5-2007

Selfish node isolation in Mobile Ad-Hoc Networks.

Michael Probus
University of Louisville

SELFISH NODE ISOLATION IN MOBILE AD-HOC NETWORKS

By

Michael Wayne Probus
B.S. University of Louisville

A Thesis
Submitted to the Faculty of the
University of Louisville
Speed Scientific School
as Partial Fulfillment of the Requirements
for the Professional Degree

MASTER OF ENGINEERING

Department of Computer Engineering & Computer Science
University of Louisville

Spring 2007

SELFISH NODE ISOLATION IN MOBILE AD-HOC NETWORKS

Submitted by: ________________________________
Michael Wayne Probus

A Thesis Approved on

____
(Date)

By the Following Reading and Examination Committee:

____
Dr. Anup Kumar, Thesis Director (CECS)

____
Dr. Mehmed Kantardzic, CECS

____
Dr. Julius Wong, Mechanical Engineering

Acknowledgements

I would like to thank Dr. Anup Kumar for his help and guidance on this thesis. I would also like to thank the members of the committee, Dr. Wong and Dr. Kartardzic, for their time. Lastly, I would like to thank my loving wife for her patience while I worked to support our family and finish degree. Without her love and understanding, I wouldn’t have been able to complete my Master’s of Engineering and Computer Science.

Abstract

This thesis will focus on the topic of Selfish Nodes within a Mobile Ad-Hoc Networks (MANET), specifically sensor networks due to their lower power and bandwidth. The approach used is a reputation based algorithm to isolate the selfish nodes from communication by using past history to determine how reliable the node is. The reputation of each node is determined by their behavior within the network. As a node continuously acts selfishly, their reputation is decreased, until finally meeting the minimum threshold; therefore they are determined to be malicious. A node's reputation is increased for successfully participation and communication with neighboring nodes, but once a node is determined to be malicious, they are ignored and cannot regain positive reputation.

Once a node is isolated, the remaining nodes must find alternate paths to send their data to avoid any and all selfish nodes, regardless of the increase in distance. The method could easily be transformed to expand such routing protocols as Destination-Sequenced Distance Vector (DSDV). By using the proposed algorithm within DSDV, functionality and performance will be increased in the MANET. As a result of the isolation, retransmission is decreased and throughput increased, therefore conserving power consumption of individual nodes and creating a more reliable network by having less error rate and spare bandwidth.

Table of Contents

List of Figures

List of Tables

Chapter 1 – Mobile Ad-Hoc Networks

1.1 Introduction

With the growing popularity of wireless communication, the popularity of MANETs (Mobile Ad Hoc Networks) has also grown. MANETs are mobile wireless networks that rapidly changing and unpredictable and have no fixed base stations or infrastructure design. The nodes are able to move about throughout the network, while still being able to communicate with other peers by using multi-hop communication. The nodes participating in the network are responsible for passing traffic between each other and carry out routing protocols

As with any type of communication, MANETs have their design flaws and security concerns. One such issue is the existence of one or more selfish nodes within the network. Selfish nodes are nodes within the network that wish to conserve their own power, therefore they deny receiving packets from other nodes, while at the same time attempt to send packets of their own to its neighbors (Kargl 2004). Selfish nodes can cause major concerns in a MANET, from dropping single packets to the point where no node can send any message, therefore taking the entire network offline.

In many ad-hoc networks, sensor nodes are used. The nodes are expected to receive and forward packets to one another, until the packet reaches its final destination. Sensor nodes have low power, small storage, low bandwidth, and limited processing capabilities. Therefore, some nodes wish to conserve their power instead of forwarding the packet from another node to its desired location.

1.2 **Types of MANETs**

There are two different basic categories of MANETs (Miranda, 2004).

1.2.1 **Closed System** (Karygiannis, 2006)

A closed system is one in which the design and specifications are proprietary to prevent third-party hardware or software from being used. A closed system usually supports one or more critical applications, such as those used in military operations. Due to the nature of the application, cooperation it at upmost importance, therefore maliciousness is not tolerable. Since maliciousness could be harmful to the operations, the nodes within a closed system are more likely to have some type of built in security mechanism to detect the nodes that are malicious.

1.2.2 **Open System** (Karygiannis, 2006)

In contrast to a closed system is the open system. Open systems allow third-party nodes and applications to run within the network. The strictness of the types of applications allowed is dependent upon the security policy of the network owner. In an open system, cooperation is optional, but encouraged. If the node doesn't cooperate, they are ignored or punished, depending on the algorithm within the network. Open systems can be a wide range of systems from the simple home user to a large organization with high security.

1.3 Review of Algorithm Design Literature

Various algorithms have been designed in recent years to resolve the issue of selfish nodes. Each algorithm takes a different approach to the problem, but the majority of these algorithms can be broken into three general categories.

1.3.1 **Reputation Based** (He, 2004)

In a reputation based algorithm, each node is responsible for either keeping track of other nodes, or obtaining the reputation from a centralized node on the network. If a node successfully participates in the transmission of data by forwarding data packets, the reputation of the node is increased, or if the node discards the packet by dropping it, the reputation is decreased. After the nodes reputation drops below a threshold set by the developer, the node is either punished or ignored.

1.3.2 **Credit-payment** (Yoo, 2005)

A credit based algorithm is similar to a reputation based algorithm. The difference is this algorithm is that each node begins with a set of credits. A node sends a packet to its neighbor node for forwarding. After successfully forwarding the packet, the sending node credits the neighbor as a reward. If nodes do not forward the packet, they will run out of credits, resulting in not having the ability to send their own packets.

1.3.3 **Game Theory** (Gupta, 2005)

In a game theory algorithm, each node uses previous history to determine the best path to send the packet. The amount of processing power utilized is dependent upon the node. The more power used, the best path can be chosen, but more power is consumed. As a result of the limited amount of power each node has, the node must choose between using a large amount of its power to find the best path, or use a small amount of its power and take chances with an alternate path.

1.4 **Problem Challenges**

The issue in which this thesis addresses is the existence of selfish nodes, specifically those that continuously drop packets, in Mobile Ad Hoc Networks. Selfishness can have disastrous effects within the MANET. If the system is a closed network, such as tracking vehicles within a particular area of land in military operations; the existence of selfish nodes could mean the difference between winning and losing a battle.

Often times, the existence of selfishness don't have such effects as described above. In open systems, usually selfishness only results in loss of data during transmission. If the network is designed correctly, the data can be retransmitted until a successful transmission. Although the data is eventually transmitted successfully, this results in an increase in bandwidth utilization and extra power usage by each of the nodes within the path of the transmission.

Some challenges of eliminating selfishness include the following:

1.4.1 **Tolerance**

If the threshold is too low in which to tolerate selfishness, then the error rate will be high due to an increased amount of discarded packets. If the threshold is too high, then there will be low error rate, but fewer nodes will be able to participate in routing because they will be seen as selfish.

1.4.2 **Bandwidth**

Since bandwidth is limited within the MANET, retransmission must be kept at a minimal. Each node is responsible for sending data using the best possible path in order to reduce retransmission. Therefore, each node must be able to recognize when it has a selfish node as a neighbor and find an alternate path to send the data if one is available.

1.4.3 **Power Consumption**

Each node is responsible for finding the best path to send the packet, but the node can't use too much power to determine the best path. Nodes are limited in amount of power available to them, therefore the more power used in finding a path results in a shorter life span for the node. If all nodes use a large amount of power trying to find the best path to route a packet, the network will eventually become unusable due to a large amount of isolated nodes.

1.5 Problem Formulation

The algorithm proposed in this paper is detection and removal based upon the reputation based algorithm described earlier. The main objective is to identify and isolate selfish nodes from the network. Through successful isolation, the MANET performance will be increased and will become more reliable.

1.6 Thesis Organization

This paper will lead the reader through the design process of the algorithm in Chapter 2. Chapter 3 will show the accuracy of the algorithm by comparing the results with and without implementation using various scenarios. Finally, in Chapter 4, an explanation will be given on how the algorithm could be improved upon and placed in a real-world environment for everyday use along with the required steps to follow for usage of the algorithm

Chapter 2 – Design of Algorithm for Selfishness in MANETs

2.1 Background Survey

The algorithm to which I am proposing is based upon previous reputation based algorithms. As mentioned earlier, reputation based algorithms are dependent upon previous history to determine the reliability of neighboring nodes. It uses this factor of reliability to determine which neighbor to use when sending data to a more distant node and which neighbor to avoid.

When designing the algorithm, I focused on improvements for functionality and performance. These considerations include:

2.1.1 Complete Isolation

In many designs, participating nodes are able to recognize a selfish node. Therefore, they avoid sending data to the selfish node to be forwarded. When accepting a request to forward data, the receiving node does not check the reputation of the sending node. This allows selfish nodes to be selective when they participate in the network by sending its own data to willing neighbors but gives them the choice of not participating when they don't want to. The refusal to send data to the selfish nodes, but accepting the data of a selfish node, encourages all nodes in the network to be selfish.

2.1.2 Route Discovery

When the network is originally set up, all nodes must participate in a route discovery to learn how to send data to other nodes. The initial reputation is dependent upon the algorithm and is set to all neighbors of all nodes. As time progresses the reputations of all nodes change. In most algorithms, a new node placed in the network at a later time uses the same strategy of doing a route discovery and using the default reputation. In my approach, a new node will get the reputation of its neighbors from other neighboring nodes. This will give a better understanding of the current network to the new node, thus providing better performance.

2.1.3 Equality of Dropping vs. Forwarding

In the former algorithms, the reputation either increases by a set amount for forwarding packets or decreases by the same set amount for dropping packets. This can result in up to a 50% error rate if a node chooses to participate in sending 50% of the requests it receives. I propose that punishment is greater than reward, therefore dropping should account for more than forwarding. For example, a drop decreases the reputation by one, but a forward increases reputation by only one-tenth. This results in less than 10% error rate.

2.2 General Algorithm Design

Each node in the network under this scheme will consist of the same configuration. They will contain two tables, a neighbor table and a packet table. The neighbor table consists of the id of each neighbor and the reputation index of its neighboring nodes. After selecting a path, the source node checks the neighbor table to see if the neighboring node is selfish or not. If so, then the packet is discarded since it can't be forwarded. The second table contains all necessary information about each packet of each received packet of data.

The network in this design will be static allowing for better test results. After the initial route discovery, the nodes will not have to perform the discovery again unless a new node is added to the network. In this case, only the neighboring nodes will be required to make changes to their neighbor table. This will allow them to conserve their power and use it for data transmission and path determination.

Two selfish nodes will be added manually to the design to assure that the selfishness exists. These particular nodes will be marked as selfishness to the algorithm, but the surrounding nodes do not know of their marking. The remaining nodes must discover which of the nodes are selfish through behavior patterns.

