DEVELOPMENT OF AN ANDROID APP TO CONTROL AND MANAGE COGNITIVE NETWORKS

RELATORE: Ch.mo Prof. Zorzi Michele

LAUREANDO: Szabo Karoly Albert

A.A. 2013-2014

UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
TESI DI LAUREA

DEVELOPMENT OF AN ANDROID APP TO CONTROL AND MANAGE COGNITIVE NETWORKS

RELATORE: Ch.mo Prof. Zorzi Michele

LAUREANDO: *Szabo Karoly Albert*

Padova, 11 aprile 2014

Indice

Abstract

The world of wireless communications is growing quickly, besides, all the needs coming from mobile devices are not completely fulfilled by infrastructure networks. On the contrary, Ad-Hoc networks, and in particular Mobile Ad-Hoc Networks, fit with the needs of mobility and dynamic reconfiguration, as nodes can join or leave the network freely. As a MANET scale the throughput can be afflicted, so it becomes essential to achieve optimal global performances.

Make intelligent decisions, in a continuous network adaptation, based on the environment and the evolution of the network, is essential to achieve optimal network-wide performances. Thanks to the wide view that they offer through multiple network layers and the network structure, Cognitive Networks may be the solution. The cognition layer have to deal with the analysis of in-stack and out-stack parameters coming from any device and learn the relationship among them.

In the literature many studies has been conduced in this direction. This work present the definition and the development of a framework that will enable the setup, the study and the analysis of a real multi-hop network, where a Cognitive Network protocol can be developed in, exploiting results in real networks instead of simulated networks.

Capitolo 1

Introduction

In recent years the amount of mobile wireless devices increased dramatically in many contests. Tactical and civilian wireless mesh networks are growing and novel protocols has been purposed at Physical, Data-Link, Transport and Application Layers by the network community. The environment where this work has been developed are mainly static networks, but in this project more focus was put on Mobile Ad Hoc Networks (MANET). MANET is a really interesting technology as it allow rapid deployment, it doesn't need fixed infrastructure and each node can participate simultaneously as source, rely and destination. This flexibility is attractive for many applications area where a fixed network infrastructure can't be available as military applications, disaster-response situations, academic environments and inter-veichular communications.

Cognitive radio are evolving for many years, but this technology is limited by scope, layers involved and complexity. This project put more focus on a new paradigm: Cognitive Networks.

Cognitive network is a paradigm with the goal to improve network performances through adaptation of the network. Current networking technologies consider only a limited set of information, have a limited scope and the response mechanism is layered. Those limits imply that those method try to achieve only sub-optimal performances. A Cognitive network is designed to make intelligent adaptations and act in consequence to all the knowledge acquired. The network stimuli usually generate reactive adaptations, a Cognitive Network may have the ability to predict changes and act in a proactive way.

All those technology are usually developed and studied in simulated networks,

or, sometimes, in static environment. In the literature there are mainly two approaches: by Simulations, usually done via Network Simulator (NS), by testbed with experimental networks. Researchers generally use simulation to analyse system performance prior to physical [4]. NS is a very complete and realistic simulation tool. The main advantage of the simulated approach is the re-utilization of software, just by adapting old experiment software to new network scenario, the network community can test many different techniques over similar scenarios very easly Simulation is useful for evaluating protocol performance and operation. However, the lack of rigour with which it's applied threatens the credibility of the published research within the manet research community. Simulation has proven to be a valuable tool in many areas where analytical methods aren't applicable and experimentation isn't feasible.

This thesis is a second step of an existing project [12], the goal of this project is to create real environment that allow the testing and the study of cognitive networking techniques. The tool that is going to be developed must be modular and flexible, to be expanded as easily as possible and to be used in different scenarios. This testbed has to be easy to deploy and to reproduce by other researchers, and finally it must give all the necessary tool to manage the network that the experimenter want to analyse.

As the network protocols are defined, the only way found to manage, in an intelligent way, the network, is by parameters tuning of all the protocols involved through different layers. The information coming from network parameters and out-stack parameters can help to make intelligent modifications to the network, by setting up other in-stack parameters. This paradigm has already been studied and adopted in other projects [25] [29]. Out-stack parameters can be useful to infer information about the environment, the geometry of the network and the movement of each node compared to its neighbours. Some of the extracted data are GPS, gyroscope and accelerometer. Those extra information are coming from the Android framework and the device built in sensors.

This informations can be exploited to study a practical approach to a novel cognitive network protocol. For this reasons this project seems to be very promising for study of single-hop as per multi-hop networking protocols. To follow all those goals, the use of Android devices was preferred over other solutions, running over tablets as those devices meet most of the needs of this project as they

are not too expensive, commercially available, highly customizable as they run Linux and as the work is based on MANETs, those device are mobile. Android give the opportunity to develop a testbed based on a Linux environment, with a simplified interface and a lot of built-in features.

The resulting network is an Android Wireless Mesh Network (AWMN) testbed, based on the IEEE 802.11 standard. This testbed will be used to gather data from this AWMN network. Those data, obtained with many parameters combination and dynamic customization the underlying protocols, will be then analysed in the post processing phase.

This document will be structured as follow:

- A first section will describe the technologies and theory involved. It is not necessary to explain the OSI stack and other basic knowledges as this work is supposed to be exploited by the networking community;

- The second chapter is dedicated to other testbeds and projects regarding Cognitive Radios, Cognitive Networks, Mobile ad hoc networks and some analysis regarding the OLSR routing protocols. The one used for this first phase of this testbed;

- The third section will explain briefly the Android architecture and why this OS was chosen over other solutions;

- A section regarding the whole testbed environment and all the libraries exploited by the Android App developed. The underlying work made for this project were the major amount of time and effort;

- In the section regarding the Android testbed the main functionalities and the most interesting portion of code will be briefly described, mainly the ones that involve the interface between Android the the libraries described in the previous chapter;

- The results will show some data gathered during the testing of the COGNET testbed. In this chapter will be showed also how is possible to modify some behaviour and improve performances by using a multilayer approach.

