketsReceived)/(totalSwitchingReceived) ;
          double kkkk = 100*(totalDsdvPacketsReceived -
            totalSwitchingReceived)/(totalSwitchingReceived) ;
          double kkkkk = 100*(totalDsdvPacketsReceived -
            totalAodvPacketsReceived)/(totalAodvPacketsReceived - totalSwitchingReceived) ;
          double kkkkkk = 100*(totalDsdvPacketsReceived -
            totalSwitchingReceived)/(totalAodvPacketsReceived - totalSwitchingReceived) ;

  cout << "Dsdv was working " << k
          << " percent better than Aodv w.r.t AoDV"<<endl;

          cout << "Dsdv was working " << kk
          << " percent better than Switching w.r.t AoDV"<<endl;

          cout << "Dsdv was working " << kkk
          << " percent better than Aodv w.r.t Switching"<<endl;

          cout << "Dsdv was working " << kkkk
          << " percent better than Switching w.r.t Switching"<<endl;

          cout << "Dsdv was working " << kkkkk
          << " percent better than Aodv Relative to First Improvement"<<endl;

          cout << "Dsdv was working " << kkkkkk
          << " percent better than Switching Relative to First Improvement"<<endl;

 }

 if ( totalSwitchingReceived > totalDsdvPacketsReceived && totalSwitchingReceived >
totalAodvPacketsReceived)
 {
  cout << " Switching was working better!  WIN WIN WIN " << totalSwitchingReceived -
           totalAodvPacketsReceived << " packets over Aodv and " << totalSwitchingReceived -
           totalDsdvPacketsReceived << " packets over Dsdv " <<endl;

          double k = 100*(totalSwitchingReceived -
            totalDsdvPacketsReceived)/(totalAodvPacketsReceived) ;
          double kk = 100*(totalSwitchingReceived -
            totalAodvPacketsReceived)/(totalDsdvPacketsReceived) ;
          double kkk = 100*(totalSwitchingReceived -
            totalDsdvPacketsReceived)/(totalDsdvPacketsReceived) ;
          double kkkk =  100*(totalSwitchingReceived -
            totalAodvPacketsReceived)/(totalAodvPacketsReceived);

          double kkkkk = 100*(totalSwitchingReceived -
            totalDsdvPacketsReceived)/(totalAodvPacketsReceived -
            totalDsdvPacketsReceived) ;
          double kkkkkk = 100*(totalSwitchingReceived -
            totalAodvPacketsReceived)/(totalAodvPacketsReceived -
            totalDsdvPacketsReceived);


               double kkkkkkk = 100*(totalSwitchingReceived -
            totalAodvPacketsReceived)/(totalDsdvPacketsReceived -
            totalAodvPacketsReceived) ;
          double kkkkkkkk = 100*(totalSwitchingReceived -
            totalDsdvPacketsReceived)/(totalDsdvPacketsReceived -
            totalAodvPacketsReceived);

                 cout << "Switching was working " << k
          << "% =(s-d)/a"<<endl;

          cout << "Switching was working " << kk
          << "% =(s-a)/d"<<endl;

          cout << "Switching was working " << kkk
          << "% =(s-d)/d"<<endl;

          cout << "Switching was working " << kkkk
          << "% =(s-a)/a"<<endl;

          if (totalAodvPacketsReceived > totalDsdvPacketsReceived)
          {

          cout << "Switching was working " << kkkkk
          << "% (s-d)/ (a-d)"<<endl;

          cout << "Switching was working " << kkkkkk
          << "% (s-a)/(Aodv-Dsdv)"<<endl;
          }
          if (totalDsdvPacketsReceived > totalAodvPacketsReceived)
          {

          cout << "Switching was working " << kkkkkkk
          << "% (s-a)/ (Dsdv-Aodv)"<<endl;

          cout << "Switching was working " << kkkkkkkk
          << "% (s-d)/ (Dsdv-Aodv)"<<endl;
          }
 }

 ApreviousReceived=totalAodvPacketsReceived;
 DpreviousReceived=totalDsdvPacketsReceived;
 SpreviousReceived=totalSwitchingReceived;

}


void
CogMan::RecordElectionWinner(int sourceNodeId, int managerNodeId, double timeNow,
                  double timeInRangeUntil, double timePowerUntil, double density)
{
  Print("ElectionWinner",1);
  Print("Winner of Election",11); //------------------------------------//
                                  //  A vector called managersList      //
                                  //  to store the managerNode(s)       //
                                  //                                    //
  managersList.push_back(         //  The structure has variables       //
    {  sourceNodeId,              //    sourceNode        int           //
       managerNodeId,             //    managerNodeId     int           //
       timeNow,                   //    timeElected       double        //
       timeInRangeUntil,          //    timeInRangeUntil  double        //
       timePowerUntil,            //    timePowerUntil    double        //
       density                    //    density           double        //
    });                           //------------------------------------//
  vector<managerNode>::iterator vectorIndex;
  vectorIndex = managersList.end()-1;

  message << "Election Winner is node " << vectorIndex->managerNodeId << ".\n "
  << "At time: " << vectorIndex->timeElected << " will be in range until "
  << vectorIndex->timeInRangeUntil << " current density is " << vectorIndex->density;
  Print(message.str(),11);
  //cin.get();
  Management(sourceNodeId, managerNodeId);
  Print("ElectionWinner",2);
}

void
CogMan::RecordElectionCandidate(int sourceNodeId, int managerNodeId, double timeNow,
                     double timeInRangeUntil, double timePowerUntil, double density)
{
  Print("RecordElectionCandidate",1);
  Print("Candidate for Election",11);  //------------------------------------//
                                       //  A vector called managerNode       //
                                       //  to store the election candidate(s) //
                                       //                                    //
  candidatesList.push_back(            //  The structure has variables       //
    {  sourceNodeId,                   //    sourceNode        int           //
       managerNodeId,                  //    managerNodeId     int           //
       timeNow,                        //    timeElected       double        //
       timeInRangeUntil,               //    timeInRangeUntil  double        //
       timePowerUntil,                 //    timePowerUntil    double        //
       density                         //    density           double        //
    });                                //------------------------------------//

  vector<managerNode>::iterator vectorIndex;
  vectorIndex = candidatesList.end()-1;

  message << "Election Candidate is node " << vectorIndex->managerNodeId << ".\n "
  << "At time: " << vectorIndex->timeElected << " will be in range until "
  << vectorIndex->timeInRangeUntil << " current density is " << vectorIndex->density;
  Print(message.str(),11);

  cout << GetTimeNow('s') << "s Node " << vectorIndex->managerNodeId << " identified as
candidate management node for node " << vectorIndex->sourceNodeId << endl;

  Print("RecordElectionCandidate",2);
}

void
CogMan::RecordElectionCandidatePercents(int sourceNodeId, int managerNodeId, double timeNow,
  double timeInRangeUntil, double timePowerUntil, double density)
{
  Print("RecordElectionCandidatePercents",1);
  Print("Candidate for Election",11);               //------------------------------------//
                                                    //  A vector called managersList      //
                                                    //  to store the managerNode(s)       //
                                                    //                                    //
  candidatesListPercents.push_back(                 //  The structure has variables       //
    {  sourceNodeId,                                //    sourceNode        int           //
       managerNodeId,                               //    managerNodeId     int           //
       timeNow,                                     //    timeElected       double        //
       timeInRangeUntil,                            //    timeInRangeUntil  double        //
       timePowerUntil,                              //    timePowerUntil    double        //
       density                                      //    density           double        //
    });                                             //------------------------------------//

  vector<managerNode>::iterator vectorIndex;
  vectorIndex = candidatesListPercents.end()-1;

  message << "Election Candidate is node " << vectorIndex->managerNodeId << ".\n "
  << "At time: " << vectorIndex->timeElected << " will be in range until "
  << vectorIndex->timeInRangeUntil << " current density is " << vectorIndex->density;
  Print(message.str(),11);

  Print("RecordElectionCandidatePercents",2);
}

void
CogMan::Election() // nodeId is source node sending data
{
  Print("Election",1);
  Print("We are running the election process", 11);
  double timeNow = GetTimeNow('s');


  vector<traffic>::iterator vectorIndex;
  vectorIndex = myTrafficFlows.begin();
  // cout << "myTrafficFlows size is: " << myTrafficFlows.size();
  while (vectorIndex < myTrafficFlows.end())
  {
    if (vectorIndex->timeStart < timeNow &&
        vectorIndex->timeStop > timeNow)
    {
      Vector sourcePosition = GetNodePosition(vectorIndex->sNode);
      for (int dest=0; dest < numOfNodes; dest++)
      {
        if (dest != vectorIndex->sNode)
        {
          Vector destinationPosition = GetNodePosition(dest);
          if (InRange(GetDistance(sourcePosition,destinationPosition)))
          {
            // add to candidates list.
            double timeInRangeUntil = GetPredictedMaximalTimeInRange(vectorIndex->sNode,
dest);
            double timePowerUntil   = GetPredictedTimePowerUntil(dest);
            double density          = GetDensityFromNode(dest);
            RecordElectionCandidate(vectorIndex->sNode,
                                    dest, timeNow, timeInRangeUntil,
                                    timePowerUntil, density);
          }
        }
      }
    }


    if (candidatesList.size() > 0)
    {
      SelectManagementNode(vectorIndex->sNode, timeNow);
    }
    else
    {
      cout << "NO CANDIDATES FOR CURRENT SOURCE NODE!!! -- RUN ELECTION AGAIN!";
    }
    vectorIndex++;
  }

  //cout << "candidatesListSize is " << candidatesList.size() <<endl;

  //vector<managerNode>::iterator vectorIndex2;
  //vectorIndex2 = candidatesList.begin();



  //double maxTimeInRangePercents = maxTimeInRange

  // now we have a candidateList that looks like:
  // --------------------------------------------------
  // source | dest | time | inRangeUntil | powerUntil -
  //    0   |   1  | 5.01 |      34      |    98      -
  //    0   |   2  | 5.01 |      29      |    198     -
  //    0   |   5  | 5.01 |      34      |    98      -
  //    0   |   7  | 5.01 |      55      |    102     -
  // --------------------------------------------------
  // PrintVectorOfManagerNode(candidatesList);
  Print("Election",2);
}

void
CogMan::SelectManagementNode(int sNode, double timeNow)
{
  Print("SelectManagementNode",1);
  Print("We are selecting the most suitable candidate", 11);
  double maxTimeInRange=0;
  double maxTimeInPower=0;
  double maxDensity=0;

  vector<managerNode>::iterator vectorIndex;
  vectorIndex = candidatesList.begin();

  while (vectorIndex < candidatesList.end() )
  {
    if (vectorIndex->sourceNodeId == sNode && vectorIndex->timeElected == timeNow)
    {
      if (maxTimeInRange < vectorIndex->timeInRangeUntil)
      {
        maxTimeInRange = vectorIndex->timeInRangeUntil;
      }
      if (maxTimeInPower < vectorIndex->timePowerUntil)
      {
        maxTimeInPower = vectorIndex->timePowerUntil;
      }
      if (maxDensity < vectorIndex->density)
      {
        maxDensity = vectorIndex->density;
      }
         // cout << "maxDensity" << maxDensity<<endl;
    }
    vectorIndex++;
  }

  double highestScore = 0;
  int likelyManager  = -1;
  double timeInRangeUntilPercents = 0;
  double timePowerUntilPercents = 0;
  double densityPercents = 0;
  double timeInRangeUntil = 0;
  double timePowerUntil = 0;
  double density = 0;

  vectorIndex = candidatesList.begin();


  while (vectorIndex < candidatesList.end() )
  {
    if (vectorIndex->sourceNodeId == sNode && vectorIndex->timeElected == timeNow)
    {
      double score = 0;

      timeInRangeUntilPercents = (vectorIndex->timeInRangeUntil)*100/maxTimeInRange;
      timePowerUntilPercents = (vectorIndex->timePowerUntil)*100/maxTimeInPower;
      densityPercents = (vectorIndex->density)*100/maxDensity;

      score = (timeInRangeUntilPercents * timeInRangeWeight) +
              (timePowerUntilPercents * timeInPowerWeight) +
              (densityPercents * densityWeight);

      if (score > highestScore)
      {
        highestScore     = score;
        likelyManager    = vectorIndex->managerNodeId;
        timeInRangeUntil = vectorIndex->timeInRangeUntil;
        timePowerUntil   = vectorIndex->timePowerUntil;
        density          = vectorIndex->density;
      }
      //cout << "\nScore for candidate " << vectorIndex->managerNodeId << ": " << score;
    }
    vectorIndex++;
  }

  // cout << "myTrafficFlows size is: " << myTrafficFlows.size();


  // COPY
  RecordElectionWinner (sNode,
                        likelyManager,
                        timeNow,
                        timeInRangeUntil,
                        timePowerUntil,
                        density);

  vectorIndex = candidatesList.begin();

  cout << GetTimeNow('s') << "s Node " << likelyManager << " has been elected to manage node "
<< sNode << endl;
```

```
  /*
    while (vectorIndex < candidatesList.end())
    {
       double timeInRangeUntilPercents = vectorIndex->timeInRangeUntil;
       double timePowerUntilPercents   = vectorIndex->timePowerUntil;
       double densityPercents          = vectorIndex->density;


       timeInRangeUntilPercents = vectorIndex->timeInRangeUntil*100/maxTimeInRange;
       timePowerUntilPercents = vectorIndex->timePowerUntil*100/maxTimeInPower;
       densityPercents = vectorIndex->density*100/maxDensity;



       RecordElectionCandidatePercents(vectorIndex->sNode,
                                      dest, timeNow, timeInRangeUntilPercents,
                                      timePowerUntilPercents, densityPercents);

      vectorIndex++;



    }
  */

  // now we have a candidateList that looks like:
  // ---------------------------------------------
  // source | dest | time | inRangeUntil | powerUnit -
  //    0   |   1  | 5.01 |      34      |    98     -
  //    0   |   2  | 5.01 |      29      |    198    -
  //    0   |   5  | 5.01 |      34      |    98     -
  //    0   |   7  | 5.01 |      55      |    102    -
  // ---------------------------------------------
  //PrintVectorOfManagerNode(candidatesListPercents);
  Print("Election",2);


  // END of copy
}

double
CogMan::GetPredictedTimePowerUntil(int nodeId)
{
  Print("GetPredictedTimePowerUntil",1);
  double remainingEnergy = GetRemainingEnergy(nodeId);
  double elapsedTime = Simulator::Now().GetSeconds();
  double usedEnergy = initialEnergy - remainingEnergy;
  double energyUsedPerSecond = usedEnergy / elapsedTime;
  double remainingTimePowerUntil = remainingEnergy / energyUsedPerSecond;

  message << "Node " << nodeId << " uses approximately "
          << energyUsedPerSecond << "J per second - "
          << GetHoursFromSeconds(remainingTimePowerUntil) << "h "
          << GetMinutesFromSeconds(remainingTimePowerUntil) << "m "
          << GetSecondsFromSeconds(remainingTimePowerUntil) << "s "
          << "left before power depleation";
  Print(message.str(),14);
  return remainingTimePowerUntil;
  Print("GetPredictedTimePowerUntil",2);
}

string
CogMan::InRangeLongestDuration(NodeContainer nodesInRange, double duration, int sourceNode)
{
  double endTime = floor (Simulator::Now().GetSeconds())+duration;

  double fractionOfTime=1;
  // int inRangeForSecondss;
  // for each node in the container get how long we think it will be in range

  uint32_t maxTimeInRange = 0;
  string nodeInRangeLongest;


  for (unsigned int x=0; x<nodesInRange.GetN(); x++)
  {
    // get nodes pointer and then it's id
    Ptr<Node> p = nodesInRange.Get(x);
    uint32_t neighbourNode = p->GetId();
    for (double t=endTime-duration; t<=endTime; t+=fractionOfTime)
    {
      ns3::Vector predictedpositionD = PredictPosition(neighbourNode, t);
      ns3::Vector predictedpositionS = PredictPosition(sourceNode, t);

      double distance = GetDistance(predictedpositionD, predictedpositionS);

      if (InRange(distance))   // for now 50 is myRadius
      {
        if (maxTimeInRange < t)
        {
          maxTimeInRange=t;
          nodeInRangeLongest = static_cast<ostringstream*>(&(ostringstream() <<
neighbourNode))->str();
        }
        else if (maxTimeInRange == t)
        {
        string tmp = static_cast<ostringstream*>(&(ostringstream() << neighbourNode))->str();
        nodeInRangeLongest = nodeInRangeLongest + "," +  tmp;
        }
      }
    }
  }
  std::cout << "Max Time in Range is " << maxTimeInRange << std::endl;
  return nodeInRangeLongest;
}

void
CogMan::GetNeighboursInCommon (NodeContainer nodes, uint32_t nodeId1, uint32_t nodeId2)
{
  NodeContainer nodesInRange ;//= CheckNeighbours(nodes, nodeId1);
  NodeContainer nodesInRange2 ;//= CheckNeighbours(nodes, nodeId2);

  for (uint32_t x = 0; x<nodesInRange.GetN(); x++)
  {
    bool found = false;
    for (uint32_t n=0; n<nodesInRange2.GetN() && !found; n++)
    {
      if (nodesInRange.Get(x) == nodesInRange2.Get(n))
      {
        found = true;
        std::cout << "node: " << nodesInRange.Get(x) << " is in range of both nodes " <<
std::endl;
      }
    }
  }
}

void
CogMan::PrintDevices(NodeContainer myNodes, string containerName)
{
  Print("PrintDevices",1);
  message  << std::endl << std::endl << "Devices associated with " << containerName;
  Print(message.str(), 8);
  for (uint32_t i = 0; i<myNodes.GetN(); ++i)
  {
    message << i << "         ";
  }
  Print(message.str(),8);

  for (uint32_t i = 0; i<myNodes.GetN(); i++)
  {
  Ptr<NetDevice> myDevice = myNodes.Get(i)->GetDevice(0);
  message << myDevice << " ";
  }
  Print(message.str(),8);
  Print("PrintDevices",2);
}

Ipv4Address
CogMan::GetIPAddressFromNodeId(int nodeId)
{
       Print("GetIPAddressFromNodeId",1);
  Ptr<Node> myNode;
  if (currentProtocol == "aodv")
  {
    myNode = aodvNodes.Get(nodeId);
  }
  else if (currentProtocol == "dsdv")
  {
    myNode = dsdvNodes.Get(nodeId);
  }
  else
  {
    cout << "\n\nFATAL ERROR : COULD NOT GET CURRENT PROTOCOL!!!";
  }

  Ptr<Ipv4> ipv4 = myNode->GetObject<Ipv4>();
  Ipv4InterfaceAddress iaddr = ipv4->GetAddress (1,0);

  Print("GetIPAddressFromNodeId",2);
  return iaddr.GetLocal();
}

void
CogMan::DoStats()
{
  Print("DoStats",1);
  cout << "\nSimulation Time: " << simulationTime;
  cout << "\nNumber of participating nodes: " << numOfNodes << " at " << nodeSpeed << "mps";;
  cout << "\nNumber of Traffic Flows: " << myTrafficFlows.size() << " at " << dataRate;

    vector<traffic>::iterator vectorIndex;
    vectorIndex = myTrafficFlows.begin();

    while (vectorIndex < myTrafficFlows.end())
    {
      cout << endl << vectorIndex->sNode << "\t" <<
                        vectorIndex->dNode << "\t" <<
                        vectorIndex->timeStart << "\t" <<
                        vectorIndex->timeStop;
      vectorIndex++;
    }

  if (cognition)
  {
    cout << "\nCognition is ON";
    cout << "\nNumber of controllable nodes: " << numOfControlledNodes << " at " << cNodeSpeed
<< "mps";
  }
  else
  {
    cout << "\nCognition is OFF";
  }

  cout << "\nTransmission Radius is: " << myRadius;

  if (mobilityModel == 'L')
  {
    cout << "\nUsing Lake mobility scenario";
  }
  else if (mobilityModel == 'R')
  {
    cout << "\nUsing Random mobility scenario";
  }

  cout << "\nRecording node positions every " << recordPositionInterval << "seconds.";





  cout << "\n-----------AODV----------------- " << endl;
  cout << "Payload Data Sent       :"  << sendAodvDataPacket << endl;
  cout << "Control Data Sent       :"  << sendAodvControlPacket << endl;
  cout << "Total Packets Dropped   :"  << dropAodvTotal << endl;
  cout << "Payload Data Dropped    :"  << dropAodvDataPacket << endl;
  cout << "  Payload Broadcast     :"  << dropAodvFF << endl;
  cout << "  Unknown Route         :"  << dropAodvUR << endl;
  cout << "  Not for Receipient    :"  << dropAodvNFM << endl;
  cout << "  Other                 :"  << dropAodvOther << endl;
  cout << "Control Data Dropped    :"  << dropAodvControlPacket << endl;
  cout << "  Clear to send         :"  << dropAodvCTS << endl;
  cout << "  IP Route Request      :"  << dropAodvRQ << endl;
  cout << "  IP Route Reply        :"  << dropAodvRR << endl;
  cout << "  MAC Route Request     :"  << dropAodvMAC_RR << endl;
  cout << "  MAC Route Reply       :"  << dropAodvMAC_RR << endl;
  cout << "Total Data Received     :"  << totalAodvPacketsReceived << endl;
  cout << "---------------------------------- " << endl;


  cout << "\n-----------DSDV----------------- " << endl;
  cout << "Payload Data Sent       :"  << sendDsdvDataPacket << endl;
  cout << "Control Data Sent       :"  << sendDsdvControlPacket << endl;
  cout << "Total Packets Dropped   :"  << dropDsdvTotal << endl;
  cout << "Payload Data Dropped    :"  << dropDsdvDataPacket << endl;
  cout << "  Payload Broadcast     :"  << dropDsdvFF << endl;
  cout << "  Unknown Route         :"  << dropDsdvUR << endl;
  cout << "  Not for Receipient    :"  << dropDsdvNFM << endl;
  cout << "  Other                 :"  << dropDsdvOther << endl;
  cout << "Control Data Dropped    :"  << dropDsdvControlPacket << endl;
  cout << "  Clear to send         :"  << dropDsdvCTS << endl;
  cout << "  IP Route Request      :"  << dropDsdvRQ << endl;
  cout << "  IP Route Reply        :"  << dropDsdvRR << endl;
  cout << "  MAC Route Request     :"  << dropDsdvMAC_RR << endl;
  cout << "  MAC Route Reply       :"  << dropDsdvMAC_RR << endl;
  cout << "Total Data Received     :"  << totalDsdvPacketsReceived << endl;
  cout << "---------------------------------- " << endl;

  Print("DoStats",2);
}

uint32_t
CogMan::GetLengthOfInteger(uint32_t lengthOfThis)
{
  Print("GetLengthOfInteger",1);
  string tmpString;               // string for the number passed
  ostringstream convert;          // stream used for the conversion
  convert << lengthOfThis;        // get number
  tmpString = convert.str();      // convert to string
  return tmpString.length();
  Print("GetLengthOfInteger",2);
}

string
CogMan::SpacesAndTail(uint32_t noOfSpaces)
{
  Print("SpacesAndTail",1);
  string tailAndFence = "";
  for (uint32_t n=0; n<noOfSpaces; n++)
  {
    tailAndFence = tailAndFence + " ";
  }
  return tailAndFence + " |  ";
  Print("SpacesAndTail",2);
}

void
CogMan::SwitchProtocol()
{
  Print("SwitchProtocol",1);

  cout << GetTimeNow('s') << "s Switch protocol command issued:  ";
  if (currentProtocol == "aodv")
  {
    currentProtocol = "dsdv";
  }
  else
  {
    currentProtocol = "aodv";
  }
  cout << " switch to " << currentProtocol << endl;
  Print("SwitchProtocol",2);
}
```

```
// BELOW ME MIGHT BE USEFUL BUT ARE CURRENTLY NOT USED!