2.3 Detailed Design

The proposed algorithm can be broken down into several parts. These include the creation of the simulated nodes which also includes creating the neighbor list for each node, packet generation, checking the receive queue for valid packets, forwarding packets to the next hop, reputation increase or decrease, and the addition of a new node. Each of these processes are explained in more detail in the following sections.

2.3.1 Node Creation and Simulated Network Setup

To begin the process, the simulated network is designed and configured. Each node is first created. The area is based on a 30 x 30 grid with each node representing a single point within the grid. Since the network is static, the nodes are created with specific X & Y coordinates. Each node is also given other characteristics including:

ID: The unique identifier of each node. This allows all nodes to distinguish their neighboring nodes from each other when deciding whom to send the data to for forwarding. This design has 9 nodes, numbered sequentially 0 through 8. This would be similar to using Media Access Control (MAC) address or Internet Protocol (IP) address for a unique identifier in a real world environment. A MAC address is a unique hardware address that identifies every node on the network (http://www.webopedia.com/TERM/M/MAC_address.html). An IP address is a software identifier for each node on a network. (http://www.webopedia.com/TERM/I/IP_address.html).

TYPE: Each node is defined as either malicious or normal. This ensures that there are a set number of participating nodes and selfish nodes. In a real world environment normal nodes participate willingly within the network, while malicious nodes participate when they want to or more often not at all. In this network, normal nodes always participate and the malicious nodes participate about 10% of the time, explained in more detail later.

R_INC: This is the value at which a node increases the reputation of its neighbor as a reward for successfully forwarding a packet. For testing purposes, an increment value of .1 was used. With a default value of 10, it will take 50 repetitions of successfully participating before a node can reach the maximum value.

R_DEC: The value at which a node decreases the reputation of its neighbor as punishment for dropping a packet. This implementation uses a value of 1.0. With a default value of 10, it will take only 5 repetitions of not participating within the network before the node reaches the minimum value, while it will take 50 repetitions of participating to recover for the decrements.

Note Both the increment values and decrement values can be easily changed. The less of a difference between the two numbers indicates a less restrictive policy, but is more prone to retransmissions due to more data being sent to malicious nodes. A greater difference indicates a more restrictive policy, but a participating node may be determined to be malicious if it is unable to communicate for one of various reasons.

R_MAX: This is the maximum reputation value any neighboring node can obtain for participating. This implementation uses a value of 15 as the maximum. After a node reaches this value, it can only be decremented. Any further participation doesn't allow for further incrementing.

R_MIN: This is the minimum reputation value any neighboring node can obtain for not participating. This implementation uses a minimum value of 5. Once a node reaches this value, it is ignored by all other nodes, but in receiving and sending, therefore a node at the minimal value can never participate in the network again in this design. In a real world environment, the designer can choose to reset the reputation or give the node another chance to participate after a specific time.

R_ZERO: This is the default reputation value a node assigns to all of its neighbors within the table. This implementation has a default value of 10. All nodes created at the beginning of the network setup obtain the default reputation. Any node added to the network after this point is assigned the default reputation, but the new node uses the global average of the existing nodes for its reputation table.

R_VISION: This is the distance at which a node can see and communicate with neighboring nodes. The value is calculated using the X & Y coordinates, explained later in more detail.

After all of the nodes are created, they begin the process of finding their neighbors so they can send data to each other. This is done by doing a discovery of all nodes within the vision requirements. To find the nodes within the vision range, the geometrical distance formula is used:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Using a for loop, each pair of the X & Y coordinates is compared to the remaining pairs. The values are used within the formula to get the distance. If the value of d is less than or equal to 10, then the nodes are considered to be neighbors. After a node finds a neighbor, it adds the neighbor to its neighbor list with the default reputation. This process is then continued until all nodes have been compared with all other nodes for possible neighbors, therefore creating the network.

A real world environment will have an alternative way to find the neighbors. Each node will not know of the other nodes coordinates, therefore a for loop is not feasible. Mobile networks will use send out a beacon and wait for a reply. Any device that is able to respond is obviously within the range of the node, therefore they are able to establish communication as neighbors.

The layout of the nodes and their connections to their neighbors are shown in Figure 1.

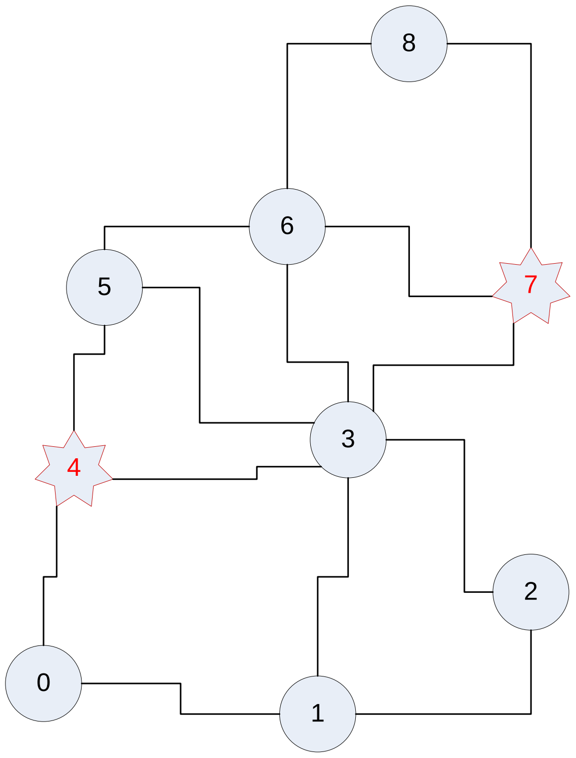

Figure 1 – Node Layout with Connections to Neighbors

2.3.2 Packet Generation

After the network is created, the nodes need data to send to one another, so the next step is to generate the packets. The packet table in this implementation is a scaled down version of the table used in real implementations. During each iteration a packet is created for each node by all neighbors. Each packet is created with specific information such as:

SOURCE: The node in which the packet is created and added to the sending queue is always the source node. This is the node that decides to increment or decrement the neighbor depending on their participation level.

DESTINATION: The final node in which the packet is to reach. This is the node that returns the acknowledgement to the source, verifying that the packet was received as expected. In this implementation, the destination is always the neighboring node. This allows for better testing results for maliciousness without having to focus on the proper routing of packets. A real world implementation would use one of numerous protocols to find the best path, ranging from least number of hops to quickest round trip response time.

SEQUENCE NUMBER: This is used to distinguish packets from each other to avoid duplicate processing; therefore conserving performance and battery. This is similar to the identification field in the IP protocol.

DATA TYPE: The packet type is defined as one of three types; default, data, or acknowledgement. The receiving node of the packet uses the data type to determine how to process it.

DEFAULT: Default packets are used as dummy packets to find the best path available when trying to determine which node to forward the packet to for further processing.

DATA: This represents the simulation of data being transferred between nodes. When receiving a data packet, the node decides to process the packet itself if it is the destination or forward the packet on to the next hop in the route.

ACKNOWLEDGEMENT: When the data is received by the destination, it returns an acknowledgement to the source, therefore verifying that the packet was received and the reputation of the neighboring node should be increased accordingly.

When a node receives an acknowledgment packet, further processing is not needed. The packet is discarded and removed from the receive queue.

TRACING PATH: This contains the path the data packet has traveled allowing the destination to know where to send the acknowledgement. This also allows for trace back in a real world environment for issues such as a node attempting a DOS or some other attack method. A trace back in the real world environment would reveal the IP address of the source, allowing the administrator of the network to take action as necessary.

2.3.3 **Receive Queue**

After the packets are generated, the next step would be to send the data to the destination. Before forwarding any packets, the node must first check its queue to see if it received any new packets that needs to be acknowledged. Directly after the network is created, no nodes would have any data in their receive queue until data is sent, but each time they prepare to send anything afterwards the node must check for new packets. This allows the node to send any data packets to their destination at the same time it is processing acknowledgements of received data packets instead of making it a two step process. To do this, the node checks the size of it's receive queue. If the size of the queue is greater than 0, then the node has packets that need to either be acknowledged or forwarded.

Not all packets in the receive queue need to be processed. Therefore, they must be checked to find out which ones are valid. If any of the following requirements are met, the packet can be discarded. Once a requirement is fulfilled, the check is stopped and node checks the next packet in the queue.

MALICIOUS NODE: Malicious nodes drop over 90% of the packets received. One out of ever ten packets received by them is checked for the remaining requirements; the remaining nine packets are immediately discarded before any checks are performed. If the packet passes all of the remaining tests, then it doesn't get dropped.

SOURCE = DESTINATION: If the packets are returned back to the source because they can't be routed, then the source drops the packet since it has no where to send it.

DUPLICATE PACKETS: If the packet has already been processed once, then the duplicate packet is dropped.

SOURCE IS MALICOUS: If the sending node has a low reputation representing that of a malicious node, then the packet is dropped by the destination. This keeps non-participating nodes from attempting to send their own data and participating in the network when they want to.

IN TABLE: The node checks its current packet table for packets currently waiting to be processed. If it finds a packet with the same type, source, destination, and sequence number, than the packet already exists and the node discards the duplicate request. As punishment for sending duplicate requests, the packet is not only discarded, but the reputation of the sending node is decremented.

2.3.4 **Packet Forwarding**

If the received packet passes all of the previous checks, it is determined to be a valid packet. It is next checked to be a data packet. If so, then the packet is added to the receiving nodes packet table for processing.

The first check in determining how to process the packet is to determine if the receiving node is the destination. If the determination is that it is the destination, then it performs the following steps.

1. Creates and acknowledgement packet to send back to the source, verifying the receipt of the packet.
2. Adds itself as the source of the acknowledgement and the source of the original packet as the destination.
3. Adds the last hop of the original packet as the next hop of the acknowledgment.
4. Increases the sequence number of the acknowledgement to distinguish it from other packets.
5. Places the acknowledgement packet onto the sending queue of the current node.
6. Marks the packet for removal from the receiving queue.

If the current node is not the destination then the packet must be forwarded to the next hop. When this is the case, the following steps are performed.

1. The current node adds itself to the route of the packet for trace-back.

2. Since routing tables are not used in this implementation, the node doesn't know the correct route. Therefore, the only option is trial and error. The node checks the reputation of all of its neighbors. If it finds a neighbor that is determined to be malicious, that node is ignored in the transmission process
3. The node sends the packet to all available neighbors attempting to get a response back from the destination, excluding those neighbors that are malicious.

If the current node is the destination and the packet is an acknowledgement, than the packet doesn't need to be processed further. The only action that needs to be taken is the removal of the packet from the receive queue.