- At the very end of the document the conclusion will finish with a description of the problems encountered and a definition of the planned improvements and new features for this testbed.

Capitolo 2

Technologies involved

The theoretical knowledge needed for this project mainly range over Cognitive Radio, Cognitive Networks, Mobile Ad-Hoc Networks and routing protocols. This chapter will describe the principles of each area covered before start to develop the real testbed. Briefly, each area is needed for the following reasons.

Cognitive Radio to fullfill the need optimise performance regarding spectrum occupancy;

Cognitive Networks Is the main goal of the testbed. Provide a base to develop a system able to tune network parameters to optimize performances, the study is useful also to define which parameters has to be used to make consistent decision.

MANET Mobile Ad-Hoc Networks, is the target of study of this project. Those network have a great amount of dynamic elements, mobile nodes and they don't rely on a fixed infrastructure.

Routing Protocols This approach need particular routing protocols. For the first step of the project OLSR was chosen as first algorithm being analysed

2.1 CogNet

The current networking technologies does not allow a network to adapt themselves by making intelligent decision. This result in sub-optimal performances. Network elements are separated by the layered structure of the protocol architecture, the

communication of network state information are limited by this structure, so individual nodes are unaware of network statuses of other elements. Any network input to an element can be only perceived as local, furthermore any response of the network element can be made only in his limited scope.
Network adaptations are typically reactive, this means that they occur after a problem has occurred. With a standard design is possible to state that a network can make only local decision, this because they are limited in state, scope and response mechanism.
Two mechanism are available to improve this situation:

- Cognitive Radio
- Cognitive network

Those mechanism promise to remove or at least reduce the limitations presented here. This can be achieved by letting networks observe, act, learn and optimize their performances.

2.1.1 Cognitive Radio

A cognitive radio is a mechanism that act at the lowest levels of the OSI layers. This idea has been developed driven by the demand for new wireless devices and the steadily increasing number of wireless users, that dramatically increased the demand of radio spectrum.
Dynamic Spectrum Access (DSA) is built to be used without interfering with the existing users, just by use unused bands, or "spectrum holes" whenever they are available. Cognitive radio techonology will enable a CR network to use spectrum in a dynamic manner. DSA, by definition, must be higly flexible and dynamic, whereby to implement CR, a Software defined Radio (SDR) will be employed. A CR is an SDR with the additional capability to sense its environment, track changes and react upon its findings.

The operational area of an SDR can be listes as follows [17]:

Multiband system supports more than one frequenza band at once.

Multistandard system support more than one standard and it can work within one standard family across different networks.

Multiservice system provides different services.

Multichannel system support many independent tx-rx channels at the same time.

Multimode system both multiband and multistandard systems

Mitola and Maguire [21] put much effort on the Radio Knowledge Representation Language (RKRL) and how this new technology can enhance the flexibility of personal wireless services. RKRL supports the cognition cycle as in figure 2.1. The mechanism principle is that external world provide stimuli, CR parse these stimuli to extract the available contextual cues necessary for the performance of its assigned tasks. CR have the objective of optimize the exploitation of the

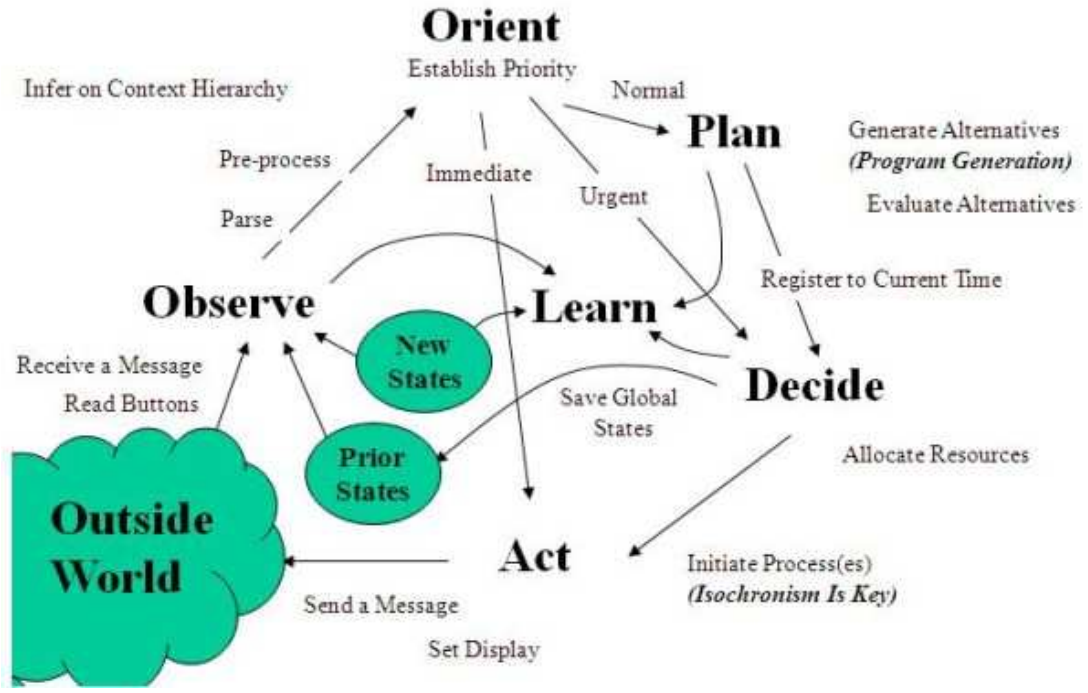

Figura 2.1: Cognition cycle for CR networks

available spectrum and they are able to do so by reconfiguring themselves using their cognivite capabilities. Formally, in fact, a CR is defined as a radio able to change its transmitter parameters based on interaction with its environment (Federal Communication Commission definition) [10].