void
CogMan::SortVector()
{
  Print("SortVector",1);
  /* let's sort the vector by time (longest first)
  bool notSorted = true;

  cout << "\ninRangeUntil ";
  PrintVectorOfDistances (inRangeUntil);

  while (notSorted)
  {
    for (vectorIndex = inRangeUntil.begin(); vectorIndex<inRangeUntil.end()-1; vectorIndex++)
    {
      notSorted = false;
      if (vectorIndex->myTime < (vectorIndex+1)->myTime)
      {
        int    sn = vectorIndex->sourceNode;        // put first elements into temp variables
        int    dn = vectorIndex->neighbourNode;
        double ds = vectorIndex->distance;
        double mt = vectorIndex->myTime;

        vectorIndex->sourceNode     = (vectorIndex+1)->sourceNode;       // move bigger time
left 1
        vectorIndex->neighbourNode  = (vectorIndex+1)->neighbourNode;
        vectorIndex->distance       = (vectorIndex+1)->distance;
        vectorIndex->myTime         = (vectorIndex+1)->myTime;

        (vectorIndex+1)->sourceNode     = sn;       // move bigger time left 1
        (vectorIndex+1)->neighbourNode  = dn;
        (vectorIndex+1)->distance       = ds;
        (vectorIndex+1)->myTime         = mt;

        notSorted = true;
        break;
      }
    }
  }
  */
  Print("SortVector",2);
}

  //if (printRoutes)
  //  {
  //    Ptr<OutputStreamWrapper> routingStream = Create<OutputStreamWrapper> ("aodv.routes",
std::ios::out);
  //    aodv.PrintRoutingTableAllAt (Seconds (20), routingStream);
  //  }

  //   Ptr<OutputStreamWrapper> routingStream = Create<OutputStreamWrapper> (cout);
  //   aodv.PrintRoutingTableAllEvery (Seconds (2), routingStream);

  // attempt at getting from the routing table
  //     aodv::RoutingTable rtable (Seconds (2));

  // Trace routing tables
  //Ipv4GlobalRoutingHelper g;
  //  Ptr<OutputStreamWrapper> routingStream = Create<OutputStreamWrapper>
  //("dynamic-global-routing.routes", std::ios::out);
  // rtable.PrintRoutingTableAllAt (Seconds (12), routingStream);
```

Appendix 2 – Explaination of Methods

This chapter explains the methods that we used in programming our simulation models. Although we call on and make use of methods or libraries that are included in NS3 many of the methods required by our simulation model were not available and had to be created and programmed by us. The methods that we created are explained in detail – we do not explain in great detail the methods that we used that are part of the NS3 library because this documentation is available from the NS3 website.

We have categorised the methods and installed some order of when the methods are presented, however, the very nature of methods is that they are reusable and may be called at many locations in the simulation model. Once we have explained the purpose of a particular method and how the method operates we do not explain it again even though it may be called many times.

The full code is not listed in this chapter. There are roughly 10,000 lines of code after we have optimized it. Should the reader of this thesis be interested in the full text, i.e. variable declarations, variable assignments, dynamic data structures, conditional statements, instructions and so-forth the full code is listed in the appendix.

Thus the chapter is structured by following the flow of the algorithm, however we group related items i.e. mobility models.

Any C++ program starts by including the headers of the libraries that are used. We were able to make use of several libraries that are included in the NS3 simulator such as: aodv-module, dsdv-module, core-module, network-module, internet-module, mobility-module, point-to-point-module, wifi-module, v4ping-helper, iostream, math.h, cmath, udp-echo-helper, netanim-module, applications-module, aodv-rqueue, trace-helper, flow-monitor, flow-monitor-helper, flow-monitor-module, energy-module, string, vector, limits, random, ioctl, stdio, types, stat and unistd. The compiler and pre-processor includes the files into our program at compilation.

Once we have included the libraries that we want to make use of we then make available the namespaces that are used, we use the standard namespace that is part of the C++ programming language and the ns3 namespace that is part of the NS3 core environment. We then define a constant variable PI and set the value to 3.14159265359 which is used in our mobility models.

The first function called in any C or C++ program is **`main`** and is the entry point of the program once the required libraries have been included and the declaration of global variables has taken place. We have kept our main function minimalistic and use it as a springboard into the object oriented program. Thus our main function performs 4 tasks:

1. Configure the network
2. Set the required level of logging
3. Run the network scheduler
4. Return control to the command prompt (program terminates)

Our program takes an object oriented approach and creates an object that is used to control the data communication network that we are modelling. When creating an object the initial state of the object is defined by calling the constructor of the class name **`CogMan`** (p. 439).

The main function allows us to pass parameters from the command prompt, we use these values when we perform Configure the network task, which allows us to further define the initial state of the network and in some situations override the default state as defined by **`CogMan`** (p. 439). We perform task one by calling the **`Configure`** (p. 439) method. Once the configuration of the network has been completed we set which components are required to be logged. We set the logging by calling the **`SetLogging`** (p. 440) method. Once the logging components have been enabled or disabled we then call our scheduler to schedule the network events, this is called **`ScheduleEvents`** (p. 441). Once the network simulation has finished we then return control to the command prompt.

CogMan::CogMan()

The variables that we define were discussed briefly in the last chapter and include variables to define state of the network, keep count of packets, control the cognition cycle and others. We use approximately 60 scalar variables.

The constructor also creates our dynamic data structures to store information as the simulation runs and which are accessed during the simulation run to determine certain actions, such as controlling channel allocation to alleviate congestion. The data structures are described in the previous chapter.

void CogMan::Configure(argc, argv)

The values passed into this method allow us to control the initial state of the network. The parameters that can be controlled for the initial state are; Number of Nodes, Number of Traffic Flows, Number of Controllable Nodes, Default data rate, Cognition on/off, Simulation Time in Seconds, Transmission Radius, Log File, Mobility Model, Initial Starting Protocol, Speed of Controlled Nodes, Speed of Nodes, Interval to record positions, and the Folder for the trace files.

In regards to the last value, a folder is automatically created by this method based on the folder name that we pass from the command prompt if the folder does not already exist. This allows us to run several simulations and save the data to be analyzed to different folders of our choice without having to alter the code. It was imperative that we introduced a batch system to manage our runs as each simulation run required several hours to complete and was often left unattended. An example of a batch file is:

```
mkdir 4_20R
./waf --run "final_5
    --numOfNodes=20
    --numOfTrafficFlows=4
    --numOfCNodes=0
    --dataRate=11Mbps
    --cognition=false
    --simulationTime=600
    --radius=100
    --logFile=T1
    --mobilityModel=R
    --initialProtocol=aodv
    --cNodeSpeed=1.8
    --nodeSpeed=1.8
    --recordPositionInterval=5
    --folderName=4_20R
" > 4_20R/output.txt 2>&1
```

When any method or function ends the flow of control is returned to the place at which the method or function was invoked (called). Therefore when this function completes program control returns to the main function in order to execute the next instruction.

void CogMan::SetLogging()

The setLogging method allows us to turn on or off certain logging components such as when methods are entered or left, output variable data, packet drop information, packet sent information, packet received information, information relating to the route a packet has taken, distance between node information, device information, IP address information, traffic flow information, election process information, node destination information, node position information, energy source information, distance information from a given node to all other nodes, distance information for a given node for all other nodes in range of the source, routing information and node information. Implementing this method was imperative because of the mass amount of information generated by each of our simulation runs. For example with five of the logging components switched on and thirteen switched off one of the output files generated was in excess of 1.7Gig and consisted of forty six million, three hundred and seventy five thousand and twelve lines.

The final purpose of the setLogging function is to allow us to debug in steps.

Scheduling

Network Simulator 3 is an event driven discrete simulator which means each event in the network needs to be scheduled in a queue. This does not mean that the scheduler requires an entire list of events before the simulation begins. The NS3 scheduler is dynamic and can grow and shrink as events are added to the scheduler or removed from the scheduler. In addition to this which proved paramount when implementing cognitive attributes is that we can schedule or remove from the scheduling queue events dynamically during the simulation run. When running a simulation we have some idea of how the network will run (i.e. we instruct when a node should begin and stop transmission), however, with the introduction of managers and allowing managers the ability to request actions at their discretion based on their intuition and past knowledge there is no way to predict the events in the network. Thus the managers add events to the scheduling queue during the run that we did not foresee or could predict.

void CogMan::ScheduleEvents()

The `ScheduleEvents` method contains the code to schedule events within the simulator. In this method we mostly call on other methods to perform actions relating to the setup, configuration and operation of the network. For example we call methods to create nodes, assign mobility models, set the default packet size and data rate, create our traffic flows, etc.

This method also calls on the FlowMonitor class in order to keep track of all the packets and data flows within the network. This information is written to disk when the simulation ends.

We also make use of the Network Animator class that creates a file that can be run in the Network Animator visualization tool.

Towards the end of the **`ScheduleEvents`** method we schedule when the simulation should stop in seconds. Immediately after the scheduling we start the simulation. When the simulation finishes we write various data files to disk such as received packet information, dropped packet information and sent packet information. We also write our dynamic systemState data structure so that we can review when decisions were made by the management node within the network and analyze if the actions made any change in regards to network performance.

As mentioned our **`ScheduleEvents`** method calls a method to create our nodes. This method is called **`CreateAodvNodes`** (p. 443).

double CogMan::GetTimeNow(char *t*)

The GetTimeNow method will check the parameter passed into t and return the current simulation time in the required format. For example if the character ‘s’ was passed in to t, the time would be returned in seconds.

int CogMan::GetHoursFromSeconds(int *seconds*)

This method accepts an integer value that represents a number of seconds. The seconds are divided by 3600. The result of the division is returned to the invoking instruction as an integer which give us the number of hours passed.

int CogMan::GetMinutesFromSeconds(int *seconds*)

This method accepts an integer value that represents a number of seconds. The seconds are divided by 3600 and the result of the division is stored in an integer variable (thus losing precision, which is intended) called hours. We then subtract from the number of seconds the total number of hours multiplied by 3600. We then return seconds divided by 60 and return the resulting value to the invoking instruction.

int CogMan::GetSecondsFromSeconds(int *seconds*)

This method accepts an integer value that represents a number of seconds. The seconds are divided by 3600 and the result of the division is stored in an integer variable (thus losing precision, which was intended) called hours. We then subtract from the number of seconds the total number of hours multiplied by 3600. We then divide seconds by 60 and store the result in an integer variable called minutes. We multiply the number of minutes by 60 and subtract this from seconds. We then return seconds.

Node Configuration

In this section we discuss how to create a node(s) and install various devices onto a node so that it can be placed within the network and participate by sending and receiving radio waves. Our network simulator uses many types of nodes which include Nodes that communicate using the AODV routing protocol, nodes that communication using the DSDV routing protocol, nodes that can be controlled by a managing node and nodes that have been place in order to create congestion. As well as different type of nodes we can configure them differently, i.e. we can alter the transmission radius.

void CogMan::CreateAodvNodes(int *n*)

The `CreateAodvNodes` method allows us to pass a parameter that specifies how many nodes we would like to create. The method then creates the required number of nodes and places pointers to the nodes in a container called aodvNodes. Once the nodes are created we then call on methods to create the network devices (p. 444), the stack (p. 444), assign IP addresses (p. 444), create an interface to the network medium (p. 445), assign initial placements of each node (p. 451), assign a mobility model (p. 454) and install an energy source that represents a lithium battery (p. 529).

void CogMan::CreateAodvDevices()

The device that we install on each node is based on that of an isotropic antenna. We use NS3 classes to implement and configure the antenna. We set various parameters that alter the operation of the antenna, for example we set values for the energy detection threshold, the transmission and receiving gain, the TX power levels and the noise model that we are using.

We then concentrate on the MAC layer and set the mode of operation to adhoc mode.

What remains is to select a frequency for the antenna to transmit and/or receive electromagnetic radio waves. This may change during the simulation but the initial value is 2412Hz which is channel 1 of the 2.4Ghz range using 802.11b wireless transmission protocol. The physical and mac layer configuration are aggregated onto the nodes in the aodvNodes container and pointers to the aggregated objects are stored in the aodvDevices container.

Next we call the method **`CreateAodvStack`** (p. 444) to create a stack for the devices.

void CogMan::CreateAodvStack()

The CreateAodvStack method creates a new stack called aodvStack and sets the routing protocol to AODV for this stack. We then install the stack onto the nodes in the aodvNodes node container.

Once the stack exists we can assign IP addresses to each node. We do this by calling the **`CreateAodvAddress`** (p. 444) method.

void CogMan::CreateAodvAddresses()

The CreateAodvAddresses method allows us to create a base IP address (from address or start address), and create an IP subnet. This configuration is stored in the aodvAddresses container. In our case the aodv nodes operate on network 10.2.0.0. Therefore our IP addresses for the aodv nodes start at 10.2.0.1. The subnet mask is set to 255.255.0.0 this allows us to create more than 256 IP addresses for this group of nodes should we require. Once the address configuration is set we then assign the addresses on to the nodes by calling the `CreateAodvInterface` method (p. 445).

void CogMan::CreateAodvInterface()

The CreateAodvInterface will aggregate the IP addresses to the interfaces associated with the aodv devices in sequential order.

void CogMan::CreateDsdvNodes(int *n*)

The `CreateDsdvNodes` method allows us to pass a parameter that specifies how many nodes we would like to create. The method then creates the required number of nodes and places pointers to the nodes in a container called dsdvNodes. Once the nodes are created we then call on methods to create the network devices (p.445), the stack (p. 449), assign IP addresses (p. 446), create an interface to the network medium (p. 449), assign initial placements of each node (p. 453), assign a mobility model (p. 454) and install an energy source that represents a lithium battery (p. 529).

void CogMan::CreateDsdvDevices()

The device that we install on each node is based on that of an isotropic antenna. We use NS3 classes to implement and configure the antenna. We set various parameters that alter the operation of the antenna, for example we set values for the energy detection threshold, the transmission and receiving gain, the TX power levels and the noise model that we are using.

We then concentrate on the MAC layer and set the mode of operation to adhoc mode.

What remains is to select a frequency for the antenna to transmit and/or receive electromagnetic radio waves. This may change during the simulation but the initial value is

2427Hz which is channel 4 of the 2.4Ghz range using 802.11b wireless transmission protocol. The physical and mac layer configuration are aggregated onto the nodes in the dsdvNodes container and pointers to the aggregated objects are stored in the dsdvDevices container.

Next we call the method **`CreateDsdvStack`** (p.449) to create a stack for the devices.

void CogMan::CreateDsdvStack()

The **`CreateDsdvStack`** method creates a new stack called dsdvStack and sets the routing protocol to DSDV for this stack. We then install the stack onto the nodes in the dsdvNodes node container.

Once the stack exists we can assign IP addresses to each node. We do this by calling the **`CreateDsdvAddress`** (p. 446) method.

void CogMan::CreateDsdvAddresses()

The CreateDsdvAddresses method allows us to create a base IP address (from address or start address), and create an IP subnet. This configuration is stored in the dsdvAddresses container. In our case the dsdv nodes operate on network 10.1.0.0. Therefore our IP addresses for the aodv nodes start at 10.1.0.1. The subnet mask is set to 255.255.0.0 this allows us to create more than 256 IP addresses for this group of nodes should we require. Once the address configuration is set we then assign the addresses on to the nodes by calling the **`CreateDsdvInterface`** method (p. 449).

void CogMan::CreateDsdvInterface()

The CreateDsdvInterface will aggregates the IP addresses to the interfaces associated with the dsdv devices in sequential order.

void CogMan::CreateControlledAodvNodes(int *n*)

The **`CreateControlledAodvNodes`** method allows us to pass a parameter that specifies how many nodes we would like to create. The method then creates the required number of nodes and places pointers to the nodes in a container called aodvControlledNodes. Once the nodes are created we then call on methods to create the network devices (p. 447), assign IP addresses (p. 447), create an interface to the network medium (p. 447), assign initial placements of each node (p. 452), assign a mobility model (p. 454) and install an energy source that represents a lithium battery (p. 529).

void CogMan::CreateControlledAodvDevices()

The device that we install on each node is based on that of an isotropic antenna. We use NS3 classes to implement and configure the antenna. We set various parameters that alter the operation of the antenna, for example we set values for the energy detection threshold, the transmission and receiving gain, the TX power levels and the noise model that we are using.

We then concentrate on the MAC layer and set the mode of operation to adhoc mode.

What remains is to select a frequency for the antenna to transmit and/or receive electromagnetic radio waves. This may change during the simulation but the initial value is 2412Hz which is channel 1 of the 2.4Ghz range using 802.11b wireless transmission protocol. The physical and mac layer configuration are aggregated onto the nodes in the aodvNodes container and pointers to the aggregated objects are stored in the aodvDevices container.

void CogMan::CreateControlledAodvStack()

The `CreateControlledAodvStack` method creates a new stack called aodvControlledStack and sets the routing protocol to AODV for this stack. We then install the stack onto the nodes in the aodvControlledNodes node container.

void CogMan::CreateControlledAodvInterface()

The `CreateControlledAodvInterface` will aggregate the IP addresses to the interfaces associated with the AODV devices in sequential order.

void CogMan::CreateControlledDsdvNodes(int *n*)

The `CreateControlledDsdvNodes` method allows us to pass a parameter that specifies how many nodes we would like to create. The method then creates the required number of nodes and places pointers to the nodes in a container called dsdvControlledNodes. Once the nodes are created we then call on methods to create the network devices (p. 448), the stack (p. 449), create an interface to the network medium (p. 449), assign initial placements of each node (p. 452), assign a mobility model (p. 454) and install an energy source that represents a lithium battery (p. 529).

void CogMan::CreateControlledDsdvDevices()

The device that we install on each node is based on that of an isotropic antenna. We use NS3 classes to implement and configure the antenna. We set various parameters that alter the operation of the antenna, for example we set values for the energy detection threshold, the transmission and receiving gain, the TX power levels and the noise model that we are using.

We then concentrate on the MAC layer and set the mode of operation to adhoc mode.

What remains is to select a frequency for the antenna to transmit and/or receive electromagnetic radio waves. This may change during the simulation but the initial value is 2427Hz which is channel 4 of the 2.4Ghz range using 802.11b wireless transmission protocol. The physical and mac layer configuration are aggregated onto the nodes in the dsdvControlledNodes container and pointers to the aggregated objects are stored in the dsdvControlledDevices container.

void CogMan::CreateControlledDsdvStack()

The `CreateControlledDsdvStack` method creates a new stack called dsdvControlledStack and sets the routing protocol to DSDV for this stack. We then install the stack onto the nodes in the dsdvControlledNodes node container.

void CogMan::CreateControlledDsdvInterface()

The `CreateControlledDsdvInterface` method will aggregates the IP addresses to the interfaces associated with the dsdv devices in sequential order.

void CogMan::CreateCongestionNodes(int *n*)

void CogMan::CreateCongestionDevices()

void CogMan::CreateCongestionStack()

void CogMan::CreateCongestionAddresses()

The four above methods (CreateCongestionNodes, CreateCongestionDevices, CreateCongestionStack and CreateCongestionAddresses) work the same as the methods previous defined. Rather than repeat the information already given we summarise the purpose of the congestion nodes. `CreateCongestionNodes` method creates nodes that are intent on transmitting traffic to interfere with our source node(s) traffic. We set a default packet size of 1000 bytes and a physical mode using DSSS at a rate of 11Mbps. We also disable fragmentation for frames that are below 2200 bytes, turn off request/clear to send for frames below 2200 bytes and set the non-unicast rate to that of the unicast rate. Our reception gain is set to 0 to prevent antenna gain being added.

The energy detection threshold is set to zero so that sending devices fail to detect a signal which will prevent them backing off. The transmission gain is set to a value that is higher than that of our source node(s) so that the strength of the signal generated is higher than that of our source node(s).

In order to avoid complete packet loss we control the rate at which the congestion node(s) generate traffic by setting an interPacketInterval variable. We are also able to specify when to start transmitting the packets, when to stop transmitting the packets and the number of packets to transmit.

We create areas of congestion by placing the nodes within the simulated geographical area at the desired locations. Our congestions nodes have a restricted transmission range of up to 100 meters which is consistent with our other nodes.

This completes the node configurations; next we discuss how our nodes are grouped.

Node Grouping

Most of our nodes are grouped by creating containers and this has been demonstrated earlier when we discussed creating node containers for our nodes that operate using the DSDV routing protocol, nodes that use the AODV routing protocol and our nodes that cause congestion.

We also create a global node container called allNodes that contains all nodes used in our simulation. The nodes are not moved from their original containers and the nodes can still be accessed using the original container descriptor. When we create a new node container we copy the pointer reference from the original container, this is useful when we want to perform an action on all nodes and saves us accessing each individual container separately.

There are other grouping strategies that are used which are mostly incorporated into our placement strategies and are discussed subsequently; first however, we discuss how we split nodes into two even groups.

void CogMan::SplitNodesTwoEvenGroups()

The SplitNodesTwoEvenGroups will split the total number of nodes into two even groups. We keep track of the groups by assigning the nodes to either a node container called eagles or a node container called sharks. We do this for each and every node in the aodvNodes and the dsdvNodes containers.

Now we discuss how we place the nodes into the simulation world. There are various placement strategies used in our simulations. We discuss the placement strategies before continuing with our program flow.

Placement Strategies

We have created various placement strategies that control the initial locations of our nodes. For example we could place all the nodes at the same location, we could place nodes within a defined area at random locations, place the nodes into two separate groups or in a grid like formation. We have total control over where and how we place our nodes in the simulated world. This section explains the placement strategies that we programmed and explains how we used them.

void CogMan::PlaceNodesTwoGroups()

Some of our simulations require that we split the nodes into two groups. One of our earlier simulations demonstrates this when we assign two groups of soldiers engaging in a search operation and navigating around lakes.

The `PlaceNodesTwoGroups` method creates a list of allocated positions and is referenced by a pointer. Each node in turn is then added to the list with positions that identify their x, y and z co-ordinates. We assign some element of randomness by using the rand() function that is included in the math library. This enables us to prevent all nodes being positioned in exactly the same position. The parameters when calling the random function are such that they only allow a random number within a given limited range, in this case between -40 and 20. Thus this allocates nodes to positions that are not the same but in the same approximate location.

Our groups are by default a third of nodes in one group and two thirds in the other group; however this is easily tailored to allow any fractional split of the nodes.

We identify each group of nodes by storing the nodes numbers for each group in a container, the containers are called Eagles and Sharks – for no reason other than that is what came to mind at time of deciding what to call our groups.

void CogMan::PlaceNodesRandomLocation(NodeContainer *nodes*)

The PlaceNodesRandomLocation method creates a list of allocated positions and is referenced by a pointer. Each node in turn is then added to the list with positions that identify their x, y and z co-ordinates. We assign some element of randomness by using the rand() function that is included in the math library. The parameter passed is 0.00000007-4 which gives us a value in the range of 0-150.

Void CogMan::PlaceNodesOneLocation()

The PlaceNodesOneLocation method creates a list of allocated positions and is referenced by a pointer. Each node in turn is then added to the list with positions that identify their x, y and z co-ordinates. When this method is called each and every node created in the simulation is assigned to the same location and occupy the same space – we do this by making use of the global node container called allNodes that was discussed earlier.

void CogMan::PlaceNodesOneLocation(NodeContainer *nodes*)

We have overloaded the PlaceNodesOneLocation method to allow for a node container to be passed. The method is similar to `PlaceNodesOneLocation` (p. 452). The difference being that all nodes in a particular container are placed in the same location. This allows us to place aodv nodes at a given location and all dsdv nodes at another location.

void CogMan::PlaceNodesTopLCRMiddleLCRBottomLCM()

The **PlaceNodesTopLCRMiddleLCRBottomLCM** method creates a list of allocated positions and is referenced by a pointer. Each node in turn is then added to the list with positions that identify their x, y and z co-ordinates.

This will place a total of 9 nodes (the first nodes in the container) at predefined positions which are top left, top middle, top right, middle left, middle, middle right, bottom left, bottom middle, bottom right. The placement pattern looks like:

x x x

x x x

x x x

void CogMan::DsdvPositionMatchAodv()

The **DsdvPositionMatchAodv** method is used to assign positions to the DSDV nodes that are derived from the current position of its corresponding AODV node. The method will loop though the container that stores the pointers for the AODV nodes, for each AODV node we get the position and then move the corresponding DSDV node to that position.

void CogMan::AodvPositionMatchDsdv()

The AodvPositionMatchDsdv method operates similarly to **DsdvPositionMatchAodv** (p. 453) with the exception that the AODV nodes will match the positions of their DSDV counterpart node.

Once the nodes have been placed into the world we then assign the nodes with a mobility model. The mobility models are discussed next.

Mobility Models

In order for Network Animator to function correctly there needs to be a mobility model assigned to each and every node in the simulation. Therefore, even nodes that do not move will require a mobility model attached to them – in order to achieve this we use the Constant Mobility model.

void CogMan::InstallConstantMobility()

This method will assign a constant mobility model to all nodes in the network simulation. We have overloaded this function so that we can pass a collection of nodes to apply the mobility model too, thus not all nodes in the network. The overloaded function accepts a node container in its parameter list.

void CogMan::MoveNodeTo (int *idOfNode*, double *x*, double *y*)

The **`MoveNodeTo`** method is never called directly but by other mobility methods and is used to update a given nodes x and y coordinates to the values passed into the parameter list. A mobility model object is created and the mobility model pointer of the node that is passed into the functions parameter list is obtained.

Once we have the pointer to the current nodes mobility model we can then use a method to get the current position of the node. This returns a vector that stores x, y and z co-ordinates. We update these co-ordinates to the values passed in the parameter list. We then set the position of the node by invoking the SetPosition method.

void CogMan::MoveControlledNodeTo(int *nodeId*, double *x*, double *y*)

The **`MoveControlledNodeTo`** method is never called directly but by other mobility methods and is used to update a given controlled nodes x and y coordinates to the values passed into the parameter list. A mobility model object is created and the mobility model pointer of the node that is passed into the functions parameter list is obtained.

Once we have the pointer to the current nodes mobility model we can then use a method to get the current position of the node. This returns a vector that stores x, y and z co-ordinates. We update these co-ordinates to the values passed in the parameter list. We then set the position of the node by invoking the SetPosition method.

void CogMan::ArchimedeanSpiral(int *nodeId*)
This method takes a node id and moves the node in an anticlockwise spiral and uses the following motion equations for the nodes

$$x(t_{i+1}) = x(t_i) + \frac{t_{i+1}}{2\pi} cos(t_{i+1}); \qquad y(t_{i+1}) = y(t_i) + \frac{t_{i+1}}{2\pi} sin(t_{i+1}),$$

here $t \in [0, \mathrm{simulationTime}]$ is time, step size is 0.05 seconds, i.e. the time mesh is

$$\{t_0 = 0, t_1 = 0.05, t_2 = 0.1, \ldots, t_{\mathrm{last}} = \mathrm{simulationTime}\}.$$

void CogMan::ArchimedeanSpiralClockwise(int *nodeId*)
This method takes a node id and moves the node in an anticlockwise spiral and uses the following motion equations

$$\mathrm{x}(\mathrm{t}_{i+1}) = \mathrm{x}(\mathrm{t}_i) + \frac{\mathrm{t}_{i+1}}{2\pi} \cos(-\mathrm{t}_{i+1}); \qquad \mathrm{y}(\mathrm{t}_{i+1}) = \mathrm{y}(\mathrm{t}_i) + \frac{\mathrm{t}_{i+1}}{2\pi} \sin(-\mathrm{t}_{i+1}),$$

where $t \in [0, \mathrm{simulationTime}]$ is time, step size is 0.05 seconds, i.e. the time mesh is

$$\{t_0 = 0, t_1 = 0.05, t_2 = 0.1, \ldots, t_{\mathrm{last}} = \mathrm{simulationTime}\}.$$

void CogMan::NodeWalkTo (int *nodeId*, *double xGoal*, *double yGoal*, double *stepSize*, double *speed*)
This method prepares a node to move to a given location. At the same time its corresponding node in the other container will move in an identical fashion. For example if node 1 of the AODV container moves, then node 1 of the DSDV container will assume the same mobility.

The method begins by checking that the node requested to move is valid, i.e. it exists. If the node does not exist the program stops with an assert message.

Next we get the current simulation time in seconds.

We then check to see if the node is moving or not. A node should not be able to move to two locations at the same time as this defies the laws of physics – albeit NS3 allows this. Therefore if the node is already moving we reschedule the walk. If the node is not moving we create a vector to store the current position of the node ID that was passed by calling the `GetNodePosition` (p. 464) method. We then create another vector called goalPosition to store coordinates of the goal (x_g, y_g) as x and y values passed to the methods parameter list.

We then called the `GetDistance` (p. 465) method and pass the current coordinates (x_c, y_c) of the node the current position vector and the goalPosition vector. The returned value is stored in a local variable called distance. We calculate the duration of the walk by dividing the distance by the speed of the object that is walking – we add very small variations to each node so that each node is not moving at exactly the same speed as this is unrealistic. We calculate the number of steps N_{steps} that the object will have to take in order to get to its destination by dividing the step size (stride size) by the distance.

Step size (stride size) allows us to account for different objects (i.e. soldiers or tanks). It also allows us to apply realistic transition from the original position to the destination position and better models how an object moves.

We then calculate how long each step will take t_{step} so that we know the position of the node at each step taken and the time that the node will be in that position. This is calculated by dividing the number of steps by the duration of the walk.

We calculate each move for x and y by the following formula

$$x = \frac{x_g - x_c}{N_{steps}}, y = \frac{y_g - y_c}{N_{steps}}.$$

A method called **`RecordMovement`** (p. 466) is invoked that records the node ID and when the journey finishes. This allows us to prevent the same node moving again whilst already in motion.

Now we have the basis for moving the node to a given location in steps. We call the method (p. 454) N_{steps} amount of times, where N_{steps} is the number of steps. The call to the method **`NodeMoveTo`** (p. 454) is based by creating a scheduled event. The time is calculated by adding $N_{steps} \quad \times t_{step}$ to the original time of the original position.

void CogMan::ControlledNodeWalkTo(int *sNode*, int *mNode*, double *atTime*)

This **`ControlledNodeWalkTo`** method prepares a controlled node to move to a given location. At the same time its corresponding controlled node in the other container will move in an identical fashion. For example if node 1 of the AODV controlled nodes container moves, then node 1 of the DSDV controlled nodes container will assume the same mobility.

The method begins by checking that the node requested to move is valid, i.e. it exists. If the node does not exist the program stops with an assert message.

Next we get the current simulation time in seconds by calling the **`GetTimeNow`** (p. 442) method.

We then check to see if the node is moving or not by calling the **`IsControlledNodeMoving`** (p. 463) method. A node should not be able to move to two locations at the same time as this defies the laws of physics – albeit NS3 allows this. Therefore if the node is already moving we reschedule the walk. If the node is not moving we create a vector to store the current position of the node ID that was passed by calling the **`GetControlledNodePosition`** (p. 527) method. We then create another vector called goalPosition to store the x and y values passed to the methods parameter list.

We then called the **`GetDistance`** (p. 465) method and pass the current position vector and the goalPosition vector. The returned value is stored in a local variable called distance. We calculate the duration of the walk by dividing the distance by the speed of the object that is walking – we add very small variations to each node so that each node is not moving at exactly the same speed as this is unrealistic. We calculate the number of steps N_{steps} that the object will have to take in order to get to its destination by dividing the step size (stride size) by the distance.

Step size (stride size) allows us to account for different objects (i.e. soldiers or tanks). It also allows us to apply realistic transition from the original position to the destination position and better models how an object moves.

We then calculate how long each step will take so that we know the position of the node at each step taken and the time that the node will be in that position. This is calculated by dividing the number of steps by the duration of the walk.

We calculate each move for x and y by the following formula

$$x = \frac{x_g - x_c}{N_{steps}}, y = \frac{y_g - y_c}{N_{steps}}.$$

A method called **`RecordControlledNodeMovement`** (p. 473) is invoked that records the node ID and when the journey finishes. This allows us to prevent the same node moving again whilst already in motion.

Now we have the basis for moving the node to a given location in steps. We call the method **`MoveControlledNodeTo`** (p. 454) n amount of times where n is the number of steps. The call to the method **`MoveControlledNodeTo`** (p. 454) is based by creating a scheduled event. The time is calculated by adding $N_{steps} \times t_{step}$ to the original time of the original position.

void CogMan::MobilityRandomWaypoint()

The MobilityRandomWaypoint mobility model allows us to install a random waypoint mobility model onto our nodes. We start by creating an object in the object factory (a class that is part of NS3). Once the object is created we assign minimum and maximum values for X and Y attributes that restrict the movements of the nodes. For example, if we assigned a minimum value of 0.0 and a maximum value of 250.0 for X and a minimum value of 0.0 and a maximum value of 250.0 for Y the node would only be able to move within a 250meter squared box.

We then assign a constant node speed value. Next a list of allocated positions is created and is referenced by a pointer. Each node in turn is then added to the list with positions that identify their x, y and z co-ordinates.

Next the mobility model is installed onto each and every node inside a given node container. In our case we assign the random waypoint mobility model to all nodes in the congestionNodes container.

The final mobility model that we discuss is the **`MoveNodeDiamondPattern`** (p. 460) mobility model.

void CogMan::MoveNodeDiamondPattern()

The MoveNodeDiamondPattern mobility model is intended to be used with groups of nodes that are defined in the placement strategy **`PlaceNodesTwoGroups`** (p. 451) – eagles and Sharks. Our method starts by looping through every node in the Eagles container and defines a movement that will move the node(s) from their current position to a new positions that is to the left and below their current position. Next we loop through the node(s) in the sharks container and define a movement model that is similar to that of the eagles container except they move to the right and down.

When the nodes reach their destination they then move back towards each other. This can be visualized as a diamond shape. This process is repeated and can be thought of as nodes zigzagging in and out. Figure 29 - Diamond Mobility Model in chapter 4 illustrates this mobility pattern.

The actual movement is performed by the **`NodeWalkTo`** (p. 455) method. The MoveNodeDiamondPattern configures the mobility and then passes this information to the NodeWalkTo method.

So far we have discussed the initialization, node configuration, placement strategies and mobility models that we use in our simulations runs. We mentioned that a node should not move if it is already moving as this would create discrete jumps. Due to our nodes not having the ability to teleport from one location to another we prevent nodes from moving if they are already moving. To achieve this we first get the mobility state of the node by calling the **`IsMoving`** (p. 462) method.

void CogMan::LakeScenario()

The `LakeScenario` mobility model is intended to be used with groups of nodes that are defined in the placement strategy `PlaceNodesTwoGroups` (p. 451) – Eagles and Sharks.

For the first group of nodes (Eagles) we move each node from their current location to the location (x=0, y=25) in meters. It would be unnatural for all the nodes to move at the same speed so we create random variations for each node but they all move approximately 1.8 meters per second with a minimum speed of 1.4 meters per second and a maximum speed of 2.5 meters per second. We apply a similar variation to the x and y coordinates because we do not want each node to occupy the same space. Once we have the speed and the destination for the node we calculate the time that this leg of the journey will take by calling the `GetTimeOfJourney` (p. 463) method. We use the value returned from the `GetTimeOfJourney` (p. 463) method as the value for when the next leg of the journey should start. Each leg of the journey is scheduled by calling the `NodeWalkTo` (p. 455) method.

The next leg of the journey sends the node to (x=100, y=25), we generate new random variations for the speed and destination (thus the node could increase or decrease speed slightly for this leg).

We repeat the process for each and every node in the eagles group until simulation time has ended. Each leg of the journey we set x and y values based on their previous values. For each leg we add 25 to y so each node will continually head south. For x we check to see if the previous value of x was less than 20. If this was the case we set x to 100, if x was not less than 20 we set x to 0. This gives us a zigzagging motion. For example, the nodes will head South East, then South West and repeat.

We apply the same strategy for the nodes in the sharks group. However, we send them in the opposite direction. Thus they will head South West, then South East and repeat.

This mobility model simulates nodes getting to a start position and then moving away from each other for a given time, before moving toward each other for a given time. Thus creating links and then breaking links.

Mobility Model Support Methods

This section details the methods that were required to support the mobility of our nodes. They are mainly used by the mobility models above, by the management nodes or by the network command centre.

bool CogMan::IsNodeMoving(int *nodeId*, int *x*, int *y*, double *stride*, double *speed*)

The **`IsNodeMoving`** method is used to check to see if a node is moving or not. The method accepts a node identifier so that we know which node to check for movement. If the node is not moving a value of false is returned to the method that invoked this method, in our case the **`NodeWalkTo`** (p. 455) method. If the node is moving we obtain from a dynamic data structure when the node will complete its current journey. We then reschedule the movement to begin at a new time using this value. We know when a node will complete its journey by calling our **`GetTimeOfJourney`** (p. 463) method.

bool CogMan::IsControlledNodeMoving(int *nodeId*, int *x*, int *y*, double *stride*, double *speed*)

The **`IsControlledNodeMoving`** method works similary to the **`IsNodeMoving`** (462) method with the exception that it checks if a controlled node is moving and if so reschedules the move when the current journey completes.

double CogMan:: IsControlledNodeMoving(int *nodeId*)

This `IsControlledNodeMoving` method gets the current simulation time by calling the `GetTimeNow` (p. 442) method and stores the value in a variable called timeNow. Then we declare a variable called inXSec and initialize this to 0.

We loop through a dynamic data structure called setControlledNodeMove until we find the node of interest which is based on the node identifier that has been passed. Once the node is found we update inXSec to a value that reflects when the node will have completed its journey with an additional second added as a safety margin and then subtract the current time. This gives us how many seconds the node will continue to move for. This value is returned to the invoking function. This method would have been clearer if it were called GetControlledNodeJourneyTime.

If the node is not located once the entire data structure has been searched the value 0 is returned to the invoking function.

double CogMan::GetTimeOfJourney (int *nodeId*, double *xGoal*, double *yGoal*, double *speed*)

In order to get the duration that the journey will take we make use of the nodes current position and its intended position. The method begins by invoking a method called `GetTimeNow` (p. 442) to get the current time in seconds.

In order to work how out how long a journey will take we need to know the current position of the node, we get this information by calling the method `GetNodePosition` (p. 464).

Next we call a method called `GetDistance` (p. 465) and pass a vector that contains the current node position and a vector that contains the node journey end position. We divide the return value from the `GetDistance` (p. 465) method by the speed of which the node is moving. We store the calculated value in a local variable called duration (the duration of the journey).

The duration value is returned to the method that invoked this one.

double CogMan::GetTimeOfJourney (double *x*, double *y*, double *xGoal*, double *yGoal*, double *speed*)

The GetTimeOfJourney is an overloaded method and gives the same return value as the one discussed on page 463. The only difference being that rather than passing a node identifier and the getting the x and y coordinates based on the identifier we pass the x and y coordinates directly.

double CogMan::GetTimeOfJourneyControlledNode(Vector *currentPosition*, double *xGoal*, double *yGoal*, double *speed*)

The `GetTimeOfJourneyControlledNode` method returns how long a journey will take given the current position, the goal position and the speed at which the node moves. We get the distance by invoking the `GetDistance` (p. 465) method and pass two vectors (current position and target position). The value returned is divided by the specified speed and stored in a temporary variable called duration. The value contained in the duration variable is returned to the invoking function.

ns3::Vector CogMan::GetNodePosition(uint32_t *nodeId*)

The current node position is obtained by invoking the method `GetNodePosition` and passing a node identifier in to the parameter list for the node that we want the position of. The `GetNodePosition` method uses the node identifier to obtain a pointer that is assigned to the mobility model for that node.

Once we have the correct pointer to the mobility model for the given node we then invoke a function called GetPosition that is part of the mobility model class. This method returns a

NS3 vector that has X, Y and Z attributes defining the current position for the node identifier that was passed to our **GetNodePosition** method.

ns3::Vector CogMan::GetControlledNodePosition(uint32_t *nodeId*)

The **GetControlledNodePosition** works in the same way as **GetNodePosition** (p. 464) but returns the position of a controlled node.

ns3::Vector CogMan::GetNodePosition(uint32_t *nodeId*, NodeContainer *nodes*)

The **GetNodePosition** works in the same way as **GetNodePosition** (p. 464) but returns the position of a controlled node for a given node container.

double CogMan::GetDistance(ns3::Vector *position1*, ns3::Vector *position2*)

The **GetDistance** method will get the distance between two points. We use the Pythagorean Theorem.

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The method returns the distance between the two points as a double value.

void CogMan::GetAllDistancesFromSource(int *sourceNodeId*)

The **GetAllDistancesFromSource** method populates a vector called nodesInMyWorld that contain the distance from the source node to each and every other node in the simulation. First we get the position of the source node by invoking the **GetNodePosition** (p. 464) method. Next we get the node position a node that is not this source node by calling the **GetNodePosition** (p. 464) method. Then we get the distance between the source node and the node that we just go the position for by invoking the **GetDistance** (p. 465) method. Finally we store to a vector nodesInMyWorld the source node identifier, the destination node identifier, the distance between the two nodes and the current simulation time if the two nodes are in range. We check the range by calling the **InRange** (p. 468) method.

void CogMan::CreateVectorOfDistancesAllNodes()

The **CreateVectorOfDistancesAllNodes** method works very similary to the **GetAllDistancesFromSource** (p. 465) method. With the exception that it does not accept a source node identifier. Instead this method populates a data structure called allNodesTheWorld. For each and every node we record every other node identifier, the distance to that node and the current simulation time.

void CogMan::RecordMovement(int *nodeId*, double *endTime*)

The **RecordMovement** method records to a dynamic data structure called setNodeMove the node identifier, the current time and the time that the journey concludes. This data structure is used each time a node is assigned mobility. We get the current simulation time by calling the **GetTimeNow** (p. 442) method.

void CogMan::CourseChange(Ptr<const MobilityModel> *model*)

The **CourseChange** method is a callback method that is called when a course change has occurred with a node using the AODV routing protocol. This method is only necessary when using the random waypoint mobility model as the path a node takes is not predetermined. The method ensures that the paired DSDV nodes move exactly as their AODV counterparts.

When the method is called a pointer to a mobility model for the node that moved or changed course is passed to this method. Using this pointer we get the position of the node that moved by calling the **GetNodePosition** (p. 464) method. We use the position returned to set the position of the corresponding DSDV node by calling the **MoveNodeTo** (p. 454) method.

double CogMan::GetDensityFromNode(int *nodeId*)

The **GetDensityFromNode** method returns a density reference for a given node identifier. We could not simply return the number of nodes within the range of the node because different nodes may be able to transmit at different distances. Consider for example a node that transmits at a radius of 100m and there are five nodes within the transmission range. Now consider another source node that is able to transmit at a radius of 200m and there are five nodes within transmission range. Returning the value five for both cases does not give the same density. The node that transmits at 100m with five nodes in its transmission range has a much higher density that that of a node that transmits at 200m with the same number of nodes within transmission range. Therefore to calculate the density reference we use the formula:

$$\mathrm{d} = \frac{\mathrm{n}}{\pi\, \mathrm{r}^2}$$

Where d is the density reference, n is the total number of nodes within transmission range of the given node and r is the transmission radius.

Before we obtain the number of nodes n in transmission range we first establish which routing protocol is currently being used. If the routing protocol is AODV we create a vector index and set this to the start address of the aodvNodes container. If the routing protocol is DSDV we create a vector index and set this to the start address of the dsdvNodes container.

We then loop through all the nodes in the container and get the distance from the source node to the current node that is being checked in the container (with the exception of the source node) by calling the **`GetDistance`** (p. 465) method. The **`InRange`** (p. 468) method is called to check if the distance is within the current transmission radius (in range of the source node) we add one to the variable totalInRange.

The transmission radius r of the node is defined when the class constructor is called. The default value that we set the radius to is 100 meters. Although this may change depending on the type of node object.

Constant π is called PI and is set to the value 3.14159265359.

Once we have a density reference we perform an election by calling the `Election` (p. 492) method to determine which node will become a management node and have the cognitive attributes that allows the node to observe the network and make decisions.

bool CogMan::InRange(int *distance*)

The `InRange` method accepts a parameter that specifies the distance between two nodes. We compare the evaluated distance with the radius and return a Boolean variable with the value true or false. True if the distance is within the transmitting range.

$$\mathrm{inRange} = \begin{cases} true, & if \quad distance < radius \\ false, & else \end{cases}$$

double CogMan::GetPredictedMaximalTimeInRange(int *sourceNodeId*, int *destinationNodeId*)

The `GetPredictiedMaximalTimeInRange` method accepts two integer values into the parameter list and stores these values in local integer variables sourceNodeId and destinationNodeId respectively.

The function creates a vector called sNodePredictedPosition that is to be used to store the predicted position of a given node (in this case the source node that is identified by the value that was passed to sourceNodeId variable). We repeat this process for the destination node and call the vector dNodePredictedPosition.

In order to predict a future node position we first get the current time and store this in a value called futureTime.

Next we enter a loop and add one second to the value that currently resides in futureTime. We set the value of the sNodePredictedPosition vector to a value returned by the **`PredictPosition`** (p. 469) method (we pass the future time value and the source node ID). We do the same for the destination node and pass the same future time value and the destination node ID, the return value is stored in dNodePredictedPosition. Our loop condition is then evaluated because we use a do while loop rather than the more commonly used while loop, this allows us to evaluate the condition at the end of the loop rather than before the loop.

The while conditional statement is evaluated by calling the method **`GetDistance`** (p. 465) and passing the vectors sNodePredictedPosition and dNodePredictedPosition. The value returned from **`GetDistance`** (p. 465) is then used to evaluate if the nodes are in range or not. We achieve this by using the value derived from **`GetDistance`** (p. 465) as the parameter for the **`InRange`** (p. 468) method. The InRange method will return the value true if the nodes are still in range or false otherwise. If the nodes are in range the process is repeated for a further one second in to the future.

ns3::Vector CogMan::PredictPosition(int *nodeId*, double *futureTime*)

The PredictPosition method accepts a node ID and a future time in order to attempt to predict that nodes position at the future time passed. We assume that the node is moving at a constant velocity.

The method begins by creating three vectors, currentNodePosition, previousPosition and predictedPosition. We also create four local variables to store a previous time value, a velocity predicted X co-ordinate, a velocity Y co-ordinate and the predicted time.

We populate the vector currentNodePosition to the nodes' current position by calling the method GetNodePosition and passing the method the ID of the node of interest.

Next we create an index to a dynamic data structure called myPositions, unlike normal circumstances where one would point to the start of the data structure we position our pointer so that it points to the end of the data structure. We do this so that we can get the last known position of the node as this helps improves the likelihood that the predicted position is correct. We loop through the predicted position data structure until the node ID is located. Once we locate the correct node we copy the recorded x and y co-ordinates in to the vector called previousPosition. We also copy the time that the node was at those co-ordinates in to the previousTime variable. We store the summation of the currentTime added to the futureTime in the predictedTime variable.

Before we can calculate the nodes predicted position we must first calculate the predicted velocity for the given node. To calculate the predicted x velocity we use the formula

$$\mathrm{v_x} = \frac{\big(\mathrm{x}(\mathrm{t_1}) - \mathrm{x}(\mathrm{t_0})\big)}{(\mathrm{t_1} - \mathrm{t_0})}$$

Where $\mathrm{v_x}$ is the predicted velocity for co-ordinate x. $\mathrm{x}(\mathrm{t})$ is the current x co-ordinate, $\mathrm{t_1}$ is the current time and is the previous $\mathrm{t_0}$ time. To calculate the predicted y velocity we use the formula

$$\mathrm{v_y} = \frac{\big(\mathrm{y}(\mathrm{t_1}) - \mathrm{y}(\mathrm{t_0})\big)}{(\mathrm{t_1} - \mathrm{t_0})}$$

Where $\mathrm{v_y}$ is the predicted velocity for co-ordinate y. $\mathrm{y(t)}$ is the current y co-ordinate, $\mathrm{t_1}$ is the current time and is the previous $\mathrm{t_0}$ time. We then calculate the predicted position of the node by using the formula

$$\mathrm{x(t_2) = x(t_1) + v_x(t_2 - t_1);\quad y(t_2) = y(t_1) + v_y(t_2 - t_1)}$$

Where $\mathrm{x(t_2)}$ is the predicted co-ordinate x and $\mathrm{y(t_2)}$ is the predicted y co-ordinate. We store these values in the vector predictedPosition and return this vector to the invoking function.

As per normal scope rules the vectors currentNodePosition, previousPosition and predictedNodePostion no longer exist after the returned vector.

int CogMan::GetNumberOfNeighbours(int *sourceNodeId,* NodeContainer *nodes*)

The GetNumberOfNeighbours accepts a node identifier and a node container and returns the number of nodes that are in transmission range of the source node identifier passed. The method begins by creating a pointer to the node address that was passed for the given node container that was passed. Next we create a pointer to the mobility model that is aggregated to the node pointer. Once we have the mobility pointer that points to the correct mobility model for this node we invoke the GetPosition method which returns a vector that contains the current location for the source node passed.

Next we loop though the node container that was passed to this method, for each entry in this container (with the exception of the entry that is the same as the node identifier that was passed to this method) we get the location of the node using the same process that we used for the source node that was passed.

We then get the distance between the source node and the node identifier from the node container by calling the **GetDistance** (p. 465) method. We pass this distance to the **InRange** (p. 468) method and if the distance is within the transmission radius we add one to a counter that keeps count of the number of nodes that are in range of the node that was passed to this method. We return the counter once we have processed all the nodes in the node container that was passed.

void CogMan::CheckNeighbours(int *sourceNodeId*)

The **CheckNeighbours** method takes a source node identifier and then for that source node gets the current position of the source node by obtaining a pointer for the mobility model assigned to that node and then invoking the GetPosition method that belongs to the mobility model class.

Then for each and every node in the network (with the exception of this node) we perform similar steps to get the position of the current node being checked. Once we have the position of the source node and the position of the node being checked we invoke the **GetDistance** (p. 465) method which returns the distance in meters between the two nodes. We check if the two nodes are in range of each other by invoking the **InRange** (p. 468) method. If the nodes are in range we write to the myNeighbours vector the source node identifier, the neighbour node identifier, the distance between them and the time at which this information was recorded.

void CogMan::RecordPosition (int *myNodeId*)

The **RecordPosition** method gets the position of the node identifier that has been passed to this method by calling the **GetNodePosition** (p. 464) method. The returned value from **GetNodePosition** (p. 464) consists of a Vector containing the x , y position. We record to a dynamic data structure called myPositions; the node identifier, the current simulation time, and the x, y positions.

double CogMan::TimeAllowedToMoveControlledNode()

The **`TimeAllowedToMoveControlledNode`** method obtains the current simulation time, sets two variables named nodeMovingUntil and backOffUntil to the duratation of the simulation run. Next we check each controllable node in turn to see if it is moving by calling the **`IsControlledNodeMoving`** (p. 463) method and passing the appropriate identifier for the controllable node. The method returns 0 if the node is not currently moving or a value that indicates when the node will be finished moving. If the node is moving and the time at which the node will be finished moving is less than the current value in back off time we update the back off time to reflect when a node will be finished first. Once each node has been checked we return back off time to the invoking method. Zero indicates that one or more control node is not busy, a value greater than zero reflects the time when a node will first become available.

void CogMan::RecordControlledNodeMovement(int *nodeId*, double *endTime*)

The method **`RecordControlledNodeMovement`** is used to record a movement for a controlled node. In particular we want to know if a controlled node is moving or not at a given time. In order to know if a node is moving or not we record to a dynamic data structure the node identifier, the current time and the time that the journey concludes. This data structure is used each time a controlled node is assigned mobility or if a request has been made by a manager node to move a controlled node.

Traffic Flows

void CogMan::CreateTrafficFlows()

The **CreateTrafficFlows** method is where AODV and DSDV traffic flows are constructed. We populate a dynamic data structure for each source and sink (destination) with the time that the source should start transmitting and when the source should cease transmitting. The transmission rate and packet size have already been set in the **ScheduleEvents** (p. 441) method.

Once we have populated the data structure we schedule the flows by calling the **CreateTrafficFlow** (p. 474) method.

void CogMan::CreateTrafficFlow(int *sourceNodeId*, int *destinationNodeId*, Time *startAt*, Time *stopAt*)

This method will accept traffic flow information and create a traffic flow for the AODV nodes by calling the **CreateAodvTrafficFlow** (p. 474) method. Similarly it will create a traffic flow for the DSDV nodes by calling the **CreateDsdvTrafficFlow** (p. 475) method.

void CogMan::CreateAodvTrafficFlow(int *sourceNodeId*, int *destinationNodeId*, Time startAt, Time stopAt)

The **CreateAodvTrafficFlow** method takes the destination node identifier that was passed to the parameter list and obtains that nodes IP address. Using the IP address and the port number an Internet socket address is created. This address allows the destination to forward the data to the correct location. Using this address we then create a local socket address based on the remote address and the UDP. We keep track of our sockets by creating a sink pointer, this pointer points to the interface address of the destination node and also allows us to set up a callback for each packet pointer – we achieve this by calling our **SetupPacketReceive** (p. 478) method.

We specify when the receiver application should start and finish by using the startAt and StopAt variables values that were passed to the parameter list.

The method then moves forward to create the source application. The remote address created prior is used when configuring the sender of the UDP packets. We also set the data rate of the source here. We install the application to a given source node that is specified in the methods parameter list. Similar to the receiving application we set when the source application should begin and when the source application should stop. These times are identical to the times of the destination application that is to receive the packets. Therefore the source application starts and finishes at the same time as the receiving (sink) application.

void CogMan::CreateDsdvTrafficFlow(int *sourceNodeId*, int *destinationNodeId*, Time *startAt*, Time *stopAt*)

The `CreateDsdvTraffic` flow method takes the destination node identifier that was passed in to the parameter list and obtains that nodes IP address. Using the IP address and the port number an Internet socket address is created. This address allows the destination to forward the data to the correct location. Using this address we then create a local socket address based on the remote address and the UDP. We keep track of our sockets by creating a sink pointer, this pointer points to the interface address of the destination node and also allows us to set up a callback for each packet pointer – we achieve this by calling our `SetupPacketReceive` (p. 478) method.

We specify when the receiver application should start and finish by using the startAt and StopAt variables values that were passed to the parameter list.

The method then moves forward to create the source application. The remote address created prior is used when configuring the sender of the UDP packets. We also set the data rate of the source here. We install the application to a given source node that is specified in the methods parameter list. Similar to the receiving application we set when the source application should begin and when the source application should stop. These times are identical to the times of the destination application that is to receive the packets. Therefore the source application starts and finishes at the same time as the receiving (sink) application.

void CogMan::StillTransmitting(int *nodeId*)

The **StillTransmitting** method accesses the myTrafficFlows vector to see if a node is currently transmitting or supposed to be transmitting. First we get the current simulation time by calling the **GetTimeNow** (p. 442) method. Then we step through the myTrafficFlows vector looking for the node of interest based on the source node identifier that was passed to this method. If the source node identifier is found we then check to see if the source node is currently scheduled to transmit data by comparing the assigned start time (when to begin transmitting) and the stop time (when to stop transmitting) with the value returned from **GetTimeNow** (p. 442). If the node is schedule to transmit we return true to the method that invoked this one, otherwise we return false.

void CogMan::CreateTrafficFlowsForCongestion()

The **CreateTrafficFlowsForCongestion** method is where traffic flows intended to cause congestion are constructed. We populate a dynamic data structure for each source and sink (destination) with the time that the source should start transmitting and when the source should cease transmitting.

Once we have populated the data structure we schedule the flows by calling the `CreateCongestionTrafficFlow` (p. 477) method.

void CogMan::CreateCongestionTrafficFlow()

The `CreateCongestionTrafficFlow` flow method takes the destination node identifier that was passed in to the parameter list and obtains that nodes IP address. Using the IP address and the port number an Internet socket address is created. This address allows the destination to forward the data to the correct location. Using this address we then create a local socket address based on the remote address and the UDP. We keep track of our sockets by creating a sink pointer, this pointer points to the interface address of the destination node and also allows us to set up a callback for each packet pointer – we achieve this by calling our `SetupPacketReceive` (p. 478) method.

We specify when the receiver application should start and finish by using the startAt and StopAt variables values that were passed to the parameter list.

The method then moves forward to create the source application. The remote address created prior is used when configuring the sender of the UDP packets. We also set the data rate of the source here. We install the application to a given source node that is specified in the methods parameter list. Similar to the receiving application we set when the source application should begin and when the source application should stop. These times are identical to the times of the destination application that is to receive the packets. Therefore the source application starts and finishes at the same time as the receiving (sink) application.

Network Traffic

Ptr<Socket> CogMan::SetupPacketReceive (Ipv4Address *addr*, Ptr<Node> *node*)

The SetupPacketReceive method accepts an IP address and a pointer of a node. The method creates a pointer to a sink object that is the local application socket. Next we create a local address by using the IP address passed to this method and the default port number. This is then bound to the sink object.

The sink object creates a callback to the method that we specify. In our case when this packet is received at the destination we invoke the `ReceivedPacket` (p. 478) method. Therefore sink will trigger the `ReceivedPacket` (p. 478) method when a packet has been received at the destination node specified.

void CogMan::ReceivedPacket(Ptr<Socket> *skt*)

In order to keep count of the number of packets received we invoke the `ReceivedPacket` method each time a packet arrives at its destination. We keep count of the total number of received packets by incrementing a variable called packetsReceived. We also obtain the size of the packet and add this size to a variable called bytesTotal which allows us to record the total number of bytes received during the simulation run. This method also invokes `PrintReceivedPacket` (p. 478) and passes the pointer to the packet and the pointer to the socket.

std::string CogMan::PrintReceivedPacket (Ptr<Socket> *socket*, Ptr<Packet> *packet*)

The purpose of the PrintReceivedPacket method is to output to screen when a packet was received, which node sent the packet and which node received the packet. We also output the distance between the two nodes engaged in the communication.

Source IP Address

To get the source IP address we create a SocketAddressTag object called tag. We then use the packet pointer that was passed and search the packet for the tag (the Socket Address field in the header of the packet), if this is found the tag from the packet is copied in to the tag object. We then create an Internet Socket Address object called addr and assign the value of an IP address which is obtained as a result of invoking a method called GetAddress() that belongs to the tag object.

Destination IP Address

In order to obtain the destination address we a use the pointer to the socket that was passed in to the parameter list of this method. Using the address that the socket is pointed to we invoke the GetNode() method. We use the value returned from get node to get the ID of the node. Once we have the ID we use the Get() method that is part of the node container call to get the pointer to a node.

Once we have the node pointer we can then invoke the GetObject<ipv4>() method for that node to return a pointer to an IPv4 object (aggregated to that node). There may be more than one network device installed on any given node, therefore next we need to get the IP addresses associated with the first device, we store this in an object called iaddr. There are typically two IP addresses associated with one device, the local address and a loop back address. In order to get the local address that the device is using we invoke the GetLocal() method for the iaddr object created. We store this value in an Ipv4Address object called addri.

We then are able to output to screen the time and destination that this packet was received and the source IP address. In addition to this we also call a method called

RecordReceivedPacket (p. 480) in order to store the IP addresses of the source and destination and the time that the packet was received.

void CogMan::RecordReceivedPacket(Ipv4Address *src*, Ipv4Address *dst*)

The **RecordReceivedPacket** method accepts a source IPv4 and a destination IPv4 address and stores them in variables defined in the parameter list for this method, in this case src and dst respectively.

Next the method calls the **GetProtocol** (p. 483) method and passes the value in the dst variable. If the **GetProtocol** (p. 483) method returns the value "aodv" we increment the totalAodvPacketsReceived by one. If the return value is "dsdv" we increment the totalDsdvPacketsReceived by one.

We then record the source IP address, the destination IP address and the time at which the packet was received in to a data structure called packetReceived. In addition to recording the information into the packetReceived data structure we also record the values in the dynamic data structure called allPackets with the additional value 'R' to indicate that this is a packet that has been received.

Finally as part of the cognition that is implemented we record if the packet that was received belongs to the current protocol that has been selected by the management node. If it is was increment the variable totalSwitchingReceived by one and add the information to the packetReceivedBySwitching data structure.

We deviate from program flow in order to continue our discussion on traffic flow and how we capture other packet information. In the **scheduleEvents** (p. 441) method we create several call back instructions that allow us to make a call back to a function of our choice when certain actions or events occur. For example the following instruction creates a call back that allows us to capture when a packet has been transmitted from the physical layer of a Wi-Fi network device.