2.3.5 Reputation Decrease

If a node sends a packet, but doesn't get a response back, it decreases the reputation of the neighboring node regardless of fault. It is the responsibility of the neighboring node to know the correct path to send the packet. The packet must be able to travel the entire path while avoiding malicious nodes. Below is an example of the reputation topology.

1. Node 1 sends packet to Node 2.
2. Node 2 has neighbors 3, 4, and 5. Node 4 is malicious. Node 2 must recognize the maliciousness of Node 3, therefore avoiding sending the packet to him.
3. Node 3 received the packet from Node 2. It has the option to sending to only Node 4.

4. The reputation of Node 2 is decreased in the table of Node 1 since it should have recognized that Node 3 had only the option to send to Node 4, a malicious node.

This algorithm isn't ideal since routing tables aren't used, but the advantage will be shown later when it is incorporated into a real world environment.

2.4 **New Node Addition**

When new nodes are added to the network after the initial setup, issues may arise if the new node is a neighbor with a malicious node. In most algorithms, the fact that the malicious node was blacklisted is ignored. When the new node is added to the network, it is allowed to transmit with the malicious node. This algorithm takes a slightly different approach to resolve this issue.

1. The new node is created and does a route discovery similar to the initial setup. The neighboring nodes are added to the neighbor list of the new node.
2. The new node is added to the neighbor list of the neighboring nodes.
3. The new node is given the default reputation of 10 by all neighboring nodes.
4. The new node assigns each node their global reputation.

Using this strategy, nodes which have been blacklisted remain blacklisted, therefore not allowing them to cause problems on the network again.

Chapter 3 – Performance Analysis and Results

3.1 Bandwidth

Since the nodes that would be using this algorithm have limited power and bandwidth, performance is a major factor in determining functionality. Therefore, to improve performance, retransmissions and packet loss should be minimized. In order to measure the accuracy of the design for packet loss, three scenarios have been formulated for analysis. In all scenarios, packets are generated every 15 iterations, therefore the only difference is the value of the reward or punishment for choosing whether to participate or not.

3.1.1 Scenario 1

Scenario 1: There is no punishment or reward for dropping packets or forwarding packets respectively. To simulate this scenario, the increment and decrement values have been changed to zero. This means that regardless of the actions taken by each node, they will be treated no different from any other node since the reputation will always remain at zero.

Node	Forwarded	Dropped	Received	Sent
0	1189	0	1228	34
1	1386	0	1471	51
2	1319	0	1409	34
3	1430	0	1566	102
4	178	2759	224	51
5	1495	0	1579	51
6	1478	0	1563	68
7	156	2659	170	51
8	1020	0	1137	34
Total	9651	5418	10347	476

Table 1 – Performance: Scenario 1 Results

As shown in Table 1, Scenario 1 had poor performance. It is easily seen that nodes four and seven are malicious nodes since they are the only nodes that dropped any packets, but the two of them dropped over 5,400 packets in only 250 iterations.

3.1.2 **Scenario 2**

Scenario 2: Punishment and reward are equal, therefore offering better performance than Scenario 1, but is prone to high error rate since a node can participate 50% of the time and remain at the default reputation of 10. To simulate this scenario, a value of 1 was used for both increment and decrement.

Node	Forwarded	Dropped	Received	Sent
0	1113	0	1152	34
1	1306	0	1391	51
2	1255	0	1345	34
3	1332	0	1458	102
4	166	2259	204	51
5	1405	0	1489	51
6	1406	0	1491	68
7	147	2534	161	51
8	1066	0	1083	34
Total	9196	4793	9774	476

Table 2 – Performance: Scenario 2 Results

As shown in Table 2, by looking at the number of packets dropped, Scenario 2 had better performance than Scenario 1. The number of packets dropped was decreased by only 625 or 11.5%. In an environment with limited bandwidth, a savings of over 11% is a considerable difference, but with a small modification, it can be improved upon. When time matters, a network needs all of the resources possible and bandwidth is a major factor in determining response time in systems.

3.1.3 **Scenario 3**

Scenario 3: Punishment is greater than reward. To simulate this, an increment value of 0.1 is used, but a decrement value of 1.0 is used. This means that it takes ten increments to make up the difference of only one decrement.

Node	Forwarded	Dropped	Received	Sent
0	815	0	840	34
1	956	0	1041	51
2	978	0	1068	34
3	939	0	1075	102
4	56	889	66	51
5	1018	0	1090	51
6	1002	0	1087	68
7	58	1165	66	51
8	791	0	808	34
Total	6613	2054	7141	476

Table 3 – Performance: Scenario 3 Results

Scenario 3 showed significantly better performance over both of the previous scenarios. The number of packets dropped was reduced to 2,054, an additional 2,739 packets from Scenario 2 which is a savings of 57.1%. The total performance savings from Scenario 1 was a reduction of 3,365 packets dropped or a 62% decrease. This means that in a network that allows malicious nodes, 62% of the packets sent never reach their destination because they are dropped in transition.

3.1.4 **Bandwidth Summary**

- Scenario 1 – No Punishment
    - Over 5,400 packets were dropped
- Scenario 2 – Equal Reward and Punishment
    - Decreased by 625; 11.5%

- Scenario 3 – 10:1 Ratio of Reward and Punishment
  - Reduced an additional 2,739; 57.1%
  - Total reduction is 3,365; 62%

3.1.5 **Additional Iterations**

As shown, Scenario 3 has significantly increased performance. Additional iterations will show an even more significant increase since the malicious nodes are completely blacklisted during this time.. When running the simulation for Scenario 3 at 1000 iterations, the expected dropped packets using straight line estimation would be about 8,216 at the rate of 2,054 per 250 iterations. When running the scenario, the following results were determined.

Node	Forwarded	Dropped	Received	Sent
0	2697	0	2772	134
1	3154	0	3489	201
2	3378	0	3734	134
3	2967	0	3503	402
4	92	1339	102	201
5	3396	0	3668	201
6	3330	0	3665	268
7	108	3013	116	201
8	2687	0	2754	67
Total	21809	4352	23803	1809

Table 4 – Performance: Scenario 3 Addition Iteration Results

As shown in Table 4, the number of dropped packets was 4,352, must lower than the estimated 8,216. By increasing only 750 iterations, we were able to show an addition

47% increase over the previous scenario. As expected, the more iterations that are ran, the better the results will be. In a network that has continuous data being transmitted, the savings will be substantial.

This can be seen by running Scenario 1 for 1,000 iterations. Scenario 1 had 5,418 dropped packets in 250 iterations. At that rate, we would expect to have 21,672 packets dropped in 1,000 iterations. Below are the actual results.

Node	Forwarded	Dropped	Received	Sent
0	4730	0	4885	134
1	5484	0	5819	201
2	5235	0	5591	134
3	5578	0	6114	402
4	709	10908	895	201
5	5826	0	6160	201
6	5703	0	6038	268
7	624	10464	666	201
8	4410	0	4477	67
Total	38299	21372	40645	1809

Table 5 – Performance: Scenario 1 Addition Iteration Results

There weren't quite the expected number, but very close at 21,372. In comparing these results against the results of scenario 3 at 1,000 iterations, we showed a decrease of over 17,000 dropped packets, or 80%. The number of total packets forwarded due to retransmissions also decreased by 16,490, or a 43% savings.

3.1.6 **Additional Iteration Summary**

- ⦿ Scenario 1 – No Punishment
    - 21,372 – Close to the expected value
    - Increase of over 17,000 dropped packets; 80%
    - Total packets increased by over 16,000; 43%
- ⦿ Scenario 3 - 10:1 Ratio of Reward and Punishment
    - 4,352 – Much lower than the expected 8,216
    - 47% increase
    - Additional iterations would show more improvement

3.2 **Error Rate**

To determine the possible error rate, the number we must find the number of iterations it takes for the participating nodes to recognize the malicious nodes and blacklist them. For comparison, we will use the same scenarios as in the performance measure.

3.2.1 **Scenario 1**

Scenario 1: No punishment or reward. In this scenario, the nodes will never be blacklisted regardless of the number of iterations. It is easily predictable that with an infinite number of iterations, all nodes will remain at the default value of 10 since the reputation never changes. This can result in up to 100% error rate when sending nodes to or through the malicious nodes.

The following chart shows every 10th iteration over the entire 250 iteration scenario. As shown below, all remain at the same value in all iterations through all 250 iterations. The upper and lower lines represent the min and max. The middle line represents the default value and all of the node reputation values.

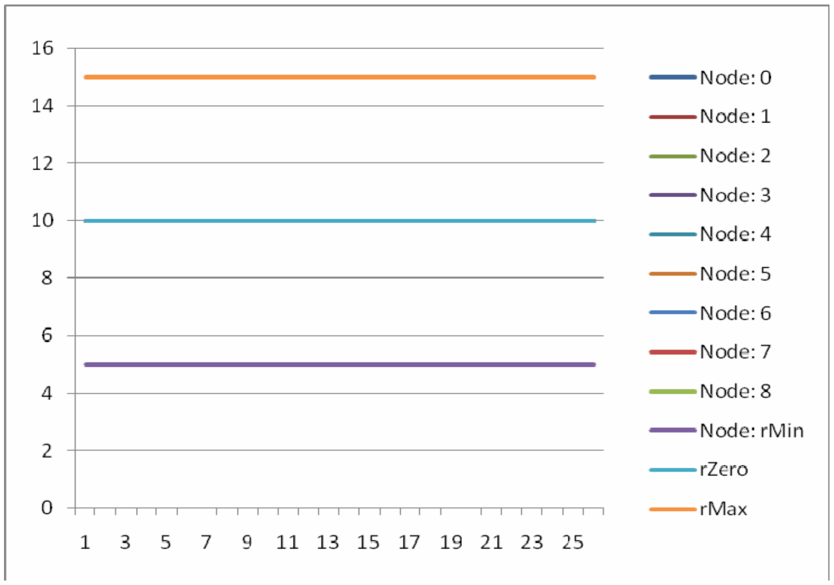

Figure 2 – Error Rate: Scenario 1 Reputation Values Over 1000 Iterations

3.2.2 **Scenario 2**

Scenario 2: Punishment and reward are equal. Since a node can choose to participate when it wants to, all nodes can remain at their default level by participating 50% of the time. If a node never gets blacklisted, then the error rate will remain high. If and when all malicious nodes get blacklisted, the error rate will improve significantly. Once the nodes get blacklisted, the only factor that will determine the error rate is the network in terms of items such as strength of signal between nodes.