We will briefly decribe those 2 main characteristics of CR [16]:

Cognitive Capabilities through real time interaction with the radio environment, spectrum holes at a specific time and location can be identified and classified as shown in figure SPECTRUM HOLES FIGURE

- black holes
- grey holes
- white holes

Reconfigurability according to the spectrum characteristics many parameters can be tuned in order to let the transmitter and receiver be switched to a different spectrum band.

CRAHN

The Cognitive Radio field have the goal to improve the optimization of the exploitation of the network spectrum, in order to exploit any vacant portion of the radio spectrum without overlapping signals and communications.

A Cognitive Radio Ad Hoc Network is basically an Ad-Hoc Network where some Cognitive process is associated to the radio antennas. It does not have a centralized entity for obtaining the needed spectrum usage information in the neighborhood, neither have the support of external entities in the formation of a spectrum broker that enables the sharing of the available spectrum resources.

2.1.2 Cognitive Network

This technology try to extend the concept of a Cognitive Radio through all the network layers, the expected result is to improve the network perfomances and throughput not only locally, but widely and in a deeper way. The demand for this new approach in networking came from the need of stable and reliable networks, with the maximum performance available. The current technology grant local optimal decisions, but they can't consider the whole shema of a network and its evolution. This approach can be very useful in both civilian and tactica or hostile schemas, everytime a stable network is needed.
A Cognitive Network [28] has to be able to learn and react to changes in the environment, possibly predicting problems before they occur, and have some self healing capabilities.

The architecture must be distributed through all the network layers, in order to take advantage of the OSI layering also in the cognition process. Isolated information may be easier to interpret than the whole stack at once.

Two main points that define a cognitive network are:

- Cognition: Some kind of interpretation of the observed datas coming from the whole network. Those data are gathred through all the network layers and they help the Cognitive netowrk to make adaptations. The cognition process evolve its perception of the network as it enlarge its 'knowledge' of the environment;
- End-to-end goals: this point is critical too, each goal is no more layer-oriented but here the optimizatio must consider all the layers

.

2.1.3 Comparison with Cognitive Radio

CN and CR have many similarities but also some differences, here will be listed the most important of them here:

Similarities those archietcures have some common goals:

- Cognitive Process: they have a goal, usually improve the performances of the communication, and to do so learn from the environment and reason over the gathered data is a key feature;
- Tunable Parameters: Software Defined Radio and Software Adaptable Network;
- Cross layer aspects: Both the systems does not focus only on one layer, but they take into consideration more than one layer at a time

Differences As the two ideas try toi cover different aspects, some differences are intrinsic:

- Scope: Cognitive Network want to reach end-to-end goals, they have a wider view of the network; Cognitive Radio on the other hand have the point-to-point optimization, in order to maximize the bandwidth locally;

- Distributed optimization of the Cognitive Network instead of the indipendent node by node process of the Cognitive Radio;
- Cooperative and Selfish behaviour must be considered, in Cognitive Network some node can act in a cooperative way and some other in selfish way;
- Complexity: as the CN process involves an increased number of states, interactions and conflicting goals between all the nodes, the complexity increase a lot compared to the CRs.

2.2 MANET

A MANET [11] is a wireless network where the communicating nodes are mobile and the network topology is continuously changing. It stands for "Mobile Ad Hoc Network." A MANET is a type of ad hoc network that can change locations and configure itself on the fly. Because MANETS are mobile, they use wireless connections to connect to various networks. This can be a standard Wi-Fi connection, or another medium, such as a cellular or satellite transmission. Some MANETs are restricted to a local area of wireless devices (such as a group of laptop computers), while others may be connected to the Internet. For example, A VANET (Vehicular Ad Hoc Network), is a type of MANET that allows vehicles to communicate with roadside equipment. Because of the dynamic nature of MANETs, they are typically not very secure, so it is important to be cautious what data is sent over a MANET.

2.3 Multihop routing protocols

There are mainly two approach for Ad-Hoc network for performing routing: Reactive and proactive, plus an hybrid solution and Hierarcical networks.
We will desecribe here the main protocols currently in use and, in particular OLSR that was our choice for the first step of our studies.

Reactive A reactive approach is extremely lightweight

- AODV (Ad hoc On-Demand Distance Vector Routing),
- DSR (Dynamic Source Routing),

- Flow State in the Dynamic Source Routing

Proactive The proactive approach on the other hand offer a more stable

- OLSR (Optimized Link State Routing),
- BATMAN (Better Approach To Ad Hoc Networking),
- DSDV Destination Sequence Distance Vector,

Hybrids They try to obtain the best aspects of both the other methoss. ZRP Zone Routing Protocol

Hierarchical
- CBRP (Cluster Based Routing Protocol) and
- FSR (Fisheye State Routing protocol)

2.3.1 AODV

The Ad hoc On-Demand Distance Vector (AODV) routing protocol is intended for use by mobile nodes in an ad hoc network. It offers 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. It uses destination sequence numbers to ensure loop freedom at all times (even in the face of anomalous delivery of routing control messages), avoiding problems (such as "counting to infinity") associated with classical distance vector protocols.