```
Config::ConnectWithoutContext("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyTxBegin", MakeCallback(&CogMan::SendPacket, this));
```

Rather than create separate call backs for each node we use the Greek asterisk symbol meaning ‘little star’. In NS3 (and many other computer science applications) the asterisk is a wildcard and means all. Therefore we connect without any context to the **PhyTxBegin** method, each time this method is called a callback is invoked which results in one of our methods being called. The method that is called is **SendPacket** (p. 481). A pointer to the packet is passed into the parameter list of the **SendPacket** (481) method.

void CogMan::SendPacket(Ptr<const Packet> *p*)

The **SendPacket** method is called every time a network device transmits a packet onto the network. The method begins by creating two data structures called scr_ip and des_ip that will hold the source and destination IP addresses from the packet header.

Next we create our own pointer that will hold a pointer to a packet. This is the packet pointer that has been passed into the parameter list of this method.

We create an index that will allow us to loop through each header of the packet. Once the IP header is reached we assign the IP source address to scr_ip and the destination IP address to des_ip. We are interested in the destination IP address of the header to determine if the packet has been sent by an application or is generated to control the network. We check to see if the packet is a network control packet, i.e. a packet transmitted by an address resolution protocol or a route request packet. Or we check if the packet is generated by an application.

In order to determine the origination of the packet we must be able to search the address for the value 255. In order to do this we must first convert the IP address to a string object which we do by calling the `ConvertIPAddressToString` (p. 537) method and pass the IPv4 Address object. Once we have a string object we can make use of the methods associated with the string class, one such method is find.

We check the string object for 255, if the adjacent characters 255 are found in the string object we are assured that the IP address of the packet was not intended for a specific destination. An address with the value 255 indicates a broadcast or route request packet.

We are interested in determining the purpose of the packet for logging purposes, we are not interested in broadcast or route request messages but in packets that contain application data and obtaining part of the IP address allows us to differentiate between these packets.

This method then calls on the `RecordPacketSent` (p. 482) method so that we can record the sent packet to a dynamic data structure for the analysis of the results.

void CogMan::RecordSentPacket(Ipv4Address *src*, Ipv4Address *dst*)

The purpose of the `RecordSentPacket` method is to keep count of the number of packets sent by the AODV and the DSDV routing protocols and to record the packet sent to various dynamic data structures.

Firstly we get the current simulation time by calling the method `GetTimeNow` (p. 442), we store this value in a local variable called timeNow in preparation to be stored in our dynamic data structures.

Secondly we need to establish which routing protocol sent the packet, in order to obtain this information we call a method called `GetProtocol` (p. 483) and send this method the destination IP address from the IP header of the packet. If the packet was generated by the AODV routing protocol we add one to the current value that resides in the scalar variable totalAodvPacketsSent. If the packet was generated from the DSDV routing protocol we add one to the current value that resides in the scalar variable totalDsdvPacketsSent.

We then record the IP address of the source node that generated the packet, the intended destination IP address and the time at which the packet was sent to a dynamic data structure called packetSent. We also record the packet sent information to a dynamic data structure called allPackets and pass an additional value 'S'. The purpose of this data structure is to record all packet information regardless of if they are sent, received or dropped.

If the packet sent was sent by the current routing protocol in operation we also record the packet information to a dynamic data structure called packetSentByCurrentProtocol.

string CogMan::GetProtocol(Ipv4Address *addr*)

The purpose of the GetProtocol method is the return the routing protocol that sent a particular packet. We begin by creating a string variable called protocol, this string variable is returned to the method that invoked this function.

In order to determine the protocol that sent the packet we must be able to search the address for the value of 1 or 2 at a specific location. In order to do this we must first convert the IP address (Ipv4Address object) to a string object which we do by calling the `ConvertIPAddressToString` (p. 537) method.

A string object is essentially an array of characters, thus we can use the array operator [] to specify a particular position in the string. Due to the fact that our IP addresses start with 10.x.y.z where x signifies the network that sent the packet we are able to search the address at position 3 – arrays start at 0 in C++ so we are looking at the fourth character. If the fourth character is 1 then the protocol that sent the packet is DSDV. If the fourth value is 2 then the protocol that sent the message is AODV.

We add a caveat into the algorithm because we do not want to return the protocol that sent the packet if the packet is not generated by an application. Therefore, we finally check the address for the value 255. If the value 255 is found in the IP address we set the protocol variable to "route discovery". This allows the invoking function to deal with the packet accordingly.

In addition to being interested in sent packets we are also interested when a packet is dropped. Therefore we create a similar call backs to handle our dropped packets.

```
Config::ConnectWithoutContext("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/Mac
TxDrop", MakeCallback(&CogMan::MacTxDrop, this));

Config::ConnectWithoutContext("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/Phy
RxDrop", MakeCallback(&CogMan::PhyRxDrop, this));