Iteration	Node: 0	Node: 1	Node: 2	Node: 3	Node: 4	Node: 5	Node: 6	Node: 7	Node: 8
1	10	10	10	10	10	10	10	10	10
1000	15	15	15	13.33	9.6667	15	12.5	8.33	15

Table 6 – Error Rate: Scenario 2 Reputation Values Over 1000 Iterations

In this scenario, the malicious nodes never reached blacklisted status in the first 1,000 iterations. The average reputation for the malicious nodes is slightly below the default, so they have began to fall and will eventually get blacklisted, but until then, the network reliability is unknown.

In the chart below, it can be seen that the average reputation of all nodes never falls below the default value of 10. The majority of them achieve the maximum value at one point in the scenario.

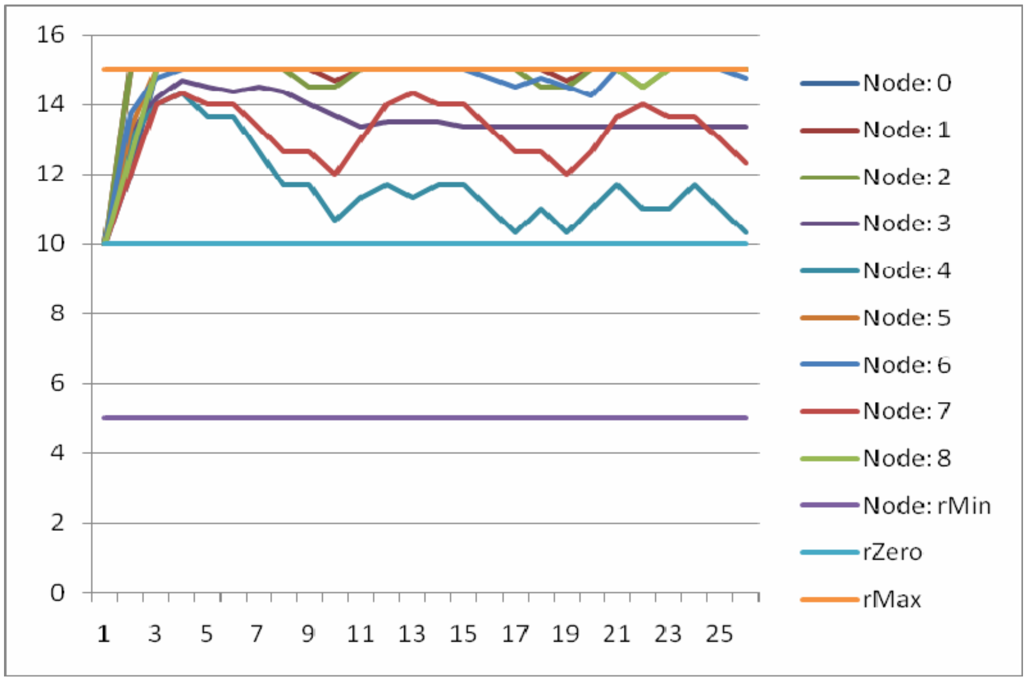

Figure 3 – Error Rate: Scenario 2 Reputation Values Over 1000 Iterations

3.2.3 Scenario 3

Scenario 3: Punishment is greater than reward. In this scenario, the error rate is determined by the difference between the increment value and decrement value. With values for increment and decrement at 0.1 and 1.0 respectively, the error rate cannot be higher than 10% since it takes 10 increments to recover from on decrement.

Iteration	Node: 0	Node: 1	Node: 2	Node: 3	Node: 4	Node: 5	Node: 6	Node: 7	Node: 8
1	10	10	10	10	10	10	10	10	10
71	13	15	15	12.2	5.267	13.93	13.7	6.93	12.25
72	13	15	14.5	11.97	5	13.93	13.7	6.8	12.25

Table 7 - Error Rate: Scenario 3 Reputation Values Over 1000 Iterations

As seen above, Node 4 reached full malicious status by all neighbors on the 71st iteration. At this point, it should have no data passed to it or received from it. Node 7 has not reached malicious status at this point, but is getting very close. At this point, the status of Node 7 from neighbors Node 3, Node 6, and Node 8 is 10.4, 5, and 5 respectively. Node 7 is now blacklisted by Nodes 6 and 8, but Node 3 will continue to see and receive information from it. Therefore, the error rate has dropped from 10% when transmitting with three different nodes, to 10% when transmitting with only one node.

In this scenario, Nodes 4 and 7 never went above the default value of 10, while the remaining nodes never dropped below the default value. This scenario clearly has the best performance of the three.

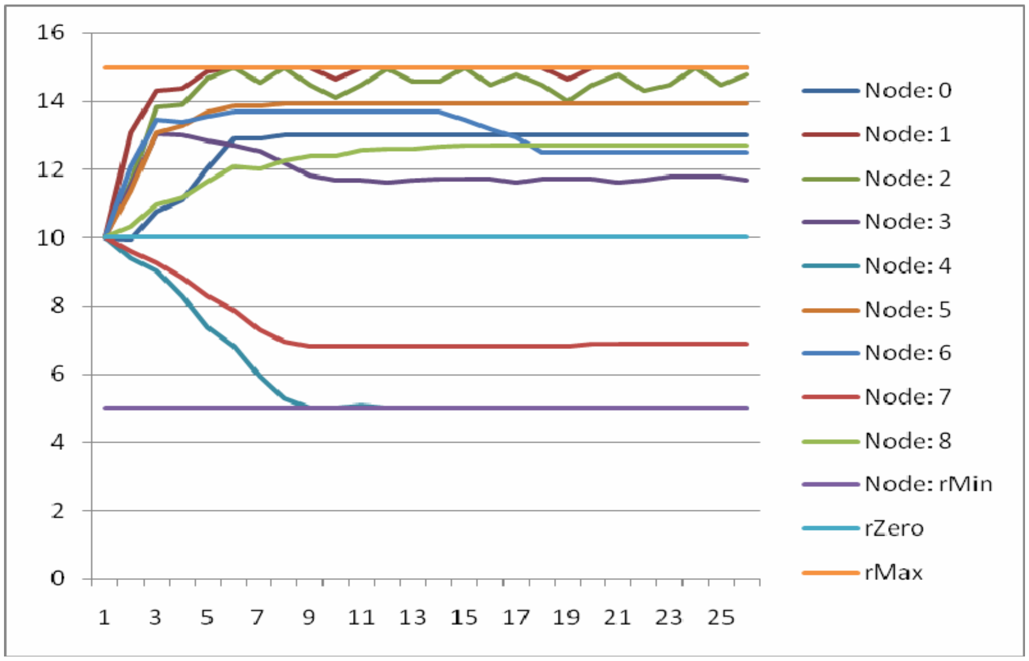

Figure 4 – Error Rate: Scenario 3 Reputation Values Over 1000 Iterations

3.2.4 **Error Rate Summary**

- ⦿ Scenario 1 – No Punishment
  - Reputation never changes
- ⦿ Scenario 2 - Equal Reward and Punishment
  - Reputations never fall below initial value
- ⦿ Scenario 3- 10:1 Ratio of Reward and Punishment
  - Reputation never goes above initial value
  - Node 4 reaches full malicious value at 71st interval

3.3 **Node Additions**

The last focus on this thesis is the addition of new nodes after the initial network has been created. To keep from increasing error after the malicious nodes have been blacklisted, the new node is to use the global reputation of its neighbors as the default value instead of the original default of 10. If the new nodes use the default value, they will send packets to the nodes that were blacklisted, therefore causing problems on the network that were already eliminated.

To show the impact we will create two scenarios using the increment and decrement values of our previous Scenario 3. The algorithm will be run for 500 iterations before the new node is added and 500 iterations after the new node is added. The new node is Node 9. It has neighbors Nodes 4 and 5. Since Node 9 is a direct neighbor of Node 4, it will be our focus to compare the results.

We will then compare the packets dropped as before. The first 500 iterations will return the same results and those results will then be compared to the 2nd 500 iterations in both scenarios. Below are the results from the first iteration.

Node	Forwarded	Dropped	Received	Sent
0	1455	0	1497	68
1	1704	0	1874	102
2	1794	0	1974	68
3	1627	0	1899	204
4	68	1039	78	102
5	1824	0	1964	102
6	1792	0	1962	136
7	76	1787	84	102
8	1432	0	1466	68
9	-	-	-	-
Total	11772	2826	12798	952

Table 8 – Node Additions: Base Results Over 500 Iterations

3.3.1 **Scenario 1**

Scenario 1: The new node will give the default value of 10 to each of its neighbors.

Node	Forwarded	Dropped	Received	Sent
0	3471	0	3560	134
1	4042	0	4377	201
2	4208	0	4564	134
3	3769	0	4305	402
4	306	3125	352	234
5	4185	0	4510	234
6	4110	0	4445	268
7	164	3390	172	201
8	3311	0	3378	134
9	1973	0	2026	66
Total	29539	6515	31689	2008

Table 9 – Node Additions: Scenario 1 Over 500 Iterations from 501 - 1000

After the first 500 iterations, Node 4 was blacklisted by all neighbors. When Node 9 was added, Node 4 was no longer blacklisted, therefore, the packets sent to Node 4 increased again, therefore increasing the packets dropped. During the 2nd 500 iterations, Node 4 dropped 2,086 packets which were about twice the amount that it dropped in the first 500, (1,039).

3.3.2 **Scenario 2**

Scenario 2: The new node will use the global reputation of its neighbors instead of using the default values.

Node	Forwarded	Dropped	Received	Sent
0	3393	0	3468	134
1	3854	0	4189	201
2	4076	0	4432	134
3	3320	0	3856	402
4	92	1333	102	234
5	3573	0	3878	234
6	3682	0	4017	268
7	136	3360	144	201
8	3034	0	3101	134
9	1430	0	1463	66
Total	26590	4693	28650	2008

Table 10 – Node Additions: Scenario 2 Over 500 Iterations from 501 - 1000

This scenario showed must better performance than the previous. Here, Node 4 dropped only 294 packets in the 2nd 500 iterations. This is a large comparison to scenario 1 at 2,086. The difference of the two resulted in a difference of 1,792 dropped packets, or a savings of 85.9%

3.3.3 Node Addition Summary

- Scenario 1
    - Node 4 dropped an additional 2,086 packets
    - About twice as much as the 1st 500 iterations
- Scenario 2
    - Node 4 dropped only 294 packets
    - 1,792 less packets than scenario 1; 86% savings

Chapter 4 – Conclusion and Further Implementations

4.1 Sensor Networks

Power management in sensor nodes is based upon supply and consumption. The more the node does, the sooner the node runs out of power. Therefore, to conserve power, the nodes try to do as little as possible. This includes trying to participate with neighboring nodes by choice. These nodes that participate when they want to are referred to as malicious or selfish. Selfish nodes drop packets from other nodes, but continuously ask other nodes to forward packets for them.