2.3.2 ZRP

The Zone Routing Protocol (ZRP) framework is a hybrid routing framework suitable for a wide variety of mobile ad-hoc networks, especially those with large network spans and diverse mobility patterns. Each node proactively maintains routes within a local region (referred to as the routing zone). Knowledge of the routing zone topology is leveraged by the ZRP to improve the efficiency of a globally reactive route query/reply mechanism. The proactive maintenance of routing zones also helps improve the quality of discovered routes, by making them more robust to changes in network topology. The ZRP can be configured for a particular network by proper selection of a single parameter, the routing zone radius.

2.3.3 OLSR

Is a proactive protocol, this means that is a table driven protocol. Routes are stored in tables and when needed the route is immediately presented by olsr without delay. In order to reduce the flooding overhead, some nodes get the role of multipoint reles (MPRs) and they became responsible to forward broadcast packets during the flooding process.
OLSR [9] performs hop by hop routing. MPRs are made in a way that covers all nodes two hop away. MPR's are sensed and selected by all nodes thanks to Hello messages. Topology messages are used by the nodes to determine it's MPRs. The Optimized Link State Routing Protocol (OLSR) is developed with the aim of becoming one of the standards for mobile ad hoc networks. As specified, it operates as a table driven, proactive protocol, so it exchanges topology information with other nodes of the network regularly. The MPR nodes are a set of neighbour nodes selected by each node, they are called Multipoint Relays. This protocol is developed so it can work with no information about the other layers, in particular about the underlying link-layer, so is totally indipendent from other protocols. OLSR inherits the concept of forwarding and relaying from HIPERLAN (a MAC layer protocol).
Other protocols should be mentioned as proactive, beyond all of them BATMAN must be cited. It was written to succeed to OLSR, as it came from similar basis but different and less complex algorithm.

MPR nodes

In OLSR, only nodes, selected as such MPRs, are responsible for forwarding control traffic, intended for diffusion into the entire network by forwarding and flooding all the nodes. MPRs provide an efficient mechanism for flooding control traffic by reducing the number of transmissions required. Those nodes have special responsabilities regarding the link state information of the network. As the only requirement for OLSR to provide the shortest path routes to all destinations is that MPR nodes declare link state information for their MPR selectors. For redundancy additional link-state information may be used.
MPR selected nodes announce this information periodically in their TC messages, thereby a node announces to the network, that it has reachability to the nodes which have selected it as their MPR node. Those nodea are used by all the other

nodes to create their routing table, so to form the route from a given node to any destination in the network. Furthermore, the protocol uses MPRs to enhance the efficiency of the flooding system for any other Control messages in the network. A node selects its MPRs from the set of 1-hop Neighbours with symmetric (or bidirectional) linkages.
Only nodes with willingness different than WILL_NEVER may be considered as MPR. Therefore, selecting the route through MPRs automatically avoids the problems associated with data packet transfer over uni-directional links (such as the problem of not getting link-layer acknowledgments for data packets at each hop, for link-layers employing this technique for unicast traffic).

Schema

The schema in figure 2.2 represent the flow for each node partecipating in the OLSR MANET network. Three portions reprsent the incoming messages, the outgoing messages to all or to a subset of the nodes and the internal processing that create all the structures regarding the network status. The last portion is the routes and their calculation, the most importat section but is made thanks to the other three sections of OLSR.

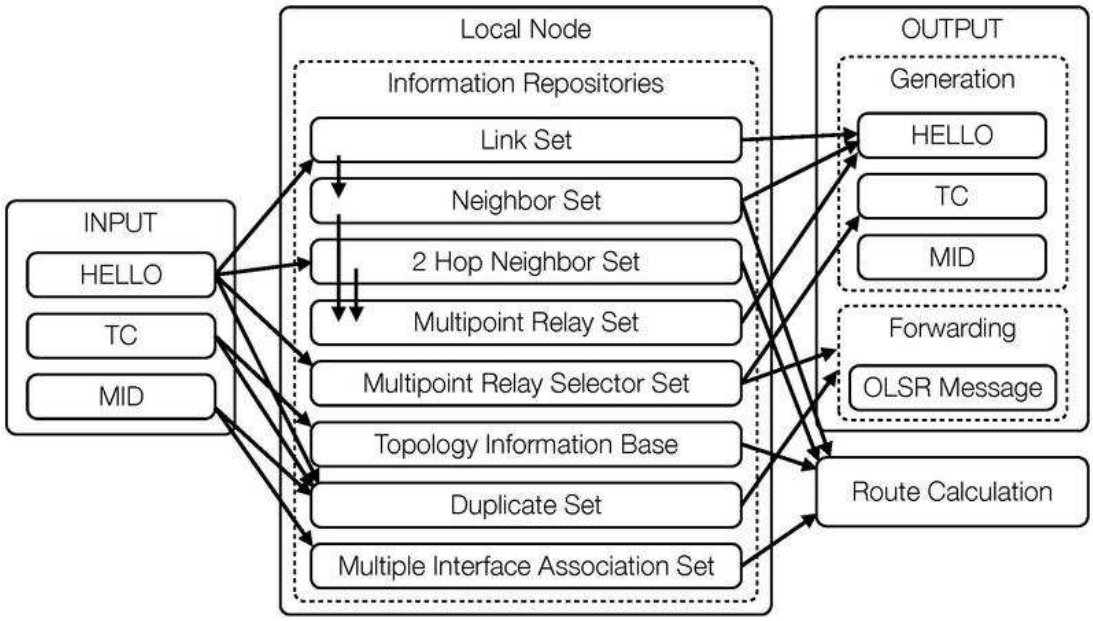

Figura 2.2: OLSR flow and table generation

As is possible to see there are Three type of messages:

HELLO Hello messages: A Hello message is sent periodically to all of a node's neighbors. They contain information about a node's neighbors, the nodes it has chosen as MPRs, and a list of list of neighbors whom bidirectional links have not yet been confirmed.