Config::ConnectWithoutContext("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/Phy
TxDrop", MakeCallback(&CogMan::PhyTxDrop, this));
```

Depending on the stage at which the packet was dropped a call is made to the appropriate method to handle the dropped packet. If for example the packet was dropped at the MAC layer the **MacTxDrop** (p. 485) method is called. If the packet was dropped by the receiving node at the PHY later the **PhyRxDrop** (p. 485) method is called. Finally if the packet was dropped by the sending node at the PHY layer the **PhyTxDrop** (p. 485) method is called.

void CogMan::PhyTxDrop(Ptr<const Packet> *p*)

void CogMan::MacTxDrop(Ptr<const Packet> *p*)

void CogMan::PhyRxDrop(Ptr<const Packet> *p*)

The above `PhyTxDrop`, `MacTxDrop` and `PhyRxDrop` methods work in a very similar way so we discuss them together. The appropriate method receives a copy of the pointed to address for the packet and places this address in to the variable p of the parameter list. We create a copy of the packets memory address (pointer) and place this in variable q.

We then loop through the headers of the packet until the IPv4 header is located. Once we have the correct header we create an object. The IP source address and IP destination address is copied from the packet header and placed in variables src_ip and des_ip respectively.

The data type for the variables src_ip and des_ip are of object IPv4 address. In order to process the address we must first convert to a string object, we do so by invoking the `ConvertIPAddressToString` (p. 537) method and passing the IPv4 address object. We search the string for the value 255 so that we can differentiate between a data packet and a network control packet. We then call a function to record the event of the dropped packet by calling the `RecordDroppedPacket` (p. 485) method.

void CogMan::RecordDroppedPacket(Ipv4Address *src*, Ipv4Address *dst*)

The `RecordDroppedPacket` method works in a very similar way to the `RecordSentPacket` (p. 482) method. We first obtain the current simulation time by invoking the `GetTimeNow` (p. 442) method, we store the return value in a local variable called timeNow which is used when we record the information to our dynamic data structure.

Next we need to establish which protocol dropped the packet by calling the `GetProtocol` (p. 483) method. If the protocol that dropped the packet was AODV we add one to the current value that resides in totalAodvPacketsDropped. If the protocol that dropped the packet was DSDV we add one to the current value that resides in totolDsdvPacketsDropped.

If the packet dropped is a data packet (with a payload) we record the source address and destination address that were passed in to the methods parameter list with the value from the timeNow variable and store this information to dynamic data structure called packetDropped. We store the same information in a dynamic data structure called allPackets with the additional parameter 'D' which indicates that the packet being recorded is a dropped packet.

Next we check to see if the packet dropped was a packet currently being transmitted by the current routing protocol, if it is we also record the packet information to a data structure called packetDroppedByCurrentProtocol.

This concludes the traffic flow section and we return to the flow of the program. Once traffic flows are established we then create a density reference point. In this thesis density is the quantity of nodes in a given area, in our case the area is the transmission radius of a node.

bool CogMan::GetCurrentReceivedPacketsForDestinationFromSource(int *sNode*, int *dNode*, double *timeFrom*, double *timeUntil*)

The **`GetCurrentReceivedPacketsForDestinationFromSource`** method returns the number of packets received for a destination node (dNode) that were sent from a specific source node (sNode) for a given duration (timeFrom → timeUntil).

All packets received are stored in a dynamic data structure called packetReceivedByCurrentProtocol with the time the packet was received, the IP address of the node where the packet originated and the IP address of the node that received the packet (not intermediate nodes). Therefore before we able to search the data structure for the correct source and destination we must first get the IP addresses for the source node identifier and the destination node identifier. We do this by calling **GetIPAddressFromNodeId** (p. 537) for each identifier.

Then we search the data structure (which is sorted by time) for appropriate packets. If we find an entry that contains the source node IP address, the destination node IP address and the time that the packet was received is within the given time period we add one to a counter called pr. We stop searching the data structure when we reach a packet that has been sent earlier than the fromTime or when we reach the beginning of the data structure (whichever comes first).

The value stored in pr is returned to the invoking method.

int CogMan::GetCurrentReceivedPacketsForSource(int *nodeId*, double *timeFrom*, double *timeUntil*)

The **GetCurrentReceivedPacketsForSource** method is used to get the number of packets received at the destination(s) that was sent by the source node identifier passed to this method. The method also accepts a fromTime and a timeUntil that allows us to specify a period to get the number of packets received at the destinations. This allows us to conduct performance analysis when actions have been issued by the management node. Initially we used global variables to keep track of the source and destination but is became apparent quickly that there could be multiple sources and multiple destinations for each source.

Therefore we loop through a dynamic data structure that is used to record all packets received by every destination that is transmitted by the current routing protocol being used. We keep count of the number of packets received that were sent by the source by initializing a counter called pr to 0. We scan though the data structure starting at the end. Each packet that is encountered we check the time that the packet was received. If the packet was received before the value passed in the timeUntil parameter list we then check to ensure that the packet was indeed sent by the source node that we are interested in, we do this by checking the recorded source IP address against the IP address of the node identifier that was passed. We get the IP address of the source node by calling the `GetIPAddressFromNodeId` (p. 537) method; the return value from this method is stored in a local Ipv4Address structure called nodeIP. If the IP address of the source node matches the IP address of the node identifier that was passed to this method we then check that the time that the packet was received was before the timeUntil value that was passed. If we are satisfied that the packet being checked was received during the period specified for the node of interest we increment the pr variable by one. If not we move and check the next packet in the packetReceivedByCurrentProtocol data structure. We stop checking the data structure when we reach a point in the structure that is outside of the time period that we are checking or when we reach the start of the data structure. The value of pr is returned to the method that invoked this method.

int CogMan::GetCurrentSentPacketsForSource(int *nodeId*, double *timeFrom*, double *timeUntil*)

The `GetCurrentSentPacketsFromSource` method takes a node identifier (nodeId) into the parameter list. We use this identifier to get the IP address of the source of interest by calling the `GetIPAddressFromNodeId` (p. 537) and pass this method the node identifier. The return value is stored in a local object Ipv4Address object called nodeIP.

Next we set up an iterator called vectorIndex to step through the **packetSentByCurrentProtocol** data structure. We set the vectorIndex to point to the end of the data structure.

When stepping through the data structure we check if the IP address of the source node in the data structure matches the IP address of the node identifier that was passed to this method. If it does we then check that the time that the packet was received was before the timeUntil value that was passed to the **GetCurrentSentPacketsForSource**. If we are satisfied that the packet being checked was received during the period specified for the node of interest we increment the ps variable by one. We move and check the next packet in the **packetSentByCurrentProtocol** data structure. We stop checking the data structure when we reach a point in the structure that is outside of the time period that we are checking or when we reach the start of the data structure. The value of ps is returned to the method that invoked this method.

int CogMan::GetCurrentDroppedPacketsForSource(int *nodeId*, double *timeFrom*, double *timeUntil*)

The **GetCurrentDroppedPacketsFromSource** method takes a node identifier (nodeId) into the parameter list. We use this identifier to get the IP address of the source node of interest by calling the **GetIPAddressFromNodeId** (p. 537) and pass this method the node identifier. The return value is stored in a local object Ipv4Address object called nodeIP.

Next we set up an iterator called vectorIndex to step through the packetDroppedByCurrentProtocol data structure. We set the vectorIndex to point to the end of the data structure.

When stepping through the data structure we check if the IP address of the source node in the data structure matches the IP address of the node identifier that was passed to this method. If it does we then check that the time that the packet was received was before the timeUntil value that was passed to the **GetCurrentDroppedPacketsForSource** (p. 489). If we are satisfied that the packet being checked was received during the period specified for the node of interest we increment the pd variable by one. We move and check the next packet in the **packetDroppedByCurrentProtocol** data structure. We stop checking the data structure when we reach a point in the structure that is outside of the time period that we are checking or when we reach the start of the data structure. The value of pd is returned to the method that invoked this method.

int CogMan::GetCurrentDropRate()

The **GetCurrentDropRate** method returns the current drop rate for the protocol that is in use. We begin by stepping through the systemState vector starting at the end and moving towards the beginning. Each entry is checked to see if it matches the current protocol that is in use. If it is we record the number of dropped packets in a local variable called currentDroppedPackets and the time that the entry was recorded and store this in a local variable called currentTime. We continue stepping through the vector until the next entry for the current routing protocol is found and store the same information in variables previousDroppedPackets and previousTime.

We then calculate the dropped rate $dr_k(t_0)$ per second at time t_0 for this source node k by applying the following formula,

$$dr_k(t_0) = \frac{dp_k(t_{-1}, t_0)}{t_0 - t_{-1}} = \frac{dp_k(t_0) - dp_k(t_{-1})}{t_0 - t_{-1}},$$

Where $dp_k(t)$ is the number of dropped packets at time t, and thus $\mathrm{dp_k}(\mathrm{t}_{-1}, \mathrm{t}_0)$ is the number of dropped packet per time interval $[\mathrm{t}_{-1}, \mathrm{t}_0]$.

The value calculated is returned to the invoking function.

int CogMan::GetCurrentReceiveRate()

The **`GetCurrentReceiveRate`** method returns the current receive rate for the protocol that is in use. We begin by stepping through the systemState vector starting at the end and moving towards the beginning. Each entry is checked to see if it matches the current protocol that is in used. If it is we record the number of received packets in a local variable called currentReceivedPackets and the time that the entry was recorded and store this in a local variable called currentTime. We continue stepping through the vector until the next entry for the current routing protocol is found and store the same information in variables previousReceivedPackets and previousTime.

We then calculate the received rate $rr_k(t_0)$ per second at time t_0 for this source node k by applying the following formula,

$$rr_k(t_0) = \frac{rp_k(t_{-1}, t_0)}{t_0 - t_{-1}} = \frac{rp_k(t_0) - rp_k(t_{-1})}{t_0 - t_{-1}},$$

Where $rp_k(t)$ is the number of received packets at time t, and thus $\mathrm{rp_k}(\mathrm{t}_{-1}, \mathrm{t}_0)$ is the number of received packet per time interval $[\mathrm{t}_{-1}, \mathrm{t}_0]$.

The value calculated is returned to the invoking function.

int CogMan::RecordPacketRate(int *timeNow*, int *source*, int *sent*, int *received*, double *dropped*)

The `RecordPacketRate` method records the current time, a source node identifier, the number of packets sent, the number of packets received and the number of packets dropped to a vector called packetRates.

Cognition and Network Management

void CogMan::Election()

The purpose of the `Election` method is to select a suitable management node for each source node that is transmitting data across the data communications network. The `Election` method begins by obtaining the current time by calling the `GetTimeNow` (p. 442) method and storing the returned value in the variable timeNow. We then create a vector index and set the value to the beginning of the myTrafficFlows container.

For each node in the myTrafficFlows container we check if the node is transmitting at this current time. If a given node is transmitting we get the current position of that node by calling the `GetNodePosition` (p. 464) method.

We then check each other node in the network to see if it is in range of the source node. We achieve this by getting the position of the node currently being checked by calling the `GetNodePosition` (p. 464) method and then getting the distance between the source node position and the current node being checked position by calling the `GetDistance` (p. 465) method which returns the distance between two vectors. The value returned from the `GetDistance` (p. 465) method is passed to the `InRange` (p. 468) method. The `InRange` (p. 468) method returns a Boolean value of true or false.

If the node being checked is in range we get the predicted time period that the node will be in range of the source node. We get the predicted power remaining in the energy source for the node being checked and we get the density for the node being checked. We achieved this by calling the methods: `GetPredictedMaximalTimeInRange` (p. 468), `GetPredictedTimePowerUntil` (p. 530) and `GetDensityFromNode` (p. 466) respectively.

Once we have the values associated with the predicted maximal time in range, the predicted time the node will have power for and the density for the node we record these values with the node ID by calling the `RecordElectionCandidate` (p. 493) method.

Next we check how many candidates were recorded using a variable called apple. If apple is greater than 0 we call a method to select a suitable candidate from the candidate list. If there were no suitable candidates we schedule the election to run again at some time in the future.

void CogMan::RecordElectionCandidate(int *sourceNodeId*, int *managerNodeId*, double *timeNow*, double *timeInRangeUntil*, double *timePowerUntil*, double *density*)

The RecordElectionCandidate method is used to keep track of nodes that are suitable candidates of the election process. One of these nodes will be chosen as a management node and have the cognition attributes installed. The method accepts a source node identifier (the node that is transmitting), a candidate node identifier, the time that this node became a candidate, how long the candidate node will be in range of the source node, how much time that candidate will have power for and the density for the candidate node. All neighbouring nodes of the source are initially candidates.

In order to store the required data we populate a dynamic data structure called candidatesList and push the required values into the list. We also output to screen the information that we have recorded and how many candidates are currently in the list. This is for informational purposes.

void CogMan::RecordElectionCandidatePercents(int *sourceNodeId*, int *managerNodeId*, double *timeNow*, double *timeInRangeUntil*, double *timePowerUntil*, double *density*)

The `RecordElectionCandidatePercents` method works exactly as the `RecordElectionCandidate` (p. 493) method. The exception is that we record percentages rather than raw values.

void CogMan::SelectManagementNode(int *nodeId*, double *timeNow*)

This method is used to eliminate less suitable nodes that are in range of the source node. We start by creating variables that store maximum values for the candidates, the maximum values are used later when we calculate which node is the most suitable. The variables we use are maxTimeInRange, maxTimeInPower and maxDensity – we initialize each of these variables to the value 0.

Next we create a vector index and point this index to the start of the candidateList vector. We loop though this vector processing each and every node in the list one by one. For each node in the list we must ensure that we are processing a neighbour of the source node that was passed to this method, therefore we use the equality operator. If the candidate node that we are checking is in range of the source node we than check the time that the candidate underwent the candidate election process. If this value is equal to the value that was passed when calling this function we get the value for how long this node will be in range of the source node. If this value is higher than the value stored in maxTimeInRange we update the value contained in maxTimeInRange by setting it equal to the value obtained from the candidate node. We undertake a similar process for maxTimeInPower and maxDensity.