4.2 Destination-Sequenced Distance Vector (DSDV)

Destination-Sequenced Distance Vector, DSDV, was created in 1994 by Charles Perkins. It is a table-driven routing scheme for wireless ad-hoc networks, based upon the Bellman-Ford algorithm.

4.2.1 Distance Vector Routing (Madhusudhan, http://www.laynetworks.com)

Distance vector routing requires that each node informs each other of its routing table. The receiving node chooses the neighbor that is advertising the lowest cost to a particular destination. It then adds this neighbor to its routing table and re-advertises its table to other nodes. The advertisement of routing tables is both periodic and triggered, meaning that is schedules advertisement transfer on a regular basis, and the advertisements are transferred when one or more changes are made to routing table.

Advantages of distance vector routing include:

1. Distribution: This algorithm enables each node receives some information from one or more of its directly attached neighbors.
2. Iteration: The process of exchanging information will continue until no more information is exchanged between the neighborhood.
3. Asynchronous: This algorithm does not require all of the nodes to operate in lock step with each other.

4.2.2 **Bellman-Ford Algorithm** (Black, 2005)

The Bellman-Ford Algorithm is used to compute a single-source shortest distance routing path in a weighted digraph where edge weights may be negative. It's main contribution is the resolve the issue of routing loop. The algorithm first initializes the source vertex to 0 and all other vertices to ∞. It then does $V - 1$ passes, where V is the number of vertices and updates all the distance of all edges. Finally, it checks for negative weight cycles. If a negative weight cycles is found, a FALSE is returned to the system.

4.2.3 **Bellman-Ford in DSDV** (Madhusudhan, http://www.laynetworks.com)

Routers must maintain distance tables in order to use distance-sequenced distance vector routing. These tables tell the distance and shortest path to each node on the network for sending packets. The information in these tables are dynamically updated by the exchange of information with neighboring nodes. The columns of the table represent the directly attached neighbors and the rows represent all destinations in the network. Included in the table is the path the packet must travel and the distance or time to transmit. Measurements to calculate the cost are hops, latency, number of outgoing packets, etc.

4.2.4 **Advantages and Disadvantages of DSDV**

DSDV claims to have the following properties. (Prasad, 2006)

1. Loop-free at all instants;
2. Dynamic, multi-hop, self-starting;
3. Low memory requirements;
4. Quick convergence via triggered updates;
5. Routes available for all destinations;
6. Fast processing time;
7. Reasonable network load;
8. Minimal route trashing;
9. Intended for operation with up to 100 mobile nodes, depending on *mobility factor*.

Disadvantages: (Perkins, 1994)

1. Requires regular updates of routing tables, therefore bandwidth efficiency is low.
2. Not very scalable, therefore not suitable for large networks.

3. Not suitable for highly dynamic networks since the network is unreliable for a short period when the network topology changes.

4. CPU utilization. As the size of the routing tables increase, the demand for CPU utilization also increases.

4.3 **Improvements**

Using the proposed algorithm with the addition of routing tables in a real world environment could help solve the some of the issues that currently exist with distance-sequenced vector routing and sensor nodes, specifically the bandwidth issue and power consumption. This would improve the overall efficiency of the network, therefore making it more reliable and trustworthy.

4.3.1 **Bandwidth Efficiency**

The proposed algorithm has proven to make the network more trustworthy by excluding the malicious nodes. Once the malicious nodes are excluded, the number of packets required to be transmitted is decreased. The number of packets at which it is decreased is dependent upon the number of nodes excluded and how many packets are transmitted through them.

Furthermore, the efficiency is increased since the participating nodes do not have to exchange routing tables with the malicious nodes. The revised algorithm would use the following steps for a guideline.

If anytime within the process, a route changes for any node, they immediately transfer routing tables with all necessary nodes

1. Check receiving queue for incoming packets.
2. Check type of packets and destination. If a destination can not be reached, return the packet back to the sender.
3. For packets needing to be forwarded, check the reputation of the sending node.
4. If the reputation of the sending node is less than or equal to the minimum acceptable value, then drop the packet. Otherwise, forward the packet on requested.
5. Send out packets as necessary.
6. If the node fails to get an acknowledgement back from a destination, decrease the reputation of the neighboring node. It is the responsibility of the neighboring node to know the correct path to send the packet to avoid retransmission or loop routing.
7. Transfer routing tables between nodes for a periodic update.
8. Continue this process until a malicious node is found. If a malicious node is found, update the corresponding routing table and transfer tables.
9. Continue the process of receiving and sending, making sure to send the packets to the correct route, therefore avoiding the malicious nodes, improvement bandwidth efficiency.

4.3.2 **Power Consumption**

Since the nodes will be sending fewer packets to complete the same tasks, they will be required to do less work. Therefore, they will be using less power. By solving the issue of bandwidth efficiency, the issue of power consumption is also reduced.

4.4 **Conclusion and Summary**

As demonstrated, the proposed algorithm works in a simulated network. The results show significant improvement over taking no action against malicious nodes. In all cases the number of dropped packets was decreased, therefore bandwidth was conserved because the retransmission rate was reduced. The savings in retransmission of packets is a determinant in the savings of power consumption for each sensor node and the increase in reliability of the network.

References

Black, Paul E., "Bellman-Ford algorithm", in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 4 March 2005. (Accessed April 2, 2007) Available from: http://www.nist.gov/dads/HTML/bellmanford.html

Gupta, R., Somani, A.K., (2005). Game Theory as a Tool to Strategize as Well as Predict Nodes' Behavior in Peer-to-Peer Networks. *Parallel and Distributed Systems*, Volume 1, 244-249, July 2005.

He, Q., Wu, D., Khosla, P. (2004). SORI: A Secure and Objective Reputation-based Incentive Scheme for Ad-Hoc Networks. *Wireless Communications and Networking Conference,* Volume 2, 825-830, March 2004.

Kargl, Frank, Klenk, Andreas, Schlott, Stefan, and Weber, Michael. (2004). "Advanced Detection of Selfish or Malicious Nodes in Ad hoc Networks" (pdf). (online), Accessed April 23, 2007, http://medien.informatik.uni-ulm.de/forschung/publikationen/esas2004.pdf

Karygiannis, A., Antonakakis, E., and Apostolopoulos, A. (2006). "Detecting Critical Nodes for MANET Intrusion Detection Systems," (pdf). (online), Accessed April 23, 2007, http://csrc.nist.gov/manet/Critical-Nodes-MANET.pdf

Madhusudhan N,. Accessed April 2, 2007, Available from: http://www.laynetworks.com/Bellman%20Ford%20Algorithm.htm

Miranda, H., Rodrigues, L., (2004). Preventing Selfish Behavior in MANETs.

Perkins, Charles E. and Bhagwat, Pravin. (1994). "Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers" (pdf). (online), Accessed April 2, 2007, http://en.wikipedia.org/wiki/Destination-Sequenced_Distance_Vector_Routing

Prasad, R., & Deneire, L. (2006). *From WPANs to Personal Networks: Technologies and Applications***.** Artech House

What is IP Address? (n.d). Retrieved April 23, 2007, from http://www.webopedia.com/TERM/I/IP_address.html

What is MAC Address? (n.d). Retrieved April 23, 2007, from http://www.webopedia.com/TERM/M/MAC_address.html