TC Topology Control messages: Every node periodically floods, using the multipoint relying mechanism, the network with a TC message. This message contains the node's MPRSelector set.

MID Multiple Interface Declaration messages: A MID message is used for announcing that a node is running OLSR on more than one interface. The MID message is flooded throughout the network by the MPRs.

Plus the HNA messages that only nodes with non-OLSR networks can send in order to act as Bridge between the two networks.

Control Messages The control messages used by OLSR are mainly the three that will be explained in this paragraphs. All those messages are sent regularly by all the nodes to all their neighbours or flooded to all the network. A timer decide when the message has to be generated and sent, those timers are configured via the OLSR configuration files. The link sensing and neighbor detection part of the protocol basically offers, to each node, a list of neighbors with which it can communicate directly and, in combination with the Packet Format and Forwarding part, an optimized flooding mechanism through MPRs. Based on this, topology information is disseminated through the network.

Hello messages OLSR needs some mechanism to detect neighbors and the state of the communication lines to them. HELLO messages are emitted on a regular interval for this purpose. When two nodes A and B try to hello each other this is the schema that they will follow:

1. A first sends an empty HELLO message.

2. B receives this message and registers A as an asymmetric neighbor due to the fact that B can not find its own address in the HELLO message.

3. B then sends a HELLO declaring A as an asymmetric neighbor.

4. When A receives this message it finds its own address in it and therefore sets B as a symmetric neighbor.

5. This time A includes B in the HELLO it sends, and B registeres A as a symmetric neighbor upon reception of the HELLO message.

HELLO messages must never be forwarded, to avoid flooding and confusion.

This process involves transmitting the Link Set, the Neighbor Set and the MPR Set. In principle, a HELLO message serves four independent tasks:

- link sensing,
- neighbor detection,
- 2-hop neighbour sensing,
- MPR selection signaling.

Those tasks are all are based on periodic information exchange within a nodes neighborhood, and serve the common purpose of "local topology discovery". A HELLO message is therefore generated based on the information stored in the Local Link Set, the Neighbor Set and the MPR Set from the local link information base.
A node must perform link sensing on each interface, in order to detect links between the interface and neighbor interfaces. Furthermore, a node must advertise its entire symmetric 1-hop neighborhood on each interface in order to perform neighbor detection.

Topology Control Messages The present section describes which part of the information given by the link sensing and neighbor detection is disseminated to the entire network and how it is used to construct routes.
Routes are constructed through advertised links and links with neighbors. A node must at least disseminate links between itself and the nodes in its MPR-selector set, in order to provide sufficient information to enable routing.
Topology Control messages are diffused with the purpose of providing each node in the network with sufficient link-state information to allow route calculation.Each node in the network maintains topology information about the network. This information is acquired from TC-messages and is used for routing table calculations. In a node, the set of Topology Tuples are denoted the "Topology Set". In

order to build the topology information base, each node, which has been selected as MPR, broadcasts Topology Control (TC) messages. TC messages are flooded to all nodes in the network and take advantage of MPRs. MPRs enable a better scalability in the distribution of topology information.

Multiple Interface Declaration messages For single OLSR interface nodes, the relationship between an OLSR interface address and the corresponding main address is trivial: the main address is the OLSR interface address. For multiple OLSR interface nodes, the relationship between OLSR interface addresses and main addresses is defined through the exchange of Multiple Interface Declaration (MID) messages. Each node that have only one interface will never generate any MID message, on the other hand is mandatory for all the nodes with 2 or more interface in OLSR MANET networks, must send those messages on regular basis. A MID message may not contain all the topology information about his networks, due to limitations on the packet size, but it must send the whole information divided in sequential packets. Similarly to TC messages, MID messages are flooded and forwarded only by MPRs.

Routing table Each node build and maintains a routing table which allows it to route data, destined for the other nodes in the network. The routing table is based on the information contained in the local link information base and the topology set. Therefore, if any of these sets are changed, the routing table is recalculated to update the route information about each destination in the network.
Each entry in the table consists of:

- dest_addr
- next_addr
- dist
- iface_addr

Those entries describe the current situation for each node, in particular the field 'dest_addr' is estimated to be 'dist' hops away from the local node (the node that are being analyzed). It indicates also that the symmetric neighbour with

interface address 'next_addr' is the next node in the route from the local node to 'dest_addr', furthermore to reach this route the right interface to be used in the local node must be iface_addr. Entries are recorded in the routing table for each destination in the network for which a route is known. All the destinations, for which a route is broken or only partially known, are not recorded in the table. More precisely, the routing table is updated when a change is detected in either:

- the link set,
- the neighbor set,
- the 2-hop neighbour set,
- the topology set,
- the Multiple Interface Association Information Base,

More precisely, the routing table is recalculated in case of neighbor appearance or loss, when a 2-hop tuple is created or removed, when a topology tuple is created or removed or when multiple interface association information changes. The update of this routing information does not generate or trigger any messages to be transmitted, neither in the network, nor in the 1-hop neighborhood.