```
  while (vectorIndex < candidatesList.end() )
  {
    if (vectorIndex->sourceNodeId == sNode &&
vectorIndex->timeElected == timeNow)
    {
      if (maxTimeInRange < vectorIndex->timeInRangeUntil)
      {
        maxTimeInRange = vectorIndex->timeInRangeUntil;
      }
      if (maxTimeInPower < vectorIndex->timePowerUntil)
      {
        maxTimeInPower = vectorIndex->timePowerUntil;
      }
      if (maxDensity < vectorIndex->density)
      {
        maxDensity = vectorIndex->density;
      }
    }
    vectorIndex++;
  }
```

Now that we have maximum values for the correct candidates at the correct time for the correct source we can begin a scoring process to identify the most suitable node to undertake the management duties.

We set some more local variables: highestScore, likelyManager, timeInRangeUntilPercents, timeInPowerPercents, densityPercents, timeInRangeUntil, timeInPowerUntil and density. We initialize all the variables to 0 with the exception of likelyManger which we initialize to -1.

We reset our vector index back to the start of the candidate list and for each node that is in range of the source node and is elected at the correct time is scored based on the following formula.

$$\mathrm{p}_1 = \frac{100\ \mathrm{t}_1}{\mathrm{m}_1}$$

Where p_1 is the timeInRangeUntilPercents, t_1 is the timeInRange until and m_1 is the maxTimeInRange.

We apply a similar formula to work to obtain the timePowerUntilPercents. Where p_2 is the timeInPowerUntilPercents, t_2 is the timeInPowerUntilRange until and m_2 is the maxTimeInPower.

$$p_2 = \frac{100\ t_2}{m_2}$$

We apply a similar formula to work to obtain the densityPercents. Where p_3 is the densityPercents, d is density until and m_3 is the maxDensity.

$$p_3 = \frac{100\ d}{m_3}$$

Once we have the values as a percentage out of 100 we apply a weighting system to give each node an overall score.

$$s = (p_1 \times w_1) + (p_2 \times w_2) + (p_3 \times w_3)$$

Where s is the score w_1 is the weight applied to range, w_2 is the weight applied to power and w_3 is applied to density.

If the current score for the node is higher than the current value in highestScore we update highestScore to reflect the new highest score. In addition to this we also copy the candidate node ID into likelyManager, timeInRangeUntil into timeInRangeUntil, timePowerUntil into timePowerUntil and density in to density. We copy from the vector into local variables.

Once every node has been checked we pass the source node identifier and the values contained in likelyManager, timeNow, timeInRangeUntil, timePowerUntil and density to the **`RecordElectionWinner`** (p. 497) method to record this candidate as the winner of the election and to assign as a management node.

void CogMan::RecordElectionWinner(int *sourceNodeId*, int *managerNodeId*, double timeNow, double *timeInRangeUntil*, double *timePowerUntil*, double *density*)

The `RecordElectionWinner` method is used to keep track of the management nodes. The method accepts a source node identifier (the node that is transmitting), a manager node identifier, the time at which point the manager will be in range of the source and the remaining power of the management node. These values are stored in a vector called managersList and is of object type managerNode.

The method also accesses the last entry in the vector and outputs to screen the manager node identifier, what time it was elected, how long it will be in range for, the current density for informational purposes.

The final purpose of the `RecordElectionWinner` is to invoke the `Management` (p. 497) method and pass the values for the source node and the management node.

void CogMan::Management(int *sNode*, int *mNode*)

Relationship Status

The Management method is where the cognition is applied to the management node. Various steps are undertaken that first check the viability of the relationship between the management node and the source node. If the relationship is not viable the re-election process is performed so that the source node can be assigned a new management node. This should be completed within a timely fashion, if not all previous knowledge is lost.

First we check the source is still transmitting by calling the `StillTransmitting` (p. 476) method and passing the identifier of the source node. If the source node has finished transmitting then we do not call the `Management` method any longer for this node – unless it begins transmitting at a later date.

If the source node is still transmitting we then get the current simulation time by calling the **`GetTimeNow`** (p. 442) method. The time is used to predict if the management node will be in range of the source node at a given time in the future, we achieve this by calling the **`WillBeInRange`** (p. 535) method and pass the source node identifier, the management node identifier, the current time t_0 added to a time value that reflects the time at the end of the next interval that the management method will be called. If we predict the source node will not be in range of the management node we perform the election process again to choose a more suitable management node.

If the source node will be in range of the management node at the time of the next checking interval t_1 we continue and check the battery power of the energy source(s) for the management node. We do this by calling the **`WillBeAlive`** (p. 533) method and we pass the management node identifier and the next time t_1 that the **`Management`** (p. 497) method will be called. If the battery level has almost depleted and the management node is unlikely to have enough power to transmit or receive radio waves at t_1, we perform the election process again.

Providing the management node will have enough power to continue its role and providing the management node will be in range of the source node, we then save to a dynamic data structure the source node identifier, the management node identifier, the current range from the source node to the management node, the current energy level of the energy source(s) for the management node and the current time. We save this information by calling the **`RecordRelationshipState`** (p. 520) method and passing the source node identifier, the management node identifier and the current simulation time. This information allows us to make more accurate predictions on the relationship status of the management node and the source node.

Once the management node has established that the relationship between our source node and the management node is viable we then check the state of the network.

State of the Network

At time now t_0 we first get the number of packets $rp_k(t_{-1}, t_0)$ received from the source node **k** for the last period (t_{-1}, t_0) (the period is from when the management method was last called until now) by calling the method **`GetCurrentReceivedPacketsForSource`** (p. 487). We also get the packets sent by the source $sp_k(t_{-1}, t_0)$ and the packets dropped $dp_k(t_{-1}, t_0)$ for the **k**-th source node for this period by calling the **`GetCurrentSentPacketsForSource`** (p. 488) method and by calling the **`GetCurrentDroppedPacketsForSource`** (p. 489) method respectively. We store the return values from the three methods called into local variables packetsReceived, packetsSent and packetsDropped respectively.

Next we calculate the receiving rate per second for the destination node(s)

$$rr_k(t_0) = \frac{rp_k(t_{-1}, t_0)}{t_0 - t_{-1}}$$

and store this value in a variable called recPerSec where $rr_k(t_0)$ is the receiving rate of the destination node(s) for this source node k at time t_0. We perform similar calculations to derive the sending rate per second $sr_k(t_0)$ and the drop rate $dr_k(t_0)$ per second for this source:

$$dr_k(t_0) = \frac{dp_k(t_{-1}, t_0)}{t_0 - t_{-1}} \; ; \; sr_k(t_0) = \frac{sp_k(t_{-1}, t_0)}{t_0 - t_{-1}}$$

Then we get the previous rates (for t=t$_{-1}$) for the k-th source node: $sr_k(t_{-1}), dr_k(t_{-1}), rr_k(t_{-1})$, we do this by looping though a vector called packetRates. We start our search at the end of the vector and move towards the beginning searching for the appropriate source node (source=k). Once the correct source node is found we copy from the vector the received, sent and dropped rates with the time that the information was recorded. This information is used shortly to calculate the current flow (in percents) rate between the source and the destination(s):

$$fr_k(t_0) = \frac{rr_k(t_{-1}, t_0)}{sr_k(t_{-1}, t_0)} * 100.$$

Just before this calculation we record the current packet rates ($sr_k(t_0), dr_k(t_0), rr_k(t_0)$) to the same vector that we obtained the previous packet rates ($sr_k(t_{-1}), dr_k(t_{-1}), rr_k(t_{-1})$) from, we record the current time (t_0), the source node identifier (k), the number of packets sent, received and dropped per second.

Before calculating the flow rate we first much check that the number of sent packets per second is not equal to zero in order to avoid infinity errors. If the number of packets sent per second is not equal to zero we apply a formula to calculate the current flow rate per second which is the current received rate per second divided by the current send rate per second. We store the flow rate in a variable called currentFlowRate $fr_k(t_0)$.

We apply the same formula to the previous acquired values $sr_k(t_{-1})$, $rr_k(t_{-1})$ to get the previous flow rate and store this value in a variable called previousFlowRate ($fr_k(t_{-1})$).

At this stage we have access to the totals for the packets received, sent and dropped for this checking period and the same information for the previous period. This information is based on the current routing protocol that is in operation or in the case of the previous period the routing protocol that was in operation.

We are also interested in how the network would be behaving if an alternative routing protocol was being used; we do this by invoking the **`PrintAllPacketStats`** (p. 542) method and sending the information that relates to this period in time (the start time of the period and the time now). We do not use this information in the cognition process because in a realistic scenario this information would not be available – we do however find it useful to quickly compare if the decisions made by the management node are having a positive or negative impact on the performance of the network.

Next we check to see if the network is not in a state of fluctuation due to a prior action being initiated, for example if we had just changed routing protocol, switched channel or moved a controllable node we give the network time to adjust to this change and to become stabilized. The amount of time that we allow to stabilize is based on the action that was performed. The time to stabilize also guards us against other competing management nodes putting into action their decisions when a prior decision has just occurred. All management nodes are instructed to back off until the time to stabilize period has passed.

If the network is not in a state of fluctuation we compare the current flow rate against the previous flow rate. Should the currentFlowRate be equal to zero or should the currentFlowRate be lower than the previousFlowRate, i.e. $fr_k(t_{-1})$> $fr_k(t_0)$, it can be assumed that network performance has decreased (we have already checked that the source is still sending packets, that the relationship beween source and destination is viable and that the network has stabilised). However, before the management node requests a change that could influence the network state we perform another check to see if this degradation in performance is a momentarily fluctuation of if a trend is occurring and indeed the performance of the network is degrading. We do this by getting the previous previous sending rate ($sr_k(t_{-2})$) and the previous previous receiving rate ($rr_k(t_{-2})$). We calculate the difference between the previous previous sending rate ($sr_k(t_{-2}) - sr_k(t_{-1})$) and the previous previous receiving rate ($rr_k(t_{-2}) - rr_k(t_{-1})$) and store this in a variable called previousPreviousFlowRate. Should the previousPreviousFlowRate be higher than the previousFlowRate, i.e. $fr_k(t_{-2})$> $fr_k(t_{-1})$, then the rate of flow is decreasing and we dismiss the reason as a momentarily fluctuation and allow the management node to consider its options, to better inform the manager we use least squares approximation to obtain a predicted receive rate if the trend of the network continues along this path.

The network will use the least squares approximation algorithm to get the predicted receive rate by using the following six values for the k-th source node

$fr_k(t_{-2})$	$fr_k(t_{-1})$	$fr_k(t_0)$
t_{-2}	t_{-1}	t_0

The equation of the best fit straight line is

$$fr_k(t) = a_k + b_k t,$$

$$E_K = (a_k + b_k(t_0) - fr_k(t_0))^2 + (a_k + b_k(t_{-1}) - fr_k(t_{-1}))^2 + (a_k + b_k(t_{-2}) - fr_k(t_{-2}))^2 \quad \rightarrow min$$

The gradient of the best fitted straight line is

$$b_k = \frac{3\sum_{j=-2}^{0} fr_k(t_j)t_j - \sum_{j=-2}^{0} fr_k(t_j)\sum_{j=-2}^{0} t_j}{3\sum_{j=-2}^{0} {t_j}^2 - (\sum_{j=-2}^{0} t_j)^2}$$

And the shift is

$$a_k = \frac{\sum_{j=-2}^{0} fr_k(t_j) \sum_{j=-2}^{0} t_j{}^2 - \sum_{j=-2}^{0} fr_k(t_j)t_j \sum_{j=-2}^{0} t_j}{3\sum_{j=-2}^{0} t_j{}^2 - (\sum_{j=-2}^{0} t_j)^2}$$

Thus NCC is being passed a string of predicted values $\{fr_k(t_1)\}$, $k \in \{1, \dots, n\}$.

The result from the calculation above is stored in a variable called estimatedRecPerSec and is checked against a lower and higher boundary. If the estimatedRecPerSec value is a negative value we change the value to 0 (the lower boundary). If the estimatedRecPerSec value is greater than the maximum receiving rate we set estimatedRecPerSec to the maximum receiving rate (the higher boundary).

Next we get the density for the management node by calling the **`GetDensityFromNode`** (p. 466) method and we get the channel that the management node is operating on by calling the **`GetChannel`** (p. 534) method. We then set an instinctive feeling (gutFeeling) for the action that the management node might request based on the current information available (before the management node recalls on its experience and the outcomes of past events when there was a similar network state). The information we use to set the instinctive feeling is based on current flow rate, previous flow rate, previous previous flow rate and the number of nodes in direct communication range of the manager node. We get the number of neighbours by calling the **`GetNumberOfNeighbours`** (p. 471) method. The instinctive feeling is set to C, M or P where C is the feeling to request to change frequency channel, M is the feeling to request control of a node and P is the feeling to request a change in routing protocol.

We only allow the management node the opportunity to act upon their intuition if after consideration there is no other alternative. The management node considers alternative actions by calling the **`GetAction`** (p. 512) method.

Providing the management node has not been told to back off for all available actions a request will be made by the management node. We record the requested action to a data structure by invoking the **`RecordActionRequest`** (p. 507) method and passing all the available data (current time, the source node that the request is on behalf of, the management node making the request, the drop rate per second, the receive rate per second, the send rate per second, the density, the current operating channel, the current protocol and the requested action). The action is then requested by calling the **`RequestAction`** (p. 507) method.

What remains is the management node to instruct the Network Command Center to consider the requested action. If the network has had sufficient time to stabilise since a previous request and if the Network Command Center is not already schedule (by this manager or another manager) the management node will request the actions to be considered by the Network Control Center immediately. However, if the Network Command Center has a previous request pending the management node will schedule a request at some point in the future. The time in the future is four seconds after the time to stabilise period has passed. In either case (immediately or at a scheduled future time) we call the **`NetworkCommandCentre`** (p. 508) method to consider the requested action.

Should this be the case the management node associated with source node k can instruct a controlled node to move to a new location in order to re-establish or strengthen the link. The location that the controlled node will move to is based on the current location of the destination, say coordinates $(x^d(t_0), y^d(t_0))$, the current location of the source $(x^k(t_0), y^k(t_0))$, and the location $(x^b(t_0), y^b(t_0))$ of the node that has the most dropped packets for the current source node. We use predict position method by calling **`PredictPosition`** (p. 469) to calculate the predicted location of the source and destination nodes at time $t = t_1$, $(x^k_{predicted}(t_1), y^k_{predicted}(t_1))$, $(x^d_{predicted}(t_1), y^d_{predicted}(t_1))$, and send the controlled node to the new goal:

$$(x^c_{goal}(t_1), y^c_{goal}(t_1)) = (\frac{x^k_{predicted}(t_1)+x^d_{predicted}(t_1)}{2}, \frac{y^k_{predicted}(t_1)+y^d_{predicted}(t_1)}{2}),$$

i.e. to the mid-point between the two predicted locations. Repeating this procedure at each checking interval we have better and better approximations of the predicted locations. We finish when the connection is re-established.

Once we have established that the network performance for source m is degrading we calculate the gradient for the two intervals $(\mathrm{t}_{-2}, \mathrm{t}_{-1}), (\mathrm{t}_{-1}, \mathrm{t}_0)$, i.e. $\mathrm{gfr_k}(\mathrm{t}_0) < 0$, network would base its action on the previous gradient $\mathrm{gfr_k}(\mathrm{t}_{-1})$. If the previous gradient $\mathrm{gfr_k}(\mathrm{t}_{-1}) > 0$, was positive, in this case the manager will not perform the action this time but will retain this information and increase the checking frequency so that the next checking time, i.e. t_1, is halved not to miss this trend in performance at the next checking interval time.

However, if $-1 < gfr_k(t_{-1}) < 0$, the manager will look through the history of flow rates (searching through $\{t_i\}, i < -2, t_i \quad \rightarrow 0 \; as \; i \; decays$) to find a similar state of the network and then will look at the history of the actions to make a decision. So the next stage in decision making procedure will consist of two major steps: looking for the systemstate structure (full history of the flow rates) and looking through the archive of performed actions with later reflections on how clever they were.

The management node will look through the actions taken at time t_i, and since this historic time is similar to present network conditions (state of the network) the network will apply an action applied at time t_i, provided it was successful. Each node k's manager then produces either 0 or 1 as a request for switching protocols. So NCC is being passed the string of zeroes and ones: $\{j_k\}, j \in \{0,1\}, k \in \{1, \dots, n\}$.

At time t_0 NCC collects requests from all managers for switching the protocol. The set of collected data will look like: {0,1,1,0,1,...,1,1,1,0}. The size of this set is n (the number of source nodes), together with it, NCC gets the predicted values $\{fr_k(t_1)\}$, $k \in \{1, \dots, n\}$.

FR(t)=fr_1(t)+fr_2(t)+fr_3(t)+fr_4(t)+...+fr_n(t) , where fr_m(t) is the current flow rate calculated for each node m at time t. NCC also calculates the predicted performance based on predicted by managers rates: FR(t_{k+1})=fr_1(t_{k+1})+fr_2(t_{k+1})+fr_3(t_{k+1})+fr_4(t_{k+1})+...+fr_n(t_{k+1}). Compare the two values: FR(t_{k+1}) and FR(t) to check if the increase is expected.

Step 5 Then it calculates FR(t_{-1}) and FR(t_{-2}) and repeats the procedure of finding the most similar situation in the history (Step 1 and Step 2 above) and makes a decision based on previous actions.

By moving through time points (on the current protocol) towards past (t_k,t_{k-1},t_{k-2}), k<t_{-2}, each time the network evaluates the following error function

$$E^i = (FR(t_i) - FR(t_0))^2 + (FR(t_{i-1}) - FR(t_{-1}))^2 + (FR(t_{i-2}) - FR(t_{-2}))^2$$

$$= \sum_{l=0}^{2} (FR(t_{i-l}) - FR(t_{-l}))^2 \ \to min$$

The set of values E^i is then examined on the minimal value. The set of times (t_i,t_{i-1},t_{i-2}) that bring minimum to the error function are treated as similar to present state and the network is ready to move to step 2. The NCC will look through the actions taken at time t_i, and since this historic time is similar to present state of the network the network will apply an action applied at time t_i provided it was successful.

void CogMan::RecordSystemState(int *sNode*, int *mNode*)

The `RecordSystemState` method records the current state of the system to a vector called systemState. The method begins by incrementing a counter to indicate if the network has switched protocol. Next it will get the density for the manager node by calling the `GetDensityFromNode` (p. 466) method. Next we get the current operating channel for the management node by calling the GetChannel (p. 534) method. Then we update the systemState vector.

void CogMan::RequestAction(double *atTime*, int *mNode*, char *request*, double *prr*)

The `RequestAction` method adds the requested action to a data structure called newRequests. The information added is: the time of the request, the management node making the request, the requested action and the predicted receive rate should no action be taken.

The network command will also acknowledge the request and the management nodes memory will be updated to reflect that the request has been acknowledged. This is achieved by setting the requestProgress value of the managerNodeMemory vector to 'A'.

void CogMan:: RecordActionRequest (double *timeNow*, int *sNode*, int *mNode*, double *droPerSec*, double *recPerSec*, double *senPerSec*, double *density*, int *channel*, string *currentProtocol*, char *request*)

The `RecordActionRequest` method commits to memory the current time, the source node being managed at this time, the identifier of the management node, the drop rate per second, the received rate per second, the send rate per second, the current density for the management node, the current operating channel, the current protocol and the requested action.

void CogMan:: NetworkCommandCentre()

The `NetworkCommandCentre` method is called by a management node(s) to consider their requests – the request could be authorized or denied. It could transpire that the network is in a state of fluctuation, if this is the case the Network Command Centre (NCC) will be scheduled to be called at some point in the future. When the Network Command Centre is called the NCCschedule variable will be set to false to show that the Network Command Centre is not currently scheduled and therefore can be scheduled by a management node.

Each time the Network Command Centre is called we set a time to stabilise which is 2 seconds by default.

The NCC is to consider the current requests of the manager nodes. There may be one or many requests depending on the number of manager nodes and the time elapsed since the NCC was last called. We want to allow manager nodes the ability to make requests even if the NCC is busy dealing with current requests. Therefore we copy the current requests into a data structure called newRequests and clear the requests data structure. This allows the manager to deal with the current requests but also allows managers to make requests in the normal way. Any requests made whilst the NCC is busy will be dealt with the next time the NCC is called.

The NCC will only authorize one request each time the NCC is called upon, this is to prevent manager nodes cancelling out each other's requests. For example, consider manager node 2 makes a request to switch to another routing protocol and manager 4 requests the same action. If the NCC authorized both requests we may be in a situation of switching twice.

Therefore the NCC keeps counters: M (control of mobile node), C (change frequency channel) and P (switch protocol). The NCC will attempt to authorize the action with the most requests, however, it could transpire that the most requested action is not suitable, in this case the NCC will then consider the next most popular request.

If the most popular request was to move a controlled node the NCC has to consider carefully which manager should be authorized to instruct the controlled node, this is because the number of controlled nodes in finite. In our simulations that had full cognition switched on we typically set the number of controlled nodes to 2, however, this is at the discretion of the application of the network and can be increased or reduced depending on circumstance.

We do not want to be in a situation where the NCC allows the use of a controlled node shortly after already authorizing a previous manager the use of the same controlled node. Therefore we use a back off time that allows the previous action(s) of the controlled nodes to complete. In order to get the minimum back off time (a time of zero indicates that a controlled node is not currently busy performing a previous request) we call the **`TimeAllowedToMoveControlledNode`** (p. 473) method.

Should we have a situation where all controlled nodes are busy performing a previous request the NCC will set the M counter to 0 so that other actions may be considered. In addition to this it will deny all requests from any manager that has requested the use of a controlled node and send them a back off time so that the manager cannot make the same request again until at least one controllable node becomes available. We achieve this by the NCC calling the **`DenyAllRequestForControlledNode`** (p. 516) method.

Providing that there is at least one controllable node available the NCC should carefully consider which manager node(s) to allow the use of this resource. We stated earlier that the NCC will only authorize one request per NCC call and that there may be several requests to consider. In the case of controllable nodes we could have a situation where there are multiple manager nodes requesting the use of a controllable node, if the controllable nodes are available the NCC will assign the available controlled nodes. Consider for example we have three manager nodes that have requested the use of a controllable node and that we have two controllable nodes. The NCC will make an effort to accommodate the request for two of the manager nodes and deny the request for the third. We considered using a first come first serve scheduling algorithm to accommodate this, however, the NCC should be more interested in improving overall network performance. Therefore we applied a scheduling algorithm based on current receiving rates and predicted receiving rates which are explained when we discuss the **`GetManagerToMoveNodeFor`** (p. 516). Once we have established which manager node to move the controllable node for we predict how long the manager should make use of the controllable node so that we know when we can reallocate this resource again or as the case maybe if the NCC should deny this resource at some point in the future. This value is also used by the NCC in its final consideration to allow or deny the request of the resource. We get the predicted time that the controllable node will be in use by calling the **`MoveControlledNode`** (p. 522) and pass the appropriate manager identifier. There may be a situation where a manager node is looking after multiple source nodes, in this situation the manager will select the most appropriate source node and this is explained when we discuss the **`MoveControlledNode`** (p. 522) method.

The NCC makes one final check to see if the use of the controllable node is suitable for this situation. It does this by checking how long the controllable node will be used for, if the value is negative (which is the indication that the source node will already be in range of the destination node before the controllable node reaches its goal destination) the NCC will deny this request. It does this by calling the **`DenyARequestForControlledNode`** (p. 516) and passes the appropriate manager identifier and a back off time so that the same manager will not make the same request for this given situation.

If the NCC is satisfied that the use of the controllable node will be of some use for the manager node that requested this action, the NCC will authorize the request by calling the **`GrantRequestForControlledNode`** (p. 519) and pass the appropriate manager identifier and a back off time so that this manager cannot make another request to move a controlled node until the current request has been completed.

If the most popular request was to change channel, of if the second most popular request was to change channel and the most popular request was to move a controllable node but the NCC denied all requests for that action the NCC will switch channel. In order to switch channel the NCC first checks which channel the data communications network is operating on by calling the **`GetChannel`** (p. 534) method and passing an arbitrary node number (all nodes transmit on the same channel). Once the current operating channel has been established, the NCC will instruct all nodes, including the controllable nodes to switch to the alternative channel by calling the **`SetChannel`** (p. 534) method and the **`SetChannelControlledNodes`** (p. 535) method. Once the action has been completed the

NCC gets a back off time for this action by calling the **`GetPeriodToStablise`** (p. 517) method. The NCC will also send information to the management nodes that requested the change channel action with the back off time by calling the **`GrantRequestForChangeChannel`** (p. 521) method. The final action that occurs if we change channel is to print the channel each node is operating on. This does not alter the state of the network and is for information purposes so that we are assured that channels have changed correctly for each and every node, we do this by calling the **`PrintAllChannels`** (p. 544) method.

If the most popular request was to switch routing protocol, of if the second most popular request was to switch routing protocol and the most popular request was to move a controllable node but the NCC denied all requests for that action the NCC will switch routing protocol. First we get a back off time for this action by calling the **`GetPeriodToStablise`** (p. 517) method. Next the NCC will inform all the management nodes that requested to switch routing protocol that this action has been authorized, it does this by invoking the **`GrantRequestForSwitchProtocol`** (p. 517) method. The NCC then instructs all nodes to switch to an alternative routing protocol by calling the **`SwitchProtocol`** (p. 520) method.

In the unlikely event there is the same number of requests for all the actions a message is generated informing us of a three way tie.

double CogMan::GetAction(char *gutFeeling*, int *mNode*, double *droPerSec*, double *recPerSec*, double *senPerSec*, double *density*, int *channel*, string *currentProtocol*)

The **`GetAction`** method is the method where the management node considers its memory of previous actions and the consequences of those actions in order to make a decision that will best serve the network in regards to performance by using the current network state.

We have a vector called managementNodeMemory that stores the memory of past events. The management node loops through its memory looking at past events. We assume that historic events may be forgotten so we include a pastCount, this allows us to control how far back in time the manager can recall memories of previous requests or actions. The management nodes begins to recall memories starting with the most recent and working backwards to the lesser recent memories.

When the management node puts a request to the Network Command Center to change the operation of the network the Network Control Center will consider the request and allow or deny the request. In either case the Network Command Center will send a back off time to the management node to prevent the management node making the same request until the back off time period has elapsed. The management node is able to request three different actions and we may have a situation where the management node has sent back off times for each of the three requests, therefore to prevent the management node making any further requests we have included a strikeCount. If the strike count becomes equal to 3 the management node is not allowed to make any further requests until the minimum back off time has elapsed.

The management node as it recalls previous actions from memory is only interested in the actions that resulted in a positive outcome (i.e. performance of the network increased), therefore, the manager will only consider events that have resulted in a positive outcome. If the management node recalls a previous memory that did result in a positive outcome the management node will recall the drop, receive and send rates for this period, in addition to the density, the operating channel and the routing protocol that was in use. Using the information obtained the management node will attempt to calculate the how similar the conditions were then from what they are now, this is achieved by performing the following calculation and will give us a value that we call the distance.

For each memory line, that corresponds to the moment $t = t_m$, we calculate dropped, received and sent errors comparing to the current (at $t = t_0$) rates:

$$DR_k(t_m) = (dr_k(t_0) - mdr_k(t_m))^2 * w_d$$

$$RR_k(t_m) = (rr_k(t_0) - mrr_k(t_m))^2 * w_r$$

$$SR_k(t_m) = (sr_k(t_0) - msr_k(t_m))^2 * w_s$$

Here above $w_d = 0.1, w_r = 0.5, w_s = 0.4$ are the weights we use to stress the importance of receiving rates. Also, $mdr_k(t_m), mrr_k(t_m)$, $msr_k(t_m)$ are correspondingly the dropped, received and sent rates taken from the memory of the manager node. Hence we calculate the distance from current state of rates to the one that happened at $t = t_m$

$$distance_m^k = \ DR_k(t_m) + RR_k(t_m) + SR_k(t_m),$$

and find the state at the minimal distance from $t = t_0$.

We then check to see if the distance from this memory is less that the already established closest distance. The closest distance is the action that has the most similarities in network state. For the closest distance we obtain the action that was performed previously, we record this action so that the same action can be requested.

There may be a situation in which the current memory being recalled does not have back off time attached and shows that the performance of this action hindered network performance. If this is the case we update counter for that action, for example if the action was to change channel we add the performance value to CCount. We do this so that we can establish the least hinderer's action.

Once we have recalled from memory the past actions we will either be in a position where the management node has three strikes, an action that resulted in a positive outcome which reflects the current state of the network or three counters with values derived from past experiences when performance did not improve.

If the management node has three strikes that action sent back to the method that invoked this function is the minimum back off time.

Otherwise, if at this stage an action has not been determined, i.e. in the case of not being able to establish a positive outcome from memory (i.e. if the management node is new to the role) we allow the manager to go with their intuition (gut feeling). However, this is only the case if their intuition does not conflict with the manager being told to back off for this action. For example, consider a manager making a request for the use of a controlled node, the Network Control Center authorizes this requests and asks the manager to back off from making the same request again until a certain time has elapsed (normally until the node has reached a given location). We can and do override the decision based on intuition if there is evidence that the gut feeling has some history of hindering network performance rather than improving. If this is the case we set the action to that of a better informed decision using the performance counters.

If the managers gut feeling was to make a request that has a back off time attached or if we cannot reach a decision based on performance counters then an action is chosen at random.

When choosing a random action we have taken steps to ensure that that random action is not one that has a back off time attached.

void CogMan:: DenyAllRequestForControlledNode(int *backOffUntil*)

The `DenyAllRequestForControlledNode` method will update all manager nodes that have requested the use of a controlled node and instruct that they back off from asking for the same request again until the back off time has elapsed. Each manager's nodes memory will be updated to reflect that the request was denied and when it is appropriate to make the same request again.

void CogMan::DenyARequestForControlledNode(int *backOffUntil,* int *mNode*)

The `DenyARequestForControlledNode` method will update a specific manager node that has requested the use of a controlled node and instruct that they back off from asking for the same request again until the back off time has elapsed. This manager nodes memory will be updated to reflect that the request was denied and when it is appropriate to make the same request again.

int CogMan::GetManagerToMoveNodeFor()

The `GetManagerToMoveNodeFor` method will establish which manager is most in need of the use of a controllable node. The method begins setting a minimum receive rate to 999999 and a mNode variable to 0. Next the method will check each manager node that has made a request to move a controllable node. We check the predicted receive rate for each manager to determine which manager is most in need of a controllable node. Should the node that is being checked have a predicted receive rate that is less than the minimum receive rate the minimum receive rate is updated to reflect the predicted receive rate for that node. We also update the mNode variable so that we can keep track of the manager node. Once we have checked all manager nodes that have requested the use of a controllable node we return the management node identifier to the invoking function.

double CogMan::GetPeriodToStablise(string *action*)

The `GetPeriodToStablise` returns a value that indicates how long the Network Command Center should wait before considering new action based on the current action just performed.

void CogMan::GrantRequestForSwitchProtocol (double *backOffTime*)

The `GrantRequestForSwitchProtocol` method grants the request for each manager node that requested to switch routing protocol. In addition to granting the request a back off time is sent so that the manager cannot make the same request again until the back off period has elapsed. Each manager that was granted this request will call the `RecordActionOutcome` (p. 517) in order to record if performance has increased or decreased for the node that they are managing at the time specified in the back off time. We also call the `PrintVectorOfMemoryEntries` (p. 544) once all the requests have been granted for this period. This is for informational purposes only and does not alter the state of the network.

void CogMan::RecordActionOutcome(double *atTime*, int *mNode*, char *action*)

The `RecordActionOutcome` method will record the outcome of any actions authorized by the NCC. We use a relatively simple process in establishing if the outcome improved the performance for the source node that the management node requested the action for.

First we get the current simulation time by calling the `GetTimeNow` (p. 442) method and store this in a local variable called timeNow. We also create a variable called outcome and initialize this to 0.

The management node memory is checked to establish which source node the action was requested for by searching the memory for the time of the request (atTime) and the action that was requested (action) for the appropriate management node (mNode).

Once the correct entry has been found we record the source node by storing the node identifier in a variable called sNode. Then for this source node we get the number of packets received for a specific duration which is from the time of the request until the current time by calling the `GetCurrentReceivedPacketsForSource` (p. 487) method. We store this in a local variable called packetsReceived. Next we perform a similar action to establish how many packets were sent for this source node for the same period by calling the `GetCurrentSentPacketsForSource` (p. 488) method and store this in a local variable called packetsSent. We do the same for the number of packets dropped for this source using the same period by calling the `GetCurrentDroppedPacketsForSource` (p. 489) and store this in a variable called packetsDropped.

We use the number of received, sent and dropped packet values and calculate the number of packets received per second, sent per second and dropped per second by applying the following formulae:

$$rr_k(t_0) = \frac{rp_k(t_{-1}, t_0)}{t_0 - t_{-1}} , sr_k(t_0) = \frac{sp_k(t_{-1}, t_0)}{t_0 - t_{-1}} , dr_k(t_0) = \frac{dp_k(t_{-1}, t_0)}{t_0 - t_{-1}}$$

Now that we have the received, sent and dropped rates per second we can compare these rates against the previous rates (the rates recorded at the time of the request). For example, we compare the current received rate per second against the recorded received rate per second when the management node made the request. If the current received packet rate has increased then network performance has increased and we add one to outcome, if the current received packet rate is less than the previous received packet rate we subtract one from outcome to indicate that network performance has decreased. Should the current receive rate and the previous receive rate be identical we do not change the value of outcome.

We perform similar actions for dropped packets. We add one to outcome if the number of dropped packets for this period has decreased and deduct one from outcome if the number of dropped packets has increased. In relation to send rate we add one to outcome if the rate of sending packets has increased and deduct one from outcome if the rate of sending packets has decreased.

This gives us normalized values of -3, -2, -1, 0, 1, 2 and 3. Zero indicates that the has not been any change in regards to performance for the source node, -1 indicates that performance has decreased slightly, -2 indicates that performance has decreased, and -3 indicates that performance has dropped further. The positive values indicated performance has increased. One indicates a slight increase in performance, two - an increase in performance and 3 indicates that performance has increased further.

The outcome is recorded for this manager node and used when evaluating future recommendations by this manager node.

void CogMan:: GrantRequestForControlledNode (int *manager*, double *backOffTime*)

The `GrantRequestForControlledNode` method will grant authorization for a manager node the ability to move a controlled node. The method begins by getting the current simulation time by calling the `GetTimeNow` (p. 442) method. Next the appropriate manager will update its memory to reflect that authorization has been given that shows it has been permitted use of a controlled node. The manager also updates its back off time so that it cannot make the same request again until this request has completed. In addition to this the manager will schedule and event based on the back off time in order to record the outcome of this action by calling the **`RecordActionOutcome`** (p. 517) method. The ability to schedule when to check the outcome of the action allows the management node accurate times to check if performance has increased or decrease for the given action. This method also calls upon the **`PrintVectorOfMemoryEntries`** (p. 544) so that we can see memory management is being access and updated correct.

bool CogMan::RecordRelationshipState(int *sNode*, int *mNode*)

The **`RecordRelationshipState`** method accepts a source node and a management node identifier. We get the remaining energy of the sNode by calling the **`GetRemainingEnergy`** (p. 531) method and store the return value in a local variable called energyLevel. We get the current time by calling the **`GetTimeNow`** (p. 442) method and store this in a local variable called timeNow. We get the node position of the source node and store a vector with x,y and z co-ordinates in a local variable called locSNode by calling the **`GetNodePosition`** (p. 464) method. We also get the management node position by calling the **`GetNodePosition`** (p. 464) and store the return value in a variable called locMNode. The final value we get is the distance by calling the **`GetDistance`** (p. 465) method and passing the source node and management node location vectors, we store the returned distance in a variable called range. We then push these values to a dynamic data structure called **`recordRelationshipState`** (p. 520).

void CogMan::SwitchProtocol()

The **`SwitchProtocol`** method first checks to see which routing protocol the network is operating on. If the network is operating using the AODV routing protocol the network will switch to the DSDV routing protocol. If the network is operating using the DSDV routing protocol it will switch to the AODV routing protocol.

void CogMan::GrantRequestForChangeChannel(double *backOffTime*)

The `GrantRequestForChangeChannel` method grants the request for each manager node that requested change channel. In addition to granting the request a back off time is sent so that the manager cannot make the same request again until the back off period has elapsed. Each manager that was granted this request will call the `RecordActionOutcome` (p. 517) in order to record if performance has increased or decreased for the node that they are managing at the time specified in backOffTime. We also call the `PrintVectorOfMemoryEntries` (p. 544) once all the requests have been granted for this period. This is for informational purposes only and does not alter the state of the network.

double CogMan::AutonomousControlledNode(int *n*)

The `AutonomousControlledNode` is designed allow a node autonomous movement when not being requested by a manager node and when not currently performing a mission for a manager node. Although we allow autonomy for the controlled nodes we do not want random movements, we want the nodes to assume good mobility in order to better aid the entire network.

We begin by getting the current simulation time by calling the `GetTimeNow` (p. 442) method. Next we determine if the controlled node is moving or not by calling the `IsControlledNodeMoving` (p. 463) method. If the node is not moving the controlled node will determine the best place to move to by applying the following formula:

$$x_{center} = \frac{\sum_{k=1}^{n} x_k}{n}, y_{center} = \frac{\sum_{k=1}^{n} y_k}{n}$$

The controlled node will access the x_k and y_k coordinates of each node k in the data communications network for the last recorded positions. A running total is kept of the x_k and y_k coordinates for each node for this given period. We then get the current controlled nodes' position by calling the `GetControlledNodePosition` () method. A goal position (x_{center}, y_{center}) is set by calculating the center point of the world which is based on the summation of the $x_k$and y_k coordinates. The summation is divided by the number of nodes in the network.

Next we get the distance between the current position (x_c, y_c) and the goal position (x_{center}, y_{center}) by calling the `GetDistance` (p. 465) method. Next we calculate the number of steps that will need to be taken for the given distance.

$$s = \frac{d}{0.8}$$

Now that we have the goal position, the current position and the step count we can calculate each x_{step} and y_{step} by applying the following formulae

$$(x_{step}, y_{step}) = (x_c + \frac{x_{center} - x_c}{N_{steps}}, y_c + \frac{y_{center} - y_c}{N_{steps}})$$

We then move the controlled node by invoking the `MoveControlledNodeTo` (p. 454) method.

double CogMan:: MoveControlledNode (int *mNode*)

The `MoveControlledNode` is designed to select a controllable node for a manager node. We could have a situation where a management node is managing multiple sources, therefore first the management node needs to establish which source node is in most need of the use of a controllable node; the latter is done by calling the **GetSourceNodeForControlledNode** (p. 525) method.

Once we have the source node we need to determine which of the sources destinations is receiving the fewest data packets with the aim to strengthen the link between the source node that is performing worst for this manager and the destination that is performing worst for the selected source. We achieve this by calling the **GetDestinationNodeForSourceNode** (p. 525) and passing the identifier of the source node.

Next we get a controllable node identifier (that is not busy) which is the closest to the midpoint between the source and the destination by calling the **GetClosestControlledNode** (p. 526) and passing the source node identifier and the destination node identifier. Next a vector is created that will store the x,y and z coordinates of the controllable node selected by calling the **GetControlledNodePosition** (p. 527) method.

At this point we have values that represent the source node, the destination node and the current selected controlled nodes position. We used this information to determine the future position at which the controllable node will be at the midpoint between the source node and the destination node, this new position is return and stored in a vector called cNodeNewPosition by calling the **GetControlNodeMidPointXY** (p. 527) method. If the controlled node cannot get to the midpoint within a given time constraint the returned value will be equal to the controlled nodes current position.

Since we now have the future position that the controllable node will need to reach we calculate how long it will take the controllable node to get there by calling the

`GetTimeOfJourneyControlledNode` (p. 464) and pass the current controlled nodes position, the controlled nodes goal position and the speed at which the control node moves.

Next two checks are performed; the first is to ensure that the controllable node will reach its target position within the given time constraint (i.e. a possible constraint is the remaining duration of the simulation). This is the case if the current position of the controllable node is not the same as the position stored in cNodeNewPosition.

The second check is based on predicted positions of the source node, the destination node and time of which it takes for the controllable node to reach its intended new position. We get the predicted position of the source node at the time it would take in order for the controllable node to get to its intended goal by calling the `PredictPostion` (p. 469) method and pass the source node identifier and the time of which the journey takes. We do the same for the destination node. Next we calculate the distance between the source node and the destination node from the predicted positions by calling the `GetDistance` (p. 465) method and pass the predicted source node and destination node positions. Then we use the value returned from `GetDistance` (p. 465) and check that the distance is within the transmission radius by calling the `InRange` (p. 468) method and passing the distance. We do this because we do not want to allow the use of a controllable node if the source and destination are in predicted transmission range before the controllable node gets there.

If the controllable node will get to its intended goal within the given time constraint and if the controllable node will get there before the source node and destination node are in transmission range we allow the manager node control of the node. In order to control the node the `ControlledNodeWalkTo` (p. 457) method is called and passed the control node identifier, the position that the node should move to, the current step size (stride size) and the speed at which the controlled node moves.

The final action performed by this method is to return the time it takes the node to reach its destination to the invoking method.

bool CogMan::GetSourceNodeForControlledNode(int *mNode*)

The `GetSourceNodeForControlledNode` method is used to establish which of the source nodes that this manager is managing is performing the worst. First we set a minimum receiving rate to 9999999 and store this in a variable called minRR. Then for each source node we check if the receive rate is lower than the value in minRR, if the value is lower we update minRR to reflect the receive rate for that source. We also record the source node identifier. When each source node that this manager is responsible for has been checked we return the source node identifier with the poorest receiving rate.

bool CogMan::GetDestinationNodeForSourceNode(int *sNode*)

The `GetDestinationNodeForSourceNode` method is used to establish which of the destination nodes that this source node is receiving the least number of packets. First we set a minimum received number of packets to 9999999 and store this in a variable called minPR.

Next we get the current simulation time and store this in a variable called timeNow by calling the `GetTimeNow` (p. 442) method. We use this later when defining a time interval.

We do not want to check every source → destination pair, only the pairs that are currently transmitting, we do this by only checking the source → destination pairs that are currently transmitting or that are scheduled to be transmitting at this time.

For each of the destinations that are paired with the source node we get the number of received packets for the destination that we are currently checking for a given period. In our case the period is the last five seconds, this is achieved by calling the **GetCurrentReceivedPacketsForDestinationFromSource** (p. 486) and passing the source node identifier, the destination node identifier, the from time and the until time. The from time is five seconds before timeNow and the until time is timeNow. The value returned from this method reflects how many packets the destination has received from the source node for the last five seconds, we assign this value in numPR variable.

Should the value received be less than the minPR value we update the minPR value to the value in numPR, we also record the destination node identifier.

Once we have checked each destination paired with this source node for all the active (or should be active) transmission paths we return the source node identifier. The source node identifier is the source node that has received the fewest packets within the given time period.

int CogMan::GetClosestControlledNode(int *sNode*, int *dNode*)

The **GetClosestControlledNode** method gets the closest available controlled node between the source node and the destination node. The method begins by declaring a variable called chosenCNode and assigning the value -1 and minDist and assigning the value 999999999.

For each controllable node we first check that the node is available (not currently moving) by calling the **IsControlledNodeMoving** (p. 463) method. If the node is available we then get the current controlled node position by calling the **GetControlledNodePosition** (p. 527).

We also get the destination node position and source node position by calling the `GetNodePosition` (p. 464) for each node and passing the appropriate node identifier.

Now that we have the source node position (x_s, y_s) and destination node position (x_d, y_d) we perform the following calculation in order to get the midpoint (x_m, y_m) between the source node and the destination node.

$$(x_m, y_m) = (\frac{x_s + x_d}{2}, \frac{y_s + y_d}{2})$$

Next we get the distance from the controlled node to the midpoint derived from the formula above by calling the `GetDistance` (p. 465) method and passing the two position vectors. If the distance is less than the value in minDist we update minDist to reflect the new minimum distance, we also update chosenCNode to the node identifier.

Once all controllable nodes have been checked we return the node identifier that signifies the closest node to the midpoint between the source node and the destination node. If there are no controllable nodes available we return the value -1.

Vector CogMan::GetControlledNodePosition(int *nodeId*)

The `GetControlledNodePosition` method creates a temporary pointer to a mobility model. We assign the mobility model pointer that is aggregated to the node identifier passed to the temporary pointer. Once we have the pointer to the correct mobility model we invoke the method GetPosition that is part of the ConstantMobilityModel class. We return the vector returned from GetPosition to the method that invoked this method.

Vector CogMan:: GetControlNodeMidPointXY (int *sNode*, int *dNode*, Vector *cNode*)

The `GetControlNodeMidPointXY` method will predict if the controlled node is able to reach the midpoint between a source node and a destination node within a given time constraint.

If the controlled node can reach the midpoint within the time constraint the new coordinates are returned that reflect the position of the midpoint for the time that the midpoint was reached.

We first get the current simulation time by calling the **`GetTimeNow`** (p. 442) method. We declare a futureTime variable and initialize it to the current simulation time. We set the constraint time (the time by which the controlled node must have reached the midpoint of source and destination node) by creating a variable called journeyTime and setting this to the remaining simulation time.

While we have not established the time at which the midpoint will be reached we get the predicted position of the source node by calling the **`PredictPostion`** (p. 469) method. We call the same method to get the predicted position of the destination node.

We calculate the midpoint (x_m, y_m) between the source node (x_s, y_s) and the destination node (x_d, y_d) by applying the following formula:

$$(x_m, y_m) = (\frac{x_s + x_d}{2}, \frac{y_s + y_d}{2})$$

We then get the time of which it would take the controllable node to get to the midpoint derived from the above formula by calling the **`GetTimeOfJourneyControlledNode`** (p. 464) method and pass the controlled nodes current location (the position of the controllable node is passed into a Vector called cNode), the midpoint position and the controlled node speed. We store the time in a variable called journeyTime.

If the journeyTime plus the current time is less than the future time we know that the controlled node can get to the midpoint and we update the controlled nodes position to reflect the position of when it reached the midpoint.

If the journeyTime plus the current time is greater than the future time, we increment the future time and repeat the process. We keep incrementing future time and repeating the process until the midpoint can be reached, or until we have established that the controlled node cannot reach the midpoint within the given time constraint.

The controlled nodes vector is returned to the invoking function. If the controlled nodes coordinates have been updated it means the controllable node will reach the midpoint. Otherwise the controlled nodes coordinates will not be updated.

Energy Model

void CogMan::SetBatteries()

The `SetBatteries` method creates an energy source container with an initial value. The source container is populated with one or more energy sources (in our case one energy source). The energy sources within the energy container are then aggregated to a node.

Each node has its own energy source container with one or more energy sources in that container aggregated to that node. The energy source container does not contain energy sources that are used by more than one node.

To keep track of the energy sources that are aggregated to the nodes we populate an aodvEnergySources container that holds the addresses of all the energy sources.

A new container is created called aodvDeviceEnergyModel that allows us to use an energy model; in this case we use the wifi radio energy model and set the TxCurrentA to a double value of 0.0174. This energy model is then aggregated to the aodvDevices and the aodvEnergySources and holds the memory addresses for all the energy sources that are using this model.

The method also creates trace sinks that trigger a method when an event happens. In the case when the energy changes it will call the **RemainingEnergy** (p. 530) method and the **TotalEnergy** (p. 530) method.

void RemainingEnergy (double *oldValue*, double *remainingEnergy*)

This method prints to screen the remaining energy of a given node. The given node is defined in a callback method and is called when an event occurs, for example when a packet is sent.

void TotalEnergy (double *oldValue*, double *totalEnergy*)

This method prints the total energy consumed by a given node. The given node is defined in a callback method and is called when an event occurs, for example when a packet is sent.

double CogMan::GetPredictedTimePowerUntil(int *nodeId*)

The **GetPredictedTimePowerUntil** method will accept a node ID and return a predicted time of how long the energy source(s) attached to that node will last.

The method begins by calling the **GetRemainingEnergy** (p. 531) method and storing the current remaining energy in a local variable called remainingEnergy. We then get the current simulator time by calling the **GetTimeNow** (p. 442) method and store this in a variable called elapsedTime. We create a third local variable called usedEnergy, we set this to a value derived by subtracting the remaining energy value from the initial energy value. We then calculate how much energy the node is using per second by dividing the used energy value by the elapsed time, the calculated value is stored in energyUsedPerSecond. Finally, we calculate the predicted remaining time for this energy source (how long the nodes battery will last) by dividing the remaining energy by the energy used per second. This gives us an approximation in seconds of how long the node will be able to function.

We call three functions which are `GetHoursFromSeconds` (p. 442), `GetMinutesFromSeconds` (p. 442) and `GetSecondsFromSeconds` (p. 443) so that we can display on screen the time before predicted power depletion of the nodes energy source(s) in a time format, i.e. hours, minutes and seconds.

The function returns the total number of seconds that the node has before power depletion to the invoking instruction.

double CogMan::GetRemainingEnergy(int *nodeId*)

The `GetRemainingEnergy` method accepts a node identifier that is used to get the energy source(s) associated with that node. Once the energy source(s) are located for the node the `GetRemainingEnergy (p. 531)` method for that class is called to get the remaining energy. This value is returned to the invoking instruction.

In order to do this we first create two iterators, we point the first iterator to the start of the sources container (a container that holds the pointers to the energy sources) and call this sourceBegin. The second iterator we point to the end of the same container.

We loop though all the pointers in the source container in order to locate the correct energy source pointer for the node identifier that was passed to the parameter list for this method. To do this we create a pointer to an energy source object and call this myEnergySource. We assign the value of the pointer that is pointed to by the iterator sourceBegin. We then create a pointer to a node object and copy the pointer for the node that is using the energy source pointer just obtained. This gives us the node pointer for the current energy source being checked. We then use the node pointer and invoke a method that gives us the identifier assigned to that node pointer. We compare this against the node identifier that was passed to the method. If there is a match we call the `GetRemainingEnergy` (p. 531) method that is part of the energy source class and return the value to the method that invoked this method. If the was not a match we add one to the sourceBegin iterator so that it points to the next pointer in the energy source container. We repeat this process until a match is found.

void CogMan::SetRemainingEnergy(int *nodeId*)

The `SetRemainingEnergy` method is used to set the remaining energy equal to that of its paired node in the other container. This method is called when we switch protocol to ensure that the energy levels of the corresponding nodes are operating at the same level.

The method begins by comparing the current protocol against a string value. If the protocol is equal to AODV then the protocol has just switched from DSDV, therefore we set the node energy that is operating on the AODV protocol equal to the same node identifier from the DSDV container. If the current protocol is DSDV then we have just switched from the AODV routing protocol. In this case we set the current energy level of the DSDV node to that of its corresponding node from the AODV container.

void CogMan::SetAodvRemainingEnergy(int *nodeId*)

This method will set the remaining energy for a node in the AODV node container to the same value with the same node identifier of the DSDV container.

It works by creating a pointer to an energy source that is attached to the AODV node id that is passed to the method. It also creates a pointer to an energy source that is attached to the corresponding node in the DSDV container.

We get the value that represents the remaining energy of the energy source attached to the DSDV node and use this value to set the remaining energy of energy source for the AODV node.

This method is not called directly but from the **`SetRemainingEnergy`** method.

void CogMan::SetDsdvRemainingEnergy(int *nodeId*)

This method will set the remaining energy for a node in the DSDV node container to the same value with the same node identifier of the AODV container.

It works by creating a pointer to an energy source that is attached to the DSDV node id that is passed to the method. It also creates a pointer to an energy source that is attached to the corresponding node in the AODV container.

We get the value that represents the remaining energy of the energy source attached to the AODV node and use this value to set the remaining energy of energy source for the DSDV node.

This method is not called directly but from the **`SetRemainingEnergy`** method.

bool CogMan::WillBeAlive(int *nodeId*, double *atTime*)

The **`WillBeAlive`** (p. 533) method returns true if the current energy source for the given node identifier that was passed is predicted to have enough energy for transmission/reception at the next point that the **`management`** (497) method is called. The method begins by getting the remaining energy of the energy source that is attached to a node (based on the node identifier that was passed) by calling the **`GetRemainingEnergy`** (p. 531) method. The **`GetRemainingEnergy`** (p. 531) method returns a double value which represents the current energy left for this node; we store this value in remainingEnergy. Then we create a variable called elapsedTime and store the current time by calling the

GetTimeNow (p. 442) method. We calculate how much energy has been used by subtracting the remainingEnergy value from the initialEnergy value. We then calculate energy consumption per second by dividing usedEnergy by the elapsedTime. To get how much power is required until the next call to the **Management** (p. 497) method we multiple the energy consumption per second by the time remaining until the next call. We finally return remainingEnergy > powerNeeded which will result in the return value of true if the remaining energy is greater than the power that is required. Otherwise false value will be returned.

void CogMan::RemoveDepleatedNodes()

The **RemoveDepleatedNodes** method creates two node containers called deadNodes and liveNodes. We then check the remaining energy level of each node in turn by calling the **GetRemainingEnergy** (p. 531) method. If the node currently being checked has not depleted its energy source we copy the pointer of that node into the liveNodes container. Otherwise we copy the pointer of that node into the deadNodes container.

double CogMan::GetChannel(int *nodeId*)

The GetChannel method accepts a node identifier and returns the channel that the node is transmitting on.

void CogMan:: SetChannel(int *mNode*, int *channelNumber*)

The **SetChannel** method will set the channel of the node identifier that has been passed to that of the channel number that has been passed. In order to set the channel number we first need to get the pointer to the physical layer, once we have this we can obtain the pointer to the wifi physical layer. Using the pointer to the wifi physical layer we get the pointer to the yans wifi physical layer. Once we have the yans wifi physical layer pointer we are able to set the channel number by invoking the SetChannelNumber method which is a member function for the Yans Wifi Class.

void CogMan::SetChannelControlledNodes(int *mNode*, int *channelNumber*)

The **SetChannelControlledNodes** method works in the same way as the SetChannel (p. 534) method with the exception that the node identifier that is passed accesses the controlled node container rather than the node container.

bool CogMan::WillBeInRange(int *sNode*, int *mNode*, double *atTime*)

The **WillBeInRange** method returns true if two nodes will be in predicted range of each other at some time in the future. First the method creates two vectors one called sNodePredictedPosition and the other called mNodePredictedPosition. Next we called the **PredictPosition** (p. 469) method for the first node identifier that was passed to the WillBeInRange method with the time that we want the predicted position for. The **PredictPostion** (p. 469) method returns a vector that contains of x, y and z co-ordinates which is copied into sNodePredictedPosition. We do the same for the second node identifier that was passed and store the return value from **PredictPostion** (p. 469) into mNodePredictedPosition.

Once we have the predicted positions of the two nodes at a given time in the future we can obtain the predicted distance between them by calling the **GetDistance** (p. 465) method and passing the predicted positions of the source node and the management node, the return value from **GetDistance** (p. 465) is stored in a local variable called distance. Finally we call the **InRange** (p. 468) method and pass the distance. The **InRange** (p. 468) method will return true if the nodes will be in range of each other (based on distance passed) or false otherwise – this return value is then returned to the method that invoked this one.

void CogMan::TestRadius()

The **`TestRadius`** method allows us to test the configured transmission range. It creates two nodes by calling the **`CreateAodvNodes`** (p. 443) method and one traffic flow between them by calling the **`CreateTrafficFlows`** (p. 474) method. Next we position the two nodes at the same location by calling the **`MoveNodeTo`** (p. 454) method. Next we create a conditional for loop in order to schedule one of the nodes moving away each second. We know the transmission range because each second a packet is sent. Once no more packets are received we know the transmission range by observing the output to the terminal.

void CogMan::Set21mRadius()

The **`Set21mRadius`** method reconfigures the characterises of the antenna so that the transmission radius is up to 21 meters. We set the energy detection threshold to -61.76, the CcaMode1Threshold to -61.76, the Tx Gain to 4.5, the RxGain to 4.5, the Tx Power Levels to 1, the Tx Power End to 16, the Tx Power Start to 16, the RxNoiseFigure to 4 and myRadius to 21 so that the **`InRange`** (p. 468) method works according to the new configuration.

void CogMan::Set100mRadius()

The **`Set21mRadius`** method reconfigures the characterises of the antenna so that the transmission radius is up to 21 meters. We set the energy detection threshold to -61.76, the CcaMode1Threshold to -61.76, the Tx Gain to 14.5, the RxGain to 14.5, the Tx Power Levels to 1, the Tx Power End to 16, the Tx Power Start to 16, the RxNoiseFigure to 4 and myRadius to 100 so that the **`InRange`** (p. 468) method works according to the new configuration.

Supporting Methods

void CogMan::ConfigureNetAnim()

The ConfigureNetAnim method allows us to change certain parameters for Network Animator. We use this method primarily in assigning a different colour for each group, for example we colour our Eagles group blue and our Sharks group green. We also use this method to differentiate between AODV nodes, DSDV nodes and Controllable nodes.

void CogMan::SortVector()
Algorithm used to sort a vector.

uint32_t CogMan::GetLengthOfInteger(uint32_t *lengthOfThis*)
This is used for formatting output to screen. It accepts an integer, converts to a ostringstream object, converts to string, counts the number of characters and then then returns the number of characters.

string CogMan::SpacesAndTail(uint32_t *noOfSpaces*)
The `SpacesAndTail` method returns a string with an appropriate number of spaces. This method was created entirely for a cosmetic reason and allowed us the ability to apply padding to strings of output. Which allowed us to view data in tabular format.

Ipv4Address CogMan::GetIPAddressFromNodeId(int *nodeId*)
The `GetIPAddressFromNodeId` method takes a node identifier and returns the IP address for that node. First we create a pointer to a node memory address and assign the pointer with the address for the node identifier for the current protocol that is in use. Next we create a pointer for an Ipv4 class object and assign the pointer to the object that is aggregated to this node. Using this pointer we create a type Ipv4InterfaceAddress and assign this the pointer of the address of the node. Once we have the pointer to the Ipv4InterfaceAddress we can obtain the local IP address for this node. This is returned to the invoking function.

string CogMan::ConvertIPAddressToString (Ipv4Address *addr*)
The `ConvertIPAddressToString` method converts the IPv4Address object to a string object. We first convert the value stored in the IPv4Address object to a stream object by creating a temporary stream object called convert. We then use the << operator to transfer

the value of the IPv4Address object to a steam object. Once this has been achieved we convert the stream object to a string object by calling the str method of the String class.

The converted IP address is then returned to the method that invoked this method.

Output Data Files

void CogMan::DoStats()

The `DoStats` method is called when the simulation ends. The method outputs to screen the total statistics for each routing protocol. The outputs for AODV are the total payload packets sent, the total payload data dropped, the total control data sent, the total control data dropped and the total payload data received. The same information is printed for the Dsdv protocol.

void CogMan::WriteVectorOfManagementNodeMemoryToDisk (string *theFilename*)

The `WriteVectorOfManagementNodeMemoryToDisk` method outputs the managementNodeMemory (p.) vector to disk in comma separated file format.

void CogMan::WriteVectorOfNodePositionsToDisk (string *theFilename*)

The `WriteVectorOfNodePositionsToDisk` method outputs the myPositions (p.) vector to disk in comma separated file format.

void CogMan::WriteVectorOfMemorySystemStateToDisk (string *theFilename*)

The `WriteVectorOfMemorySystemStateToDisk` method outputs the systemState (p.) vector to disk in comma separated file format.

void CogMan::WriteVectorOfAllPacketsToDisk (string *theFilename*)

The `WriteVectorOfAllPacketsToDisk` method outputs the allPackets (p.) vector to disk in comma separated file format.

void CogMan::WriteVectorOfReceivedPacketsToDisk (string *theFilename*)

The `WriteVectorOfReceivedPacketsToDisk` method outputs the packetReceived (p.) vector to disk in comma separated file format.

void CogMan::WriteVectorOfDroppedPacketsToDisk (string *theFilename*)

The `WriteVectorOfReceivedPacketsToDisk` method outputs the packetDropped (p.) vector to disk in comma separated file format.

void CogMan:: WriteVectorOfSentPacketsToDisk (string *theFilename*)

The `WriteVectorOfSentPacketsToDisk` method outputs the packetSent (p.) vector to disk in comma separated file format.

void CogMan::WriteVectorOfSentPacketsBCPToDisk (string *theFilename*)

The `WriteVectorOfSentPacketsBCPToDisk` method outputs the packetSentByCurrentProtocol (p.) vector to disk in comma separated file format.

void CogMan::WriteVectorFlowMonitorToDisk (string *theFilename*)

The vector that is created in the `CreateVectorFlowMonitorInformation` (p. 545) method is written to disk as a comma separated text file.

void CogMan::WriteAODVvsDSDVStats (double *interval*)

The `WriteAODVvsDSDVStats` method allows us to compare how the network has performed using the AODV routing protocol verses the DSDV routing protocol and in comparison to the dynamic switching or the routing protocol.

The method begins be creating a pointer to the allPackets vector, a pointer to the packetSentByCurrentProtocol vector, a pointer to the packetReceivedByCurrentProtocol vector and a pointer to the packetDroppedByCurrentProtocol. All pointers are set to the beginning of the relevant vectors.

Next we initialising counters that will keep count of the amount of packets that have been sent, received and dropped for each routing protocol (for within a given interval, i.e. 2 seconds). We also initialise variables to keep count of how many times the switching of a protocol had a positive and a negative effect on the performance of the network.

We start at the beginning of the allPackets vector and work our way packet entry by packet entry until we reach the end of the vector. Each packet entry that exists is checked to establish which routing protocol is in operation (for that packet) and the reason why the packet was recorded (i.e. it could have been sent, received or dropped). We increment the relevant counter for each packet, for example, if the routing protocol in operation was DSDV for that packet and the packet is recorded as being dropped, we increment a counter labelled `dsdvD`. Thus for each packet we update one of the following counters:

```
aodvS routing protocol is aodv, packet is recorded as being sent.
aodvR routing protocol is aodv, packet is recorded as being received.
aodvD routing protocol is aodv, packet is recorded as being dropped.
```

We have similar counters for the dsdv routing protocol: dsdvS, dsdvR, dsdvD.

This method accepts an interval into the parameter list. This allows us to group statistics into manageable chunks of time (i.e. every 2 seconds). Therefore as we step through the allPackets vector we pause at each interval, perform some analysis and then write the results to a data file.

Three new counters are created (for each interval). We then loop through the vector packetSentByCurrentProtocol (for the same interval, i.e. the first interval could be 0-2 seconds). We count how many packets were sent by the currently operating protocol for this interval. We do the same for received packets and dropped packets.

Next we compare how many packets were received by the AODV routing protocol for this period against the DSDV routing protocol for the same period. We increment a counter called goodA if the AODV routing protocol received more packets for this period. If the DSDV routing protocol received more packets for this period we increment a counter called goodD. If both routing protocols received the same number of packets for this period we increment a variable labelled bad. We then write this information to file as depicted in Table 12- Packets 65 - 85 seconds which is a snapshot of this information from 65 seconds to 85 seconds with an interval set to 5 seconds (with four traffic flows and 80 nodes).

time	AODV S	AODV R	AODV D	DSDV S	DSDV R	DSDV D	Swi S	Swi R	Swi D
65	2797	380	456	3196	1078	758	1747	380	574
70	3127	2403	436	4366	1913	754	4318	1895	572
75	3977	3745	978	4363	1893	365	3977	3745	374
80	3765	3080	875	4228	1809	487	3764	3080	752
85	4038	742	387	4345	1864	455	4225	1490	356

Table 12- Packets 65 - 85 seconds

After each interval we increment our interval based on the parameter that was passed to this method, update our running totals for the number of received packets for DSDV, AODV and dynamic switching. We also reset our dsdvS, dsdvR, dsdvD, aodvS, aodvR, aodvD counters for the next interval.

The method concludes by outputting to the terminal an overview based on the running totals. If the total number of packets received by dynamically switching protocols is higher than the running totals for DSDV and AODV for this simulation run know that dynamically switching routing protocol outperformed both AODV and DSDV and network performance increased. If the total number of received packets by the dynamically switching of routing protocols is equal to or greater than both the DSDV or the AODV total we know that switching has performed at the same rate as the better performing protocol. If the number of received packets is less than the better performing protocol but greater than the least performing protocol we know that we are doing well, but we could be doing better.

If the dynamic switching protocol is not performing as well as either the DSDV routing protocol or the AODV routing protocol we know that we the management node has in this case made a bad decision in its course of action.

Runtime Information

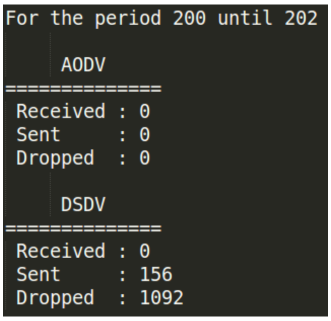
int CogMan::PrintAllPacketStats(double *fromTime*, double *untilTime*)

The `PrintAllPacketStats` method accepts a time period that we want to check all packet statistics for. The time period is determined by specifying a start time (fromTime) and a finish time (untilTime). We initialize an iterator called vectorIndex and set this to the end of the data structure called allPackets. Next we initialize to zero counters to record how many packets were received, sent and dropped by the routing protocols in use; these are dsdvReceived, dsdvSent, dsdvDropped, aodvReceived, aodvSent and aodvDropped.

We begin looping through the allPackets data structure and continue this loop until we have reached the beginning of the data structure or until the loop is broken (explained subsequently).

Next we create an inner loop to loop through the data structure until we reach a finish time specified by untilTime – we are not interested in packets that were received after the untilTime. When this period is reached the inner loop completes but we are still in the outer loop.

We then check if the time recorded in the data structure is before the fromTime passed to the parameter list. If it is we break out of the outer loop and display the statistics to screen for that period, this output is depicted in Figure 55 - Packet Statistics for a given period in time.