Yoo, Y., Ahn, S., Agrawal, D.P. (2005). A Credit-payment Scheme for Packet Forwarding Fairness in Mobile Ad-Hoc Networks. *IEEE International Conference on Communications*, Volume 5, 3005-3009, May 2005.

```
#ifndef NODE_CPP
#define NODE_CPP

#include <list>
#include <iterator>
#include <string>

using namespace std;

#define T_NORMAL 0                              //Participating node
#define T_MALICIOUS 1                           //Malicious node
#define P_REQ 0                                 //Packet types
#define P_ACK 1
#define P_DATA 2

#define D true
#define E false

int Z_DROP;

node::node(){};                                 //Default Constructor
node::~node(){};                                //Default Destructor

node::node(int idz, double posXz, double posYz, int typez, double rIncz, double rDecz,
double rMaxz, double rZeroz, double rMinz, double visionThreshz)
{
        type = typez;                           //Copy all passed initial values
        id = idz;
        rInc = rIncz;
        rDec = rDecz;
        rMax = rMaxz;
        rZero = rZeroz;
        rMin = rMinz;
        posX = posXz;
        posY = posYz;
        visionThresh = visionThreshz;

        packetsForwarded = 0;                   //Reset counters
        packetsDropped = 0;
        packetsSent = 0;
        packetsRecieved = 0;
        packetsNonRouted = 0;
}


int node::getID(){ return id;}
```

```
bool node::isDest(){ return ((*recvIter).dest == id);}

bool node::isACK(){  return ((*recvIter).type == P_ACK);}

bool node::isACKPTable(){ return ((*packetIter).type == P_ACK);}

bool node::isREQ(){  return ((*recvIter).type == P_REQ);}

bool node::isDATA(){ return ((*recvIter).type == P_DATA);}

void node::addHost(int hostID){hostList.push_back(hostID);}

void node::deletePacketEntry(){       if (inPacketTable()) packetTable.erase(packetIter);}

void node::addPacketEntry(){packetTable.push_back((*recvIter));}

double node::getPosX(){ return posX;}

double node::getPosY(){ return posY;}

double node::getVision(){ return visionThresh;}

bool node::isNeighbor(int neighborID)
{
        for (neighborIter = neighborList.begin(); neighborIter != neighborList.end();
neighborIter++)
        {
                if (neighborID == (*neighborIter).id) return true;
                                //NeighborIter will now be pointing to the last neighbor
        }
        return false;
}

bool node::inPacketTable()
                                //Need to verify this.  Checks if the recieved packet has
                                same type, src, dest, and more or equal seqNum
{
        for (packetIter = packetTable.begin(); packetIter != packetTable.end();
packetIter++)
        {
                if ( ((*recvIter).type == (*packetIter).type) && ((*recvIter).src ==
(*packetIter).src) && ((*recvIter).dest == (*packetIter).dest) && ((*recvIter).seqNum
== (*packetIter).seqNum))
                {
                        bool same = true;
                        list<int>::iterator routeIter2;
                        for (routeIter = (*recvIter).route.begin(), routeIter2 = (*packetIter).route.begin(); routeIter != (*recvIter).route.end(), routeIter2 != (*packetIter).route.end();routeIter++, routeIter2++)
                        {
                                if ((*routeIter) != (*routeIter2)) same = false;
                        }
                        return same;
                }
        }
        return false;
}

void node::decrementRep()
{
        (*neighborIter).reputation -= rDec;
        if ((*neighborIter).reputation < rMin)
                (*neighborIter).reputation = rMin;
        if (D)cout << "Node: " << id << "\tDecrementing Reputation of Node: " << (*neighborIter).id << "\tNew Reputation: " << (*neighborIter).reputation << endl;
}

void node::deletePacketDummies(int lastHop)
{
        for (packetIter = packetTable.begin(); packetIter != packetTable.end();)
        {
                if ( ((*recvIter).dest == (*packetIter).dest) && ((*recvIter).src == (*packetIter).src) && ((*packetIter).type == P_ACK) && ((*packetIter).nextHop == lastHop) )
                {
                        if (D)cout << "Erasing PACKET!" << endl;
                        if (D)printPacket();
                        packetTable.erase(packetIter);
                        return ;
                }
                else packetIter++;
        }
}




void node::dropMarkedPackets()
```

```
{
        for (recvIter = recvQueue.begin(); recvIter != recvQueue.end();)
        {
                if ((*recvIter).dest == -1)       recvIter = recvQueue.erase(recvIter);
                else recvIter++;
        }
}

void node::forwardPacket()
{
        if ((type == T_MALICIOUS) && (Z_DROP % 10 !=0))
        {
                packetsNonRouted++;
                packetsDropped++;
                return ;
        }

        int       tempID = (*recvIter).route.back();

        if (inPacketTable())
        {
                if (isREQ() || isDATA())
                {
                        int lastHop = lastHopPTable();
                        if ((lastHop != -1) && isNeighbor(lastHop))
                        {
                                decrementRep();
                                deletePacketEntry();
                                if ((*neighborIter).reputation <= rMin)
                                {
                                        (*recvIter).dest = -1;   //Drop the packet
                                        packetsDropped++;
                                        if (D)cout << "Node: " << id << "\tMarking " <<
(*neighborIter).id << "\'s Packet due to reputation of " << (*neighborIter).reputation <<
endl;
                                        return ;
                                }
                        }
                }
        }

        if (isREQ())                                      //Data Packet
        {
                addPacketEntry();
                if (isDest())                             //Data at Destination
                {
                    if (D)cout << "Node: " << id << "\tRecieved REQ packet from
Node: " << (*recvIter).src << endl;

                    if (lastHop() != (*recvIter).src)        //Don't encourage spamming.
                              incrementRep();

                    if (D)cout << "Node: " << id << "\t Creating ACK packet to Node:
" << (*recvIter).src << endl;

                    (*recvIter).type = P_ACK;
          //Change type to ACK
                    (*recvIter).dest = (*recvIter).src;
                    (*recvIter).src = id;
                                        //Swap src and dest, we are returning to source now
                    (*recvIter).nextHop = (*recvIter).route.back();
                                        //Turn packet around back to the last hop
                    (*recvIter).seqNum++;
                    (*recvIter).route.push_back(id);
                                        //Add ourselves to the very end of the route
                    sendQueue.push_back((*recvIter));
                                        //Push onto send queue
                    (*recvIter).dest = -1;
                                        //Mark for removal from recvQueue
                    packetsRecieved++;
                    packetsForwarded++;
          }
          else
                                        //Not Destination and not originator
          {
                    if (lastHop() != (*recvIter).src)
                                        //Dont encourage spamming.
                              incrementRep();

                    addPacketEntry();

                    (*recvIter).seqNum++;
                    (*recvIter).route.push_back(id);        //stamp id onto route
                    for (neighborIter = neighborList.begin(); neighborIter !=
neighborList.end(); neighborIter++)
                    {
                              if ((*neighborIter).id != tempID &&
(*neighborIter).reputation > rMin)                  //This guy just sent it to you!
                              {
                                        (*recvIter).nextHop = (*neighborIter).id;
                                        //Set next Hop to the possible neighbors
                                        if (D)cout << "Node: " << id <<
"\tForwarding REQ to Neighbor " << (*neighborIter).id << " Src: " << (*recvIter).src <<
" Dest: " << (*recvIter).dest << endl;
                                        sendQueue.push_back((*recvIter));

                                }
                        }
                        (*recvIter).dest = -1;

                        packetsRecieved++;
                        packetsForwarded++;
                }
        }
        else                                            //ACK Packet
        {
                int tempHop = lastHop();
                if (tempHop == -1) return ;
        //Our dummy packets have probably been deleted already, a route has been found
                incrementRep();

                if (isDest())                           //ACK returned to original requestor
                {
                        deletePacketDummies(tempHop);
                                        //Delete our dummy ACK packet from the table
                                        //Network would be safer if he deleted all dummies.
                                        he now has a clean route to this host
                                        //No retransmissions if not needed
                        if (D)cout << "Node: " << id << "\tRecieved ACK packet from
destination Node: " << (*recvIter).src << endl;
                        (*recvIter).dest = -1;

                        packetsRecieved++;
                }
                else
                {
                        for (routeIter = (*recvIter).route.begin(); routeIter !=
(*recvIter).route.end(); routeIter++)
                        {
                                if ((*routeIter) == id)
                                                //Our location in the forward path
                                {
                                        routeIter--;
                                                //The node that sent the REQ to us.
                                        (*recvIter).nextHop = (*routeIter);
                                        sendQueue.push_back((*recvIter));
                                        if (D)cout << "Node: " << id << "\tForwarding
ACK packet along route to " << (*recvIter).dest << endl;
                                        (*recvIter).dest = -1;
                                        packetsForwarded++;
                                        packetsRecieved++;
                                        break;
                                                //Only forward it once.
                                }
                        }

                }
        }
}

void node::incrementRep()
{
        (*neighborIter).reputation += rInc;
        if ((*neighborIter).reputation > rMax)
                (*neighborIter).reputation = rMax;
        if (D)cout << "Node: " << id << "\tIncrementing Reputation of Node: " <<
(*neighborIter).id << "\tNew Reputation: " << (*neighborIter).reputation << endl;
}

int node::lastHop()
{
        int lastID = -1;
        if (isREQ())
        {
                lastID = (*recvIter).route.back();
                        //Data packet, last hop was the neighbor at the end of the route path
        }
        else
        {
                for (routeIter = (*recvIter).route.begin(); routeIter !=
(*recvIter).route.end(); routeIter++)
                {
                        if ((*routeIter) == id)
                        {
                                routeIter++;
                                lastID = (*routeIter);
                        }
                }
        }

        for (neighborIter = neighborList.begin(); neighborIter != neighborList.end();
neighborIter++)
```

```
	{
		if (lastID == (*neighborIter).id)
		{
			return lastID;		//NeighborIter will now be pointing to the
last neighbor
		}
	}
	return lastID;
}


int node::lastHopPTable()
{
	int lastID = -1;
	if (isREQ()) lastID = (*packetIter).nextHop;	//Data packet, last hop was the
neighbor at the end of the route path
	else
		if (isACKPTable())
			lastID = (*packetIter).nextHop;
		else
		{
			for (routeIter = (*packetIter).route.begin(); routeIter !=
(*packetIter).route.end(); routeIter++)
			{
				if ((*routeIter) == id)
				{
					routeIter++;
					lastID = (*routeIter);
				}
			}
		}

	if (lastID == -1) return -1;
	for (neighborIter = neighborList.begin(); neighborIter != neighborList.end();
neighborIter++)
	{
		if (lastID == (*neighborIter).id) return lastID;	//NeighborIter will now
be pointing to the last neighbor
	}
	return lastID;
}

void node::markForDrop()		//Dont run this in the middle of a recvIter loop, it
calls dropMarkedPackets and changes recvIter
{
	for (recvIter = recvQueue.begin(); recvIter != recvQueue.end(); recvIter++)
```

```
        {
                if (isACK())continue;


                if ((type == T_MALICIOUS) && ((*recvIter).dest != id) && (Z_DROP
% 10 != 0))
                {
                        (*recvIter).dest = -1;
                        packetsNonRouted++;