Networks Bridging Some nodes may be equipped with multiple interface, and some of them may be non-OLSR interfaces, in this case they do not participate in the OLSR MANET. These non OLSR interfaces may be point to point connections to other singular hosts or may connect to separate networks. If the OLSR network have to be able to access those external networks, the protocol must consider also routes that consider this possibility.
To do so, a node with both kind of interface must shuld be able to make them interact by adding connectivity from the OLSR MANET interface(s) to these non OLSR interface(s), by injecting external route information to the OLSR MANET. An example should be our testbed, where all the nodes are connected through the primary wireless interface to our DEI network, whereas the secondary interface is connected to our OLSR MANET network. In this situation can be useful to have a connection between the OLSR MANET and a storage server for logging purpose. This ability of the protocol can enable a node to do so. Notice that this

is a different case from that of "multiple interfaces", where all the interfaces are participating in the MANET hrough running the OLSR protocol.

Host and Network Association Messages The external network bridging capability can be provided thanks to this messsages, they enable nodes to inject external routing information into an OLSR MANET, a node with such non-MANET interfaces periodically issues a Host and Network Association (HNA) message, containing sufficient information for the recipients to construct an appropriate routing table. Both HNA messages and TC-messages announce reachability to other host(s), so is possible to consider the HNA message a generalized TC message. The main differences are:

- Netmask is not required in TC messages since all reachability is announced on a per-host basis, on the other hand in HNA messages the reachability is typically announced through network and netmask address instead of individual host addresses.
- a TC node may have canceling effect on previous information, if ANSN is incremented, whereas HNA messages informations will be removed only upon expiration.

this messages should be generated only by nodes with associated hosts and/or networks. HNA message, containing pairs of (network address, netmask) corresponding to the connected hosts and networks. This messages are generated periodically, every HNA_INTERVAL. The Vtime is set accordingly to the value of HNA_HOLD_TIME and it indicates the amount of time before the message is considered no more valid. Every node without any non-OLSR interface should not generate HNA messages.

Capitolo 3

Related works

Many other works were conducted or are still running in this area of study. We will describe here the most intersting and the most related to our work in order to have a wide view of the current state of the art in other places [8].

Many of the studies we will present here are based on virtualised networks, running over Network Simulator (NS). A tool to generate networks that provide, mostly, very reliable and realistic results in most cases. By using such a tool is relatively easy to re-utilize most of the software implemented in previous experiments and adapt it to the new needs, or different scenarios. The drawback is there is no simulation that can be as accurate as a real network scenario, with problems related to connectivity, memory, performances etc..

3.1 Ad Hoc networks

First of all we will describe some Ad Hoc routing experiments, some of them with a Mobile orientation, the most related to our work are:

Autonomous Robot exploration in smart environment [5] exploiting wireless sensors and visual features to obtain information about the environment: For our purpose some part of this paper were very useful, first of all the information about how RSSi change during the movement of a node relative to other nodes in the network and secondly the triangularization approach that can be used also in a Cognitive Network approach to infer information about the network structure;

Roofnet [2] Developed by the Wireless laboratories at MIT, Roofnet is a network of about 50 nodes completely indipendent. It have the goal to prove the feasibility of self-configuring network with no planning or organization behind, each node can easily join or leave the network without affect performances; Some paper studied the performances of this project [7] and proved that, it has good performance despite lack of planning: the average inter-node throughput is 88 kilobytes/second, even though the average inter-node route is three hops. There is also a valuable comparison between Roofnet's performance and the performance of an access-point network using the same set of nodes.

End to End goals without coordination [18] This work have the goal of achieve end to end fairness in Multi-hop networks without coordination between nodes.

3.2 Cognitive Networks and Cognitive Radios

Secondly there are some studies amont Cognitive Radio, and very few regarding Cognitive Networks. Among those studies we looked at:

CRABSS [19] CalRAdio-Based advanced Spectrum Scanner for cognitive networks. This project have the aim to study the cognition layer in a radio environment. It used a modular approach with CalRadio, a tool with software definable radio antennas;

VT-CoRNET [23] Virginia Tech Cognitive Network. This study started with an emulated network, Emulab [22]. The Cognitive Radio Network Testbed (CORNET) is a collection of 48 software-defined radio nodes deployed within a four-story building. All the infrasctructure is spread into four floors and is based on static nodes. Many projects are living and use this infrastructure:

- Open-Source LTE (video)
- Spectrum Sensing
- Spectrum Access
- Wireless Distributed Computing

- Position Location
- Cognitive Engines
- Ad-Hoc/Mesh networking
- Cognitive Jamming/Electronic warfare
- SDR Frameworks and Tools

ORBIT [26] Open Access Radio Testbed for next generation wireless network. This testbed is centered around the "radio grid emulator" which provides facilities for reproducible networking experiments with large numbers ($\tilde{1}00$'s) of wireless nodes. The whole tesbed count 400 static nodes, all of them programmable and able to exploid next-generation networking protocols and applications. It is used to experiment

NeXt or DSA radios survey [3] This study present the situation a new netowrk paradigm refferred to DSA (Dynamic Spectrum Access) as well as xG (Next Generation Network). The xG network functions such as spectrum management, spectrum mobility and spectrum sharing are explained in detail. In this work are well explained also the motivation behind all this projects regarding Cognitive Networks and Mobile Wireless networks.

FIT CortexLab FIT (Future Internet of Things) aims to develop an experimental facility, a federated and competitive infrastructure with international visibility and a broad panel of customers. This project is divided into four area: a Network Operations Center (NOC), a set of Embedded Communicating Object (ECO) testbeds, a set of wireless OneLab testbeds, and a cognitive radio testbed (CorteXlab). This testbed is mainly oriented in the development and analysis of Cognitive Radios [20].