```
For the period 200 until 202

     AODV
==============
 Received : 0
 Sent     : 0
 Dropped  : 0

     DSDV
==============
 Received : 0
 Sent     : 156
 Dropped  : 1092
```

Figure 55 - Packet Statistics for a given period in time

Each packet that is checked within the given time period could have been recorded by any of the networks configured, for example a network that causes congestion, the network operating on the DSDV routing protocol or the network operating on the AODV routing protocol. We are only interested in data packets that are using either DSDV or the AODV routing protocol (we have already filtered out the control packets). Therefore we get the protocol that the packet currently being checked by calling the **`GetProtocol`** method and pass the destination IP address for the packet being checked. The **`GetProtocol`** method returns a string value of either "aodv" or "dsdv". Each packet that is recorded is flagged with 'S', 'D', or 'R', which allows us to differentiate between sent, dropped and received packets. If the protocol returned is aodv and the flag is set to 'S' we increment aodvSent by one, if the flag is set to 'D' we increment aodvDropped by one. If the flag is set to 'R' we increment aodvReceived by one. We do the same for the dsdv protocol and update dsdvSent, dsdvReceived or dsdvDropped using the flag method just described.

void CogMan::PrintRemainingEnergy(int *nodeId*)

This method will print the screen the remaining energy of a given energy source that is attached to the node id that is passed.

void CogMan::Print(string *theOutput*, int *type*)

Prints a log message to screen or not depending if the type of message has logging enabled. The logging for the type of message is set in the SetLogging method.

void CogMan::PrintVectorOfDistances(vector<nodeDistance> *theDistances*)

Takes a vector of nodeDistance (currently we have myNeighbours and nodesInMyWorld) and prints it to screen.

void CogMan::PrintAllChannels()

The **`PrintAllChannels`** method loops through all the available node containers and prints the operating channel for each node within the container. This method is for informational purposes only and does not make any changes.

void CogMan::PrintVectorOfMemoryEntries()

The **`PrintVectorOfMemoryEntries`** method prints to screen all the current memory entries for each and every management node. The information printed is the time of the request, which source node the request was on behalf of, the management node identifier making the request, the drop rate at the time the request was made, the receive rate at the time the request was made, the send rate at the time the request was made, current density at the time the request was made, the routing protocol in use at the time of the request, the request that was made, the progress of the request (i.e. waiting acknowledgement, acknowledge, permitted or denied), the back off time for making the same request and a performance metric indicating if performance has been checked since the action and if it has the normalized value indicating the performance based on the action. This method is used for information to ensure the management nodes memory is working correctly.

void CogMan::PrintNodePosition(int *nodeId*)

Takes a nodeId and gets the current position of that node. This information is printed to screen.

void CogMan::PrintVectorOfNodePositions(vector<currentPosition> *theDistances*, string *theFilename*)

This method takes any vector of type currentPosition and a filename. Each row in the vector is then printed to screen. Each element in the row is separated with a comma.

void CogMan::CreateVectorFlowMonitorInformation(Ptr<FlowMonitor> *monitor*)

This method accepts a pointer to a FlowMonitor class that has already been defined. It read the vector at the passed address which is created for flowmonitor stats and writes them to a new vector called myDataFlows which is of type flowInformation.

void CogMan::PrintFlowMonitorInformation(Ptr<FlowMonitor> *monitor*)

This method is similar to `CreateVectorFlowMonitorInformation`(Ptr<FlowMonitor> monitor) except that the information is printed to screen rather than written to disk.

void CogMan::PrintNodeInformation(Ptr<Node> *node*)

The `PrintNodeInformation` was mainly used during our testing phase and allowed us the ability to output information to the terminal for the node pointer passed the following: the global node id, the number of applications, the number of network devices, the number of network interfaces and the local IP address for each interface.

Data structures

Structure nodeDistance

Reference	Data Type	Description
sourceNode	int	The source node identifier
neighbourNode	int	The neighbour node identifier
distance	double	The current range in meters between the source node and the destination node
myTime	double	The time the entry was added

Vector myNeighbours

The `myNeighbours` vector is used to store the neighbours that are in communications range of a given source node. The structure is used by the `CheckNeighbours` (p. 472) method.

Vector nodesInMyWorld

The myNeighbours vector is used to store all the neighbours of a given source node regardless of if they are in communications range or not. The structure is used by the `GetAllDistancesFromSource` (p. 465) method.

Vector allNodesTheWorld

The allNodesTheWorld vector is used to store all the neighbours of each and every source node regardless of if they are in communications range or not. The structure is used by the `CreateVectorOfDistancesAllNodes` (p. 466) method.

Structure flowInformation

Reference	Data Type	Description
flowId	int	The flow identifier
sourceAddress	Ipv4Address	The source IP address
destinationAddress	Ipv4Address	The destination IP address
txPackets	int	The total number of packets transmitted for this flowId
txBytes	int	The total number of bytes transmitted for this flowId
rxOffered	double	The total number of bytes offered (received)
rxPackets	int	The total number of received packets
rxBytes	int	The total number of bytes received for this flowId
throughput	double	The throughput for this flowId

Vector myDataFlows

The myDataFlows vector is used to store information about each traffic flow in the data communications network. The structure is used by the `CreateVectorFlowMonitorInformation` (p. 545) method and the `WriteVectorFlowMonitorToDisk` (p. 539) method.

Structure currentPosition

Reference	Data Type	Description
nodeId	int	The node identifier
time	int	The time the position was recorded
x	double	The x coordinate for the node identifier
y	double	The y coordinate for the node identifier

Implementation

This data structure is used to store the positions of the nodes in the network simulation at various intervals. The positional information is used when making predictions.

Vector myPositions

The myPositions vector is used to store the positions of the nodes in the simulation. We record the information using the `RecordPosition` (p. 472) method. We can print the vector of positions by calling the `PrintVectorOfNodePositions` (p. 544) method. The primary use for this vector is to allow us to predict the next position a node may be at, we do this by calling the `PredictPosition` (p. 469) method which is used for the election and by manager nodes when checking relationship states.

The vector is also used for our controlled nodes when they assume autonomous control (i.e. when they are not performing a mission for a manager) and is used in the `AutonomousControlledNode` (p. 521) method.

At the end of the simulation run the historic position information can be written to a csv file by making use of the `WriteVectorOfNodePositionsToDisk` (p. 538) method.

Structure managerNode

Reference	Data Type	Description
sourceNodeId	int	The source node ID
managerNodeId	int	The candidate/management node ID
timeElected	double	The time the candidate/manager was recorded
timeInRangeUntil	double	The time the source will be in range for
timePowerUntil	double	The time the candidate/manager will have power for
density	double	The density of the candidate/manager node

Implementation

This data structure is used to store attribute values that relate to the candidate and election process.

Vector candidatesList

The candidatesList vector is used to record suitable candidates that could potentially become a manager of a source node. We record candidates to this data structure using the `RecordElectionCandidate` (p. 493) method. We also access the vector in the during the election process. The election process uses the **`Election`** (p.492) method to check if there are candidates available to be elected. The vector is also used in the **`SelectManagementNode`** (p. 494) method when we choose the most suitable candidate.

Vector candidatesListPercents

The candidatesListPercents stores the same information as the candidatesList, however, rather than storing absolute values it stores the values as a percentage. The values converted to percentage are timeInPowerUntil, timeInRangeUntil and density. This vector is used by the **`RecordElectionCandidatePercents`** (p. 494) method.

Vector managersList

The managersList vector is used to record the managers for each source node. We record each manager for each source. We do not delete any entries from the vector even if a manager stops managing a particular source node. The method **`RecordElectionWinner`** (p. 497) records the information to the vector.

Structure IpAddresses

Reference	Data Type	Description
source	ns3::Ipv4Address	Store a source IP Address
destination	ns3::Ipv4Address	Store a destination IP Address
myTime	double	Store a time value

Implementation

This data structure is used to store packets that are sent, received and dropped within the network. We create several vectors based on this data structure which are explained next.

Vector packetSent

The packetSent data vector stores a packet that is sent within the network if the packet is not a route discovery or other network control traffic. The information recorded is the source IP address the destination IP address and the time at which the packet was sent. This information is recorded in the **`RecordSentPacket`** (p. 482) method. The packetSent vector is also used in the **WriteVectorOfSentPacketsToDisk** (p. 539) that writes the vector to secondary storage when the simulation finishes.

Vector packetSentByCurrentProtocol

The **packetSentByCurrentProtocol** records the same information as packetSent; however, it will only record the packet information if the packet has been sent by the routing protocol that is currently in use. This information is recorded in the **RecordSentPacket** (p.482) method. The **packetSentByCurrentProtocol** vector is also used by the **WriteVectorOfSentPacketsBCPToDisk** (p. 539) which writes the vector to secondary storage when the simulation finishes. We also compare performance by comparing the use of the AODV and the DSDV protocol by making use of the **WriteAODVvsDSDVStats** (p. 539) method. The data structure is also used by the **GetCurrentSentPacketsForSource** (p. 488) method which allows us to calculate how many packets a given source has sent within a given period.

Vector packetReceived

We record every packet received by calling the **RecordReceivedPacket** (p. 480) method. The vector packetReceived is also accessed by the **WriteVectorOfReceivedPacketsToDisk** (p. 539) method when writing all packets that have been received to a CSV file for analysis when the simulation completes.

Vector packetReceivedByCurrentProtocol

We make use of the packetReceivedByCurrentProtocol data structure when a packet is received and the **RecordReceivedPacket** (p. 480) method is called. The data structure is accessed by the **GetCurrentReceivedPacketsForSource** (p. 487) method and the GetCurrentReceivedPacketsForDestinationFromSource (p. 486) method.

We write received packet statistics to disk by calling the **`WriteAODVvsDSDVStats`** (p. 539) method, this method allows us to quickly compare performance with the AODV and DSDV routing protocol.

Vector packetDropped

We record every packet that is dropped by calling the **`RecordDroppedPacket`** (p. 485) method. The vector packetDropped is also accessed by the **`WriteVectorOfDroppedPacketsToDisk`** (p. 539) method when writing all packets that have been dropped to a CSV file for analysis when the simulation completes.

Vector packetDroppedByCurrentProtocol

We make use of the packetReceivedByCurrentProtocol data structure when a packet is dropped and the **`RecordDroppedPacket`** (p. 485) method is called. The data structure is also accessed by the **`GetCurrentDroppedPacketsForSource`** (p. 487) method and the GetCurrentReceivedPacketsForDestinationFromSource (p. 486) method.

We write dropped packet statistics to disk by calling the **`WriteAODVvsDSDVStats`** (p. 539) method, this method allows us to quickly compare performance with the AODV and DSDV routing protocol.

Structure AllPackets

Reference	Data Type	Description
source	ns3::Ipv4Address	Store a source IP Address
destination	ns3::Ipv4Address	Store a destination IP Address
myTime	double	Store a time value
action	char	Store a char to indicate if the packet was sent, received or dropped

Implementation

This data structure is used to store packets that are sent, received and dropped within the network. We create a single vector based on this data structure which collates all packets.

Vector allPackets (check data structure)

The allPackets data structure will stores each packet that is sent, received and dropped within the network for the duration of the network run with a flag to indicate the type of packet being recorded. If the packet recorded has been sent we record the character 'S', if the packet is received we record the character 'R'. To identify the dropped packet we record the character 'D'. This information is recorded in the **`RecordSentPacket`** (p. 482) method, **`RecordReceivedPacket`** (p. 480) method and the **`RecordDroppedPacket`** (p. 485) method. The allPackets data vector is also access by the **`PrintAllPacketStats`** (p. 542) method which is used to produce a summation of the packet statistics and print this to the terminal.

We write dropped packet statistics to disk by calling the **`WriteAODVvsDSDVStats`** (p. 539) method, this method allows us to quickly compare performance with the AODV and DSDV routing protocol.

The final method that accesses the allPackets vector is the **`WriteVectorOfAllPacketsToDisk`** (p. 538) method which creates a csv file for our analysis.

Structure controlledNode

Reference	Data Type	Description
nodeId	int	Stores a node identifier

Implementation

This data structure is used to store node identifiers that we wish to group.

Vector AodvEaglesNode

Vector AodvSharksNode

Vector DsdvEaglesNode

Vector DsdvSharksNode

The **`AodvEaglesNode`**, **`AodvSharksNode`**, **`DsdvEaglesNode`** and the **`DsdvSharksNode`** vectors stores the node identifiers of the nodes that join a particular group. The vectors are populated and used in a similar fashion so we explain them together. The information is recorded to the appropriate vector by calling the **`SplitNodesTwoEvenGroups`** (p. 451) method and the **`PlaceNodesTwoGroups`** (p. 451) method. We access the data structure when we define mobility patterns in the **`MoveNodeDiamondPattern`** (p. 460) method and the **`LakeScenario`** (p. 461) method.

We also call on the data structure in the **`ConfigureNetAnim`** (p. 536) method when we change the configuration for the representation of the node in Network Animator.

Structure nodeMoving

Reference	Data Type	Description
nodeId	int	Store a node identifier
timeStart	double	Stores the time a node will begin to move
timeStop	double	Stores the time when a node will finish its journey

Implementation

This data structure is used to keep track of nodes that are scheduled to move. We do this so that we do not assign multiple movements to the same node at the same time.

Vector setNodeMove

The setNodeMove vector is populated each time a node is assigned mobility by calling the **`RecordMovement`** (p. 466) method.

The setNodeMove vector is access by the **`IsNodeMoving`** (p. 462) method to check if the node is currently moving or not.

Vector setControlledNodeMove

The setControlledNodeMove vector is populated each time a node is assigned mobility by calling the **`RecordControlledNodeMovement`** (p. 466) method.

The setNodeMove vector is access by the **`IsControlledNodeMoving`** (p. 463) method to check if the node is currently moving or not.

Structure memorySystemState

Reference	Data Type	Description
sNode	int	The source node Id
mNode	Int	The management node Id
time	double	Store a time value
density	double	The density for the management node
protocol	string	The routing protocol being used
droppedPacketsAodv	int	The total number of AODV packets dropped
droppedPacketsDsdv	int	The total number of DSDV packets dropped
receivedPacketsAodv	int	The total number of AODV packets received
receivedPacketsDsdv	int	The total number of DSDV packets received
channel	int	The channel the nodes are communicating on
action	string	Any action performed by the management node

Implementation

The memorySystemState structure stores information relating to the state of the network at a given moment time. The state refers to the current source node(s), destination node(s), management node(s), drop rate for the source node's packets, the receive rate at the destination node, the density, the channel and routing protocol currently being used to transmit and any action taken by the management node.

Vector systemState

The systemState vector is populated by calling the `RecordSystemState` (p. 507) method. The data structure is accessed by the `GetCurrentDropRate` (p. 490) method and the `GetCurrentReceiveRate` (p. 491) method.

Structure cognitivity

Reference	Data Type	Description
atTime	double	Store a time value
sNode	int	The source node Id
mNode	int	The management node Id
dropRate	double	The drop rate for the source node
receiveRate	double	The receive rate for the destination node
sendRate	double	The send rate for the source node
density	double	The density for the management node
channel	int	The channel the nodes are communicating on
routingProtocol	string	The routing protocol being used
request	char	An action requested by the management node, Switch **P**rotocol, Change **C**hannel, **M**ove Node or **U**ndefined.
requestProgress	char	Current progress of the request (**A**cknowledged, **D**enied, **P**ermitted)
backOffTime	double	Time at which a similar request can be made. If requestProgress is P also the time to check performance
performance	int	Performance indicator based on action

Implementation

This data structure is used to store attribute values that relate to cognition. We create one vector based on this structure.

Vector managementNodeMemory

The `managementNodeMemory` vector stores information relating to the state of the network at a given time. The state refers to the current source node(s), destination node(s), management node(s), drop rate for the source node's packets, the receive rate at the destination node, the density, the channel and routing protocol currently being used to transmit and any action taken by the management node.

The information is recorded each time the **Management** (p. 497) method is called (determined by a checking time interval variable called checkingIntervalTime).

During the management process the management node will access this vector when the **GetAction** (p. 512) method, the **RequestAction** (p. 507) method, the **RecordActionRequest** (p. 507) method and the **GetSourceNodeForControlledNode** (p. 525) method.

The Network Control Center will update the contents on the vector when authorising or denying requests. The **DenyAllRequestForControlledNode** (p. 516) method, **DenyARequestForControlledNode** (p. 516) method, **GrantRequestForControlledNode** (p. 519) method, **GrantRequestForSwitchProtocol** (p. 517) method, **GrantRequestForChangeChannel** (p. 521) method make use of the requestProgress attribute for a given instance of the vector.

The managementNodeMemory vector is also used by the **RecordActionOutcome** (517) method when updated performance metrics are recorded based on a prior action.

During the testing period of our simulations we make use of the **PrintVectorOfMemoryEntries** (p. 544) method to ensure that information is being recored to the vector correctly.

When the simulation finishes the **WriteVectorManagementNodeMemoryToDisk** (p. 538) method allows us to write the vector to secondary storage in CSV file format.

Structure request

Reference	Data Type	Description
atTime	double	Store a time value
mNode	int	The management node Id

request	char	An action requested by the management node, Switch **P**rotocol, Change **C**hannel, **M**ove Node or **U**ndefined.
predictedReceivingRate	double	The predicted receiving rate if the action is not authorised.

Implementation

This data structure is used to store attribute values that relate to requests made by managers. We create two vector based on this structure. The first called newRequests that the manager node primarily uses and one that the Network Command Center uses.

Vector newRequests

The **`newRequests`** vector is populated by the **`RequestAction`** (p. 507) method which is invoked by a manager node. It is only ever accessed by the NetworkCommandCenter by copying the contents and then clearing the vector to allow for new requests.

Vector requests

The **`requests`** vector is populated by the **`NetworkCommandCenter`** (p. 508) and is an exact copy at a given moment in time as the newRequest vector. Once the copy has completed the NetworkCommandCenter clears the newRequest vector to allow for newRequests to be made whilst dealing with current requests.

The vector is also accessed by the **`GetManagerToMoveNodeFor`** (516) method to determine the most suitable manager to assign a controlled node to. The **`NetworkCommandCenter`** (p. 508) method copies the requests from this data structure to requests and clears the current values in newRequests.

Structure traffic

Reference	**Data Type**	**Description**
sNode	int	The source node Id
dNode	Int	The management node Id
timeStart	double	When to begin transmitting traffic
timeStop	double	When to stop transmitting traffic

Implementation

This data structure is used to schedule a traffic flow between a source node and a destination node. We record the source node identifier, the destination node identifier, when the source node should begin transmitting traffic and when the source node should cease transmitting traffic.

Vector myTrafficFlows

The myTrafficFlows vector is populated by the **`CreateTrafficFlows`** (p. 474) method which is invoked by a the **`ScheduleEvents`** (p. 441) method. We access the data structure to check if a source node is or is not transmitting by calling the **`StillTransmitting`** (p. 476) method. During the election process we identify the source nodes that need managers by accessing this data structure in the **`Election`** (p. 492) method. The **`GetDestinationNodeForSourceNode`** (p. 525) method also accesses this vector to get the most suitable destination node for a given source node when considering which link breakage to attempt to fix by utilising controlled node.

We also access the data structure to output traffic flow statistics to the terminal by calling the **`DoStats`** (p. 538) method.

Vector myTrafficCongestionFlows

The myTrafficCongestionFlows vector is populated by the **`CreateTrafficFlowsForCongestion`** (p. 476) method which is invoked by a the **`ScheduleEvents`** (p. 441) method in order to schedule events when congestion nodes should begin transmitting packets intent on causing congestion.

Structure relationshipState

Reference	Data Type	Description
sNode	int	The source node identifier
mNode	int	The management node identifier

time	double	The time the entry was added
range	double	The current range in meters between the source node and the destination node
energyLevel	double	The remaining energy in Joules of the energy source attached to the management node

Implementation

This data structure is used to record a given relationship state between a source node and a manager node. We store the source node identifier, the management node identifier, the time at which the state was recorded, the current distance between the source node and the manager node and the remaining energy of the energy source for the management node. We record relationship state information in order to check that the manager node is still suitable for the given source node.

Vector relationshipStates

The relationshipStates vector populated when we call the **`RecordRelationshipState`** (p. 520) method.

Structure packetRate

Reference	Data Type	Description
time	double	The current time of information
source	int	Information about the current source node
sent	double	The current send rate per second
received	double	The current receive rate per second
dropped	double	The current drop rate per second

Implementation

The packetRate data structure is used to store the rate of packets for a source node that has been assigned a management node. Each time the management node checks the state of the network it will record to this structure the current time, the source node identifier, the send rate per second for the source node, the receive rate per second for the destination(s) that the source is transmitting to and the drop rate per second for the source nodes packets.

Vector packetRates

The packetRates vector is populated when the **`RecordPacketRate`** (p. 492) method is called. The vector is primarily used by the **`Management`** (p. 497) method in order to gauge if network performance has decreased or increased. We also use this vector to predict future rates by using historic values with current values.