                        packetsDropped++;
                        if (D)cout << "Node: " << id << "\tMaliciously dropping " <<
(*recvIter).src << "\'s packet of type " << (*recvIter).type << endl;
                        continue;

                }

                if ((*recvIter).src == id)
                {
                        (*recvIter).dest = -1;          //Drop own packets if they come
back to us.
                        if (D)cout << "Node: " << id << "\tMarking own packet REQ for
drop, looping." << endl;
                        packetsNonRouted++;
                        continue;
                }
                else
                {
                        for (routeIter = (*recvIter).route.begin(); routeIter !=
(*recvIter).route.end(); routeIter++)
                        {
                                if ((*routeIter) == id)
                                {
                                        (*recvIter).dest = -1;          //Drop Datapackets
that this node has already routed once....
                                        if (D)cout << "Node: " << id << "\tMarking " <<
(*neighborIter).id << "\'s Packet due to routing loop. Type: " << (*recvIter).type << endl;
                                        packetsNonRouted++;
                                        continue;
                                }
                        }

                        if (lastHop() != -1)
                        {
                                if ((*neighborIter).reputation <= rMin)
                                {
                                        (*recvIter).dest = -1;          //If the last hops
reputation was bad, drop.
                                        if (D)cout << "1Node: " << id << "\tMarking " <<
(*recvIter).route.back() << "\'s Packet due to reputation of " << (*neighborIter).reputation
<< endl;
                                        packetsNonRouted++;
                                        continue;
                                }
                        }
                }
        }
        dropMarkedPackets();                    //Need to drop for bad reputations!
}

void node::printPacket()
{
        cout << "\t\tSrc: " << (*packetIter).src << " Dest: " << (*packetIter).dest << "
SeqNum: " << (*packetIter).seqNum << " NextHop: " << (*packetIter).nextHop << "
Type " << (*packetIter).type << endl;
        cout << "\t\t\tRoute: ";
        for (routeIter = (*packetIter).route.begin(); routeIter != (*packetIter).route.end();
routeIter++)
        {
                cout << (*routeIter) << " ";
        }
        cout << endl;
}

void node::sending(int sendTo)

                        //Called to generate packets, will be handled by process
{
        //Real implementation would use packet table for routes
        int tempDest;
        for (hostIter = hostList.begin(); hostIter != hostList.end(); hostIter++)
        {
                if (( (*hostIter) == sendTo) || (sendTo == -1))
                {
                        tempDest = (*hostIter);

                        packet tempPacket;              //Create temporary packet
that will be setup and put in send queue
                        tempPacket.src = id;            //Current node is the source
                        tempPacket.dest = tempDest;     //Destination is the
neighboring node
```

```
			tempPacket.seqNum = -1;			//Default	Sequence
Num, used to check if dest was in packet table already

			tempPacket.type = P_REQ;		//Default packet type, we are
not sending an ack unless we recieve a data
			tempPacket.route.push_back(id);		//Add our id to the
current route stored in the packet (tracing path)

			if (tempPacket.seqNum == -1)		//If the sequence
number was not set above, dest was not in packet table
			{
				tempPacket.seqNum = 1;		//Default Seq Num for
tracing path
				for (neighborIter = neighborList.begin(); neighborIter !=
neighborList.end(); neighborIter++)		//For each neighbor!
				{
					if ((*neighborIter).reputation > rMin)
	//If the neighbor is not blacklisted
					{
						tempPacket.nextHop = (*neighborIter).id;
	//Set him as next hop
						sendQueue.push_back(tempPacket);
			//Add packet to be queued
						packet tempPacket2;
						tempPacket2.seqNum = 1;
						tempPacket2.nextHop =
tempPacket.nextHop;
						tempPacket2.src = tempPacket.dest;
						tempPacket2.dest = id;
						tempPacket2.type = P_ACK;
						tempPacket2.route.push_back(id);

	tempPacket2.route.push_back(tempPacket.nextHop);//Add the neighbor we are
sending to the end of the route for decrementing if its dropped
						packetTable.push_back(tempPacket2);
			//Put a dummy ack packet
					}
				}
			}
			packetsSent++;
		}
	}
}

void node::status()
{
	cout << "Node: " << id << endl;
	cout << "\tPosition: (" << posX << "," << posY << ")" << endl;
	cout << "\tVision: " << visionThresh << endl;
	cout << "\tType: ";
	if (type == T_NORMAL) cout << "Normal" << endl;
	else cout << "Malicous" << endl;

	cout << "\tNeighbors: " << endl;
	for (neighborIter = neighborList.begin(); neighborIter != neighborList.end(); 
neighborIter++)							//For each Neighbor
	{
		cout << "\t\tID: " << (*neighborIter).id << " Rep: " << 
(*neighborIter).reputation << endl;
	}
	cout << "\tPacket Info: " << endl;
	cout << "\t\tForwarded:\t" << packetsForwarded << endl;
	cout << "\t\tDropped:\t" << packetsDropped << endl;
	cout << "\t\tRecieved:\t" << packetsRecieved << endl;
	cout << "\t\tSent:\t\t" << packetsSent << endl;
	cout << "\t\tNonRouted:\t" << packetsNonRouted << endl;

	cout << "\tSend Queue: " << endl;
	for (sendIter = sendQueue.begin(); sendIter != sendQueue.end(); sendIter++)
										//For 
each Packet in SendQueue
	{
		cout << "\t\tSrc: " << (*sendIter).src << " Dest: " << (*sendIter).dest << " 
SeqNum: " << (*sendIter).seqNum << " NextHop: " << (*sendIter).nextHop << " Type: " 
<< (*sendIter).type << endl;
		cout << "\t\t\tRoute: ";
		for (routeIter = (*sendIter).route.begin(); routeIter != 
(*sendIter).route.end(); routeIter++)
		{
			cout << (*routeIter) << " ";
		}
		cout << endl;
	}

	cout << "\tRecieve Queue: " << endl;
	for (recvIter = recvQueue.begin(); recvIter != recvQueue.end(); recvIter++)
									//For each 
Packet in RecvQueue
	{
		cout << "\t\tSrc: " << (*recvIter).src << " Dest: " << (*recvIter).dest << " 
SeqNum: " << (*recvIter).seqNum << " NextHop: " << (*recvIter).nextHop << " Type " 
<< (*recvIter).type << endl;
```

```
			cout << "\t\t\tRoute: ";
			for (routeIter = (*recvIter).route.begin(); routeIter !=
(*recvIter).route.end(); routeIter++)
			{
				cout << (*routeIter) << " ";
			}
			cout << endl;
		}
		cout << endl;
}

void node::basicProcess(int retrans, int Z_VALUE)
{
		Z_DROP = Z_VALUE;
		if (D)cout << "Node: " << id << "\tTotal Packets in queue: " << recvQueue.size()
<< endl;

		markForDrop();
		if (D)cout << "Node: " << id << "\tPackets in queue after drop: " <<
recvQueue.size() << endl;

		for (recvIter = recvQueue.begin(); recvIter != recvQueue.end(); recvIter++)
		{
			forwardPacket();
		}
		recvQueue.clear();


		if (retrans == 0)
		{
			for (packetIter = packetTable.begin(); packetIter != packetTable.end();)
			{
				if (((*packetIter).dest == id) && ((*packetIter).type == P_ACK))
			//If this is one of our dummy ack packets
				{
					if (D)cout << "Node: " << id << "\tRetransmitting!" <<
endl;

					if (D)printPacket();

					int tempHop = lastHopPTable();

					if (tempHop != -1)
					{
						decrementRep();
```

```
                    packet tempPacket;
                    tempPacket.src = (*packetIter).dest;
                    tempPacket.dest = (*packetIter).src;
                    tempPacket.nextHop = (*packetIter).nextHop;
                    tempPacket.seqNum = (*packetIter).seqNum;
                    tempPacket.route.push_back(id);
                    tempPacket.type = P_REQ;
                    sendQueue.push_back(tempPacket);
                    if (D)
                    {
                        cout << "New Packet " << endl;

                        cout << "\t\tSrc: " << tempPacket.src << "
Dest: " << tempPacket.dest << " SeqNum: " << tempPacket.seqNum << " NextHop: " <<
tempPacket.nextHop << " Type " << tempPacket.type << endl;
                        cout << "\t\t\tRoute: ";
                        for (routeIter = tempPacket.route.begin();
routeIter != tempPacket.route.end(); routeIter++)
                        {
                            cout << (*routeIter) << " ";
                        }
                        cout << endl;
                    }

                    sendQueue.push_back(tempPacket);
                }

                return ;
            }
            else packetIter++;
        }
    }
}

#endif
```

```
#ifndef NODE_H
#define NODE_H

#include <list>
#include <iterator>
#include <string>
#include <queue>

using namespace std;

typedef struct neighbor
{
	int id;				//Neighbor Table Entries
	double reputation;		//ID and Reputation are all that we need
};

typedef struct packet
{
	int src;			//Source of packet
	int dest;			//Dest of Packet
	int type;			//Packet Type P_DATA, or P_ACK
	int seqNum;			//Sequence Number
	int nextHop;			//Next Hop
	list<int> route;			//Route taken so far
};


class node
{
	public:
		node();
		~node();
		node(int idz, double posXz, double posYz, int typez, double rIncz, double
rDecz, double rMaxz, double rZeroz, double rMinz, double visionThreshz);
		void addHost(int hostid);


		void basicProcess(int retrans, int RANDOM);
	//
		void sending(int sendTo);		//Create some packets.
		void forwardPacket();
		void dropMarkedPackets();
		void decrementRep();
		void incrementRep();
		void markForDrop();


		void status();					//Output Status
```

```
double getPosX();                                    //Get PositionX

double getPosY();                                    //Get PositionY

double getVision();                                  //Get Vision

int getID();                                         //Get ID


int lastHop();

int lastHopPTable();




bool isREQ();

bool isACK();

bool isACKPTable();

bool isDATA();

bool isDest();

bool isNeighbor(int neighborID);

bool inPacketTable();


void deletePacketEntry();

void addPacketEntry();




void deletePacketDummies(int lastHop);
```

```
                void deleteAllPacketDummies();

                void printPacket();

                list<int>      hostList;                        //The list of hosts to
send to
                list<packet> sendQueue;                         //The queue\list that
packets are placed in upon creation.
                list<packet> packetTable;                   //The table that stores the
packet information.
                list<packet> recvQueue;                         //The queue\list that
recieved packets are pushed in
                list<neighbor> neighborList;           //Neighbors in visible range.

                list<neighbor>::iterator neighborIter; //Used for iterating through the
nodes neighborlist
                list<packet>::iterator packetIter;          //Used for iterating through
the packet Table
                list<packet>::iterator sendIter;            //Used for iterating through
the send Queue/List
                list<int>::iterator routeIter;              //Used for iterating through
the route within each packet
```

```
                list<packet>::iterator recvIter;                //Used for iterating through
the recieve Queue/List
                list<int>::iterator hostIter;                   //Used for iterating through
the send list of hosts


        private:
                int type;                                                               //The
nodes type T_MALICIOUS or T_NORMAL
                int id;
        //Nodes id/index into the main programs vector
                double rInc;                                            //Reputation
Scheme's        Increment value, per node setting.
                double rDec;                                            //
                        rDecrement
                double rMax;                                            //
                        rMax
                double rZero;                                           //
                        rZero
                double rMin;                                            //
                        rMin
                double posX;                                            //Node
PositionX
```

```
		double posY;							//Node
PositionY
		double visionThresh;					//Nodes Vision


		int packetsForwarded;					//Integers for tracking
node behavior.
		int packetsDropped;
		int packetsSent;
		int packetsRecieved;
		int packetsNonRouted;
};

#endif

#include "node.cpp"
```

```
//Runner.cpp

#include <iostream>

#include <math.h>


#include "node.H"


using namespace std;


#define T_NORMAL 0                                    //Node types

#define T_MALICIOUS 1

#define P_REQ 0                                               //Packet Types

#define P_ACK 1

#define P_DATA 2


#define D true                                        //Print outs

#define E false


#define RETRANS_RATE 9                //Should be around the number of nodes in
the network.

#define R_INC 0.1

#define R_DEC 1                                               //RDEC can also be
used to show that the algorithm is working by
```

```
#define R_MAX 15.0                         //scaling the output (global average)
to show that malicious nodes are generally lower
#define R_ZERO 10.0                             //but it makes it more
apparent because these are global averages of the malicious nodes
#define R_MIN 5.0                               //neighbors opinions.  If this
were a real network the DATA packets could be used
#define R_VISION 10                        //to further reinforce strong bonds
SIMPLE ACK BACK


int tempID = -1;


                    //First ID will be 0
int i, j, k;


vector<node> nodeList;
                                //The Vector of ALL the Nodes!
vector<double> globalRep;


vector<node>::iterator iter1;
             //Useful iterators for said nodeList.
vector<node>::iterator iter2;
```

```
void discoverNeighbors()

{

        double distance;

                                                //Distance to next node

        neighbor tempNeighbor;

                                        //A neighbor structure for temp usage


        for (i = 0; i < nodeList.size(); i++)                                   //For

all nodes

        {

                for (j = 0; j < nodeList.size(); j++)                           //For

all combinations of nodes

                {

                        if (j == i)     continue;

                                                    //Node is not its own neighbor


                        distance = sqrt( pow(( nodeList[i].getPosX() -

nodeList[j].getPosX() ), 2) + pow(( nodeList[i].getPosY() - nodeList[j].getPosY() ), 2));


                        if ( distance < nodeList[i].getVision() )

                        {
```

```
                                        tempNeighbor.id = nodeList[j].getID();                    //We
know the neighbors id
                                        tempNeighbor.reputation = R_ZERO;
                    //Default Reputation from define

                                        nodeList[i].addHost(tempNeighbor.id);                     //Add
this neighbor to this nodes hostList

                                        nodeList[i].neighborList.push_back(tempNeighbor);
          //Add this neighbor to this nodes neighborlist

                                        if (D)cout << "Node: " << nodeList[i].getID() << " has a
Neighbor: " << nodeList[j].getID() << endl;  //Print status
                              }
                    }
          }
}

void generatePackets()
{
          for (iter1 = nodeList.begin(); iter1 != nodeList.end(); iter1++){ (*iter1).sending( -
1); }
}
```

```
void nodeBasicProcess(int retrans, int Z_VALUE)
{
        for (iter1 = nodeList.begin(); iter1 != nodeList.end(); iter1++){
(*iter1).basicProcess(retrans, Z_VALUE);}
}

//Call status on every node in the nodeList.
void status()
{
        for (iter1 = nodeList.begin(); iter1 != nodeList.end(); iter1++){ (*iter1).status();}
}

void transmitPackets()
{
        list<packet>::iterator sendIter;
        //Create iterator for each nodes send Queue

        for (i = 0; i < nodeList.size(); i++)                                    //For
all nodes in the network
        {
                if (D)cout << "Processing Node: " << nodeList[i].getID() << endl;
```

```
			for (sendIter = nodeList[i].sendQueue.begin(); sendIter !=
nodeList[i].sendQueue.end(); sendIter++) //For all the packets in the ith node's send
queue
			{
				nodeList[(*sendIter).nextHop].recvQueue.push_back((*sendIter));
										//Push this packet into the
next hops recieve queue
				if (D)cout << "Moving packet from Node: " << nodeList[i].getID()
<< " forwarding to Node: " << (*sendIter).nextHop << endl;
			}
			nodeList[i].sendQueue.clear();
			//Clear last nodes queue now.
		}
}

void newnode()
{
		list<neighbor>::iterator iter3;

		int nodevalue;

		node *tempNode2;
```

```
        tempNode2 = new node(9, 0, 15, T_NORMAL, R_INC, R_DEC, R_MAX,
R_ZERO, R_MIN, R_VISION);
        nodeList.push_back((*tempNode2));



        double distance;
                                                  //Distance to next node
        neighbor tempNeighbor;
                                  //A neighbor structure for temp usage
        neighbor tempNeighbor2;
                                  //A neighbor structure for temp usage

        nodevalue = (*tempNode2).getID();



        for (i = 0; i < nodeList.size(); i++)                              //For
all nodes
        {

                if (nodevalue == i)      continue;
                        //Node is not its own neighbor
```

```
                    distance = sqrt( pow(( nodeList[i].getPosX() -
nodeList[nodevalue].getPosX() ), 2) + pow(( nodeList[i].getPosY() -
nodeList[nodevalue].getPosY() ), 2));

                    if ( distance < nodeList[nodevalue].getVision() )
                    {

                            tempNeighbor.id = nodeList[i].getID();
                                                            //We know the neighbors id

                            tempNeighbor.reputation = (double)globalRep[nodeList[i].getID()]
/ (double)nodeList[i].neighborList.size();       //Default Reputation from global average
//                  tempNeighbor.reputation = R_ZERO;
                            nodeList[nodevalue].addHost(tempNeighbor.id);
                                    //Add this neighbor to this nodes hostList

                            nodeList[nodevalue].neighborList.push_back(tempNeighbor);
            //Add this neighbor to this nodes neighborlist

                            tempNeighbor2.id = nodeList[nodevalue].getID();
                                    //We know the neighbors id
```

```
                              tempNeighbor2.reputation = R_ZERO;
                                                                                    //Default Reputation
from define

                              nodeList[i].addHost(tempNeighbor2.id);
                                                            //Add this neighbor to this nodes hostList
                              nodeList[i].neighborList.push_back(tempNeighbor2);
                              //Add this neighbor to this nodes neighborlist

                              if (D)cout << "Node: " << nodeList[nodevalue].getID() << " has a
Neighbor: " << nodeList[i].getID() << endl;  //Print status
                    }

          }

          if (D)status();

}

int main()
{
          int seed;
          seed = 401; //Change random seed
```

```
        srand(seed);

        list<neighbor>::iterator iter;

        if (D)cout << endl << endl << "***Program Startup***" << endl << endl;
        if (D)cout << endl << endl << "***Generating Nodes***" << endl << endl;

        node *tempNode;
        //Node Creation
        //tempNode = new
node(ID,POSX,POSY,TYPE,R_INC,R_DEC,R_MAX,R_ZERO,R_MIN,R_VISION);
        tempNode = new node(0, 6, 5, T_NORMAL, R_INC, R_DEC, R_MAX,
R_ZERO, R_MIN, R_VISION);
        nodeList.push_back((*tempNode));
        tempNode = new node(1, 15, 4, T_NORMAL, R_INC, R_DEC, R_MAX,
R_ZERO, R_MIN, R_VISION);
        nodeList.push_back((*tempNode));
        tempNode = new node(2, 22, 8, T_NORMAL, R_INC, R_DEC, R_MAX,
R_ZERO, R_MIN, R_VISION);
        nodeList.push_back((*tempNode));
        tempNode = new node(3, 16, 13, T_NORMAL, R_INC, R_DEC, R_MAX,
R_ZERO, R_MIN, R_VISION);
        nodeList.push_back((*tempNode));
        tempNode = new node(4, 7, 12, T_MALICIOUS, R_INC, R_DEC, R_MAX,
R_ZERO, R_MIN, R_VISION);
        nodeList.push_back((*tempNode));
        tempNode = new node(5, 8, 18, T_NORMAL, R_INC, R_DEC, R_MAX,
R_ZERO, R_MIN, R_VISION);
        nodeList.push_back((*tempNode));
        tempNode = new node(6, 14, 20, T_NORMAL, R_INC, R_DEC, R_MAX,
R_ZERO, R_MIN, R_VISION);
        nodeList.push_back((*tempNode));
        tempNode = new node(7, 22, 18, T_MALICIOUS, R_INC, R_DEC, R_MAX,
R_ZERO, R_MIN, R_VISION);
        nodeList.push_back((*tempNode));
        tempNode = new node(8, 18, 26, T_NORMAL, R_INC, R_DEC, R_MAX,
R_ZERO, R_MIN, R_VISION);
        nodeList.push_back((*tempNode));



        if (D)cout << endl << endl << "***Discovering Neighbors***" << endl << endl;
        discoverNeighbors();

        if (D)cout << endl << endl << "***Network Status***" << endl << endl;
        if (D)status();
        if (D)cout << endl << endl << "***Generating Packets***" << endl << endl;
        generatePackets();

        //                              CSV HEADER
        //                              The graph provides a view of their GLOBAL
average reputation.
        //
        if (E)
                {
                        cout << "Iteration,Node: ";
                        for (int z = 0; z < nodeList.size();z++)
                        {
                                cout << nodeList[z].getID() << ",Node: ";
                        }
                }
        cout << "rMin,rZero,rMax" << endl;

        //                                                              Main Loop
        for (int z = 1; z <= 500; z++)
        {
                if (D)cout << endl << endl << "***Generating Packets***" << endl <<
endl;
```

```
                if (z % 15 == 0) generatePackets();
        //Sending on fixed interval



                if (D)cout << endl << endl << "***Processing Recv Queue***" << endl
<< endl;
                nodeBasicProcess(z % RETRANS_RATE, z);


                if (D)cout << endl << endl << "***Transmitting Packets***" << endl <<
endl;
                transmitPackets();


                globalRep.clear();
                                                //Clear old global average reputations
                for (int i = 0;i < nodeList.size();i++)
                {
                        globalRep.push_back(0.0);
                        //Add one back for each
                }
                for (int i = 0;i < nodeList.size();i++)
                {
                        for (iter = nodeList[i].neighborList.begin(); iter !=
nodeList[i].neighborList.end(); iter++)
```

```
                        {
                                globalRep[(*iter).id] += (*iter).reputation;
                        }
                }

        if (E)cout << z << ",";
                if (E)
                {
                        for (int i = 0;i < nodeList.size();i++)
                        {
                                cout << (double)globalRep[nodeList[i].getID()] /
(double)nodeList[i].neighborList.size() << ",";
                }
                        cout << R_MIN << "," << R_ZERO << "," << R_MAX << "," <<
endl;
                }
                if (D)cout << endl << endl << "***Network Status " << z << " ***" <<
endl << endl;
                if (D)status();
                if ((D) && z == 250)
                {
                                cout << z << ",";
                                for (int i = 0;i < nodeList.size();i++)
                        {
                                cout << (double)globalRep[nodeList[i].getID()] /
(double)nodeList[i].neighborList.size() << ",";
                        }
                        cout << R_MIN << "," << R_ZERO << "," << R_MAX << "," <<
endl;
                        }
                }

        newnode();

        for (int z = 1; z <= 500; z++)
        {
                if (D)cout << endl << endl << "***Generating Packets***" << endl <<
endl;

                if (z % 15 == 0) generatePackets();
        //Sending on fixed interval



                if (D)cout << endl << endl << "***Processing Recv Queue***" << endl
<< endl;
                nodeBasicProcess(z % RETRANS_RATE, z);
```

```
			if (D)cout << endl << endl << "***Transmitting Packets***" << endl <<
endl;
			transmitPackets();


			globalRep.clear();
									//Clear old global average reputations
			for (int i = 0;i < nodeList.size();i++)
			{
				globalRep.push_back(0.0);
				//Add one back for each
			}
			for (int i = 0;i < nodeList.size();i++)
			{
				for (iter = nodeList[i].neighborList.begin(); iter !=
nodeList[i].neighborList.end(); iter++)
				{
					globalRep[(*iter).id] += (*iter).reputation;
				}
			}

		if (E)cout << z << ",";
			if (E)
```

```
                    {
                              for (int i = 0;i < nodeList.size();i++)
                              {
                                        cout << (double)globalRep[nodeList[i].getID()] /
(double)nodeList[i].neighborList.size() << ",";
                    }
                              cout << R_MIN << "," << R_ZERO << "," << R_MAX << "," <<
endl;
                    }
                    if (D)cout << endl << endl << "***Network Status " << z << " ***" <<
endl << endl;
                    if (D)status();
                    if ((D) && z == 250)
                    {
                                        cout << z << ",";
                                        for (int i = 0;i < nodeList.size();i++)
                              {
                                        cout << (double)globalRep[nodeList[i].getID()] /
(double)nodeList[i].neighborList.size() << ",";
                              }
                              cout << R_MIN << "," << R_ZERO << "," << R_MAX << "," <<
endl;
                              }
```

```
                }


        return 0;

}
```