EECRT Helsinnki EECRT project creates a "living lab" cognitive radio testbed. The test-bed is intended for investigating Dynamic Spectrum & bandwidth Management (DSM) and Cognitive Radio Resource Management(CRRM) algorithms, Radio usage business models, Radio interface selection algorithms with the focus on end-to-end performance and Cognitive Radio algorithms for physical layer usage. The test-bed provides a software platform where new algorithms can be implemented and tested in a real radio environment. One part of the test-bed a Cognitive Radio network that impl-

ments TDD-LTE like radio interface in software. Over the air transmission is implemented by using remote radio unit in combination with the general purpose computer. The network contains base stations and user equipments. All the baseband processing is implemented in software and runs in the computer. EECRT project is conducted by Aalto University's Department of Communications and Networking (Comnet)

Iris testbed [27] Iris is a software architecture for building highly reconfigurable radio networks. This software was the base for a wide range of dynamic spectrum access and cognitive radio demonstration systems presented at a number of international conferences. Iris can be used in heterogeneous processing platforms and the project that exploited Iris were developed over different platforms, including general-purpose processors, field-programmable gate arrays and the Cell Broadband Engine. Focusing on runtime reconfiguration, Iris offers support for all layers of the network stack and provides a platform for the development of not only reconfigurable point-to-point radio links but complete networks of cognitive radios.

3.3 OLSR analysis

The last field involved directly with this project until this moment regard OLSR, and other multihop routing protocols. One of our goal is to proof that a correct parameter modification, can improve the network overall performances:

OLSRd tuning based study [15] This study show how the behaviour of OLSR change by tuning some parameters. This demonstration support our concept that is possible to improve network performances by parameters tuning based on the current network situation. If a Cognitive process will be able to infer the future situation of the network by interpreting all the stimuli coming from the environment, a correct dynamic tuning will reduce overhead without loosing reactivity of the network.

Comparison: BATMAN vs. OLSRd [6] We were interested in this analysis as the aim of our project is to enable and test different routing protocols in our MANET. the comparison of the two protocols under different conditions

made possible to evaluate in which contest is better to use OLSR and where BATMAN perform better.

3.4 Contribution of this project

In the literature and in the current works found, it was not possible to find another, available and working, testbed for real Mobile Ad-hoc Networks. The contribution that this work may have can be summarized in those points:

- The development of an heterogeneous mesh network with static nodes Linux based, some ALIX boards and Android based nodes to test mobility;
- Develop a tool to setup, manage, control and monitor experiments both in real time and for post process datas;
- Perform experiments in this environment with different protocols with mobile nodes involved;
- A study on the extraction and the modification of TCP and MAC layer parameters from Android devices;

Capitolo 4

Android

The above figure shows the diagram of Android Architecture. The Android OS can be referred to as a software stack of different layers, where each layer is a group of several program components. Together it includes operating system, middle-ware and important applications. Each layer in the architecture provides different services to the layer just above it.
We will examine the features of each layer.

4.1 Linux Kernel

The basic layer is the Linux kernel. The whole Android OS is built on top of the Linux Kernel with some further architectural changes made by Google.
The Linux kernel also acts as an abstraction layer between the hardware and other software layers. Android uses the Linux for all its core functionality such as Memory management, process management, networking, security settings etc. As the Android is built on a most popular and proven foundation, it made the porting of Android to variety of hardware, a relatively painless task.

4.2 Libraries

The next layer is the Android's native libraries. It is this layer that enables the device to handle different types of data. These libraries are written in c or c++ language and are specific for a particular hardware.
Some of the important native libraries include the following:

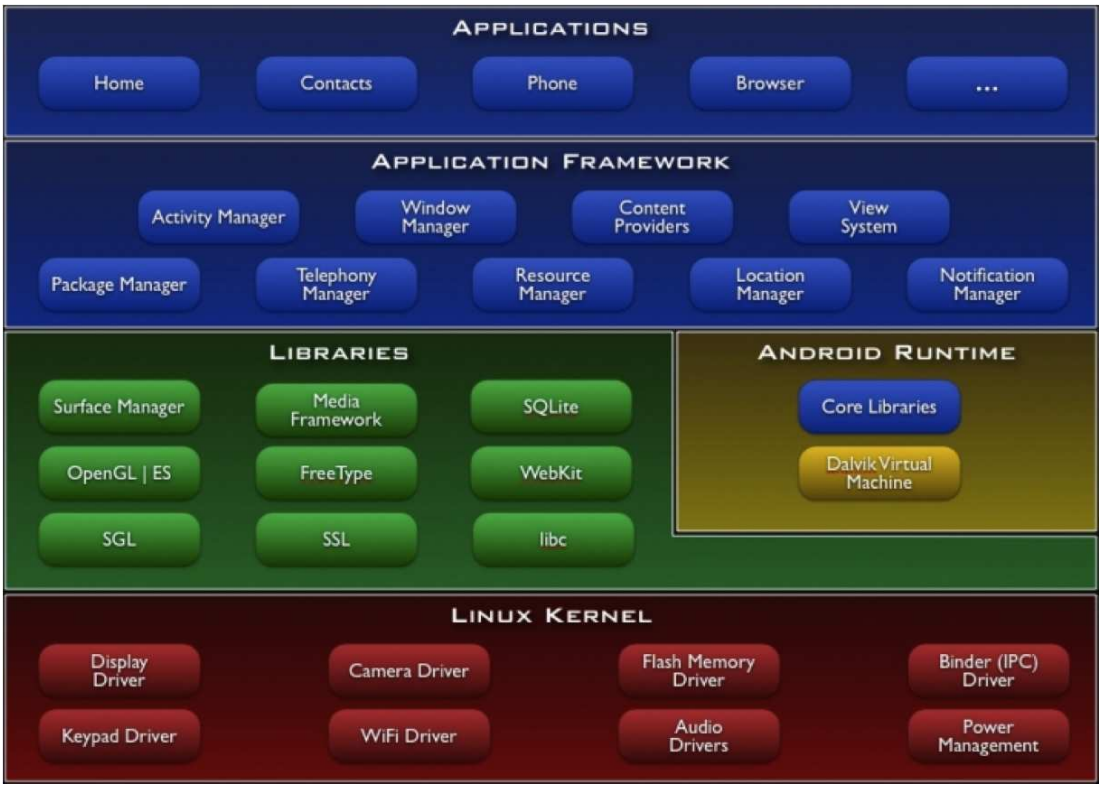

Figura 4.1: The Android architecture schema

- **Surface Manage**: It is used for compositing window manager with off-screen buffering. Off-screen buffering means you cant directly draw into the screen, but your drawings go to the off-screen buffer. There it is combined with other drawings and form the final screen the user will see. This off screen buffer is the reason behind the transparency of windows.

- **Media framework**: Media framework provides different media codecs allowing the recording and playback of different media formats.

- **SQLite**: SQLite is the database engine used in android for data storage purposes

- **WebKit**: It is the browser engine used to display HTML content

- **OpenGL**: Used to render 2D or 3D graphics content to the screen

4.3 Android Runtime

Android Runtime consists of Dalvik Virtual machine and Core Java libraries.

4.3.1 Dalvik Virtual Machine

It is a type of JVM used in android devices to run apps and is optimized for low processing power and low memory environments. Unlike the JVM, the Dalvik Virtual Machine [14] doesn't run .class files, instead it runs .dex files.
.dex files are built from .class file at the time of compilation and provides higher efficiency in low resource environments. The Dalvik VM allows multiple instance of Virtual machine to be created simultaneously providing security, isolation, memory management and threading support. It is developed by Dan Bornstein of Google.

4.3.2 Core Java Libraries

These are different from Java SE and Java ME libraries. However these libraries provides most of the functionalities defined in the Java SE libraries.

4.4 Application Framework

The android framework is the set of API's that allow developers to quickly and easily write apps for Android devices. It consists of tools for designing UIs (buttons, text fields, image panes) and system tools (intents, phone controls, media players, ect).

By default all Activities and Services run in a single process and in a single Thread (main UI Thread). It is possible to declare in your manifest file a separate process in which place a certain component.
The only exception is the code that handles IPC calls coming in from other processes. The system maintains a separate pool of transaction threads in each process to dispatch all incoming IPC calls. The developer should create separate threads for any long-running code, to avoid blocking the main UI thread.
Important blocks of Application framework are:

- **Activity Manager**: Manages the activity life cycle of applications.
- **Content Providers**: Manage the data sharing between applications.
- **Telephony Manager**: Manages all voice calls. We use telephony manager if we want to access voice calls in our application.
- **Location Manager**: Location management, using GPS or cell tower.
- **Resource Manager**: Manage the various types of resources we use in our Application.

4.5 Applications

Applications are the top layer in the Android architecture and this is where our applications are gonna fit. Essentially an android app consists of:

- **Activities**: it represent a single screen with an UI;
- **Services**: a component that runs in background; usually it perform long-running operations or work for remote processes. It has no UI;
- **Content providers**: manage a shared set of app data. Data can be stored in:
  - File system,
  - an SQLite database,
  - any other persistent location accessible by the app.

  is activated by a Content Resolver object.
- **Broadcast receivers**: is a component that responds to system-wide broadcast announcements (the battery level is low, a picture was captured, etc..).

Activities, Services and Broadcast receivers are activated by an asynchronous message called **Intent**.

4.6 Developement Kit

The environment where most of the Android sofware is developed is the Android SDK. The Development process usually follow some standard steps and a standard schema, usually purposed by the exploited IDE. Applications are usually developed in the Java programming language using the Android Software Development Kit, but other development tools are available.

4.6.1 Android SDK

A software development kit that enables developers to create applications for the Android platform. The Android SDK includes sample projects with source code, development tools, an emulator, and required libraries to build Android applications.

This tool is the most used to develop any Android application or tool. The SDK includes a comprehensive set of development tools. These include a debugger, libraries, a handset emulator based on QEMU, documentation, sample code, and tutorials. Currently supported development platforms include computers running Linux (any modern desktop Linux distribution), Mac OS X 10.5.8 or later, Windows XP or later; for the moment one can develop Android software on Android itself by using [AIDE - Android IDE - Java, C++] app and [Android java editor] app. The officially supported integrated development environment (IDE) is Eclipse using the Android Development Tools (ADT) Plug-in, though IntelliJ IDEA IDE (all editions) fully supports Android development out of the box, and NetBeans IDE also supports Android development via plug-ins. Additionally, developers may use any text editor to edit Java and XML files, then use command line tools (Java Development Kit and Apache Ant are required) to create, build and debug Android applications as well as control attached Android devices (e.g., triggering a reboot, installing software package(s) remotely).

4.6.2 NDK

The NDK is a tool used in particular cases:

- when some tasks cannot be solved by using only high level instructions;

- to exploit external compiled libraries written in other languages than Java;
- when the application need an interface with more low-level structures;

The NDK is not very well known as only a real small portion of the existing apps need to solve tasks that imply the use of C libraries. Libraries written in C and other languages can be compiled to ARM, MIPS or x86 native code and installed using the Android Native Development Kit. Native classes can be called from Java code running under the Dalvik VM using the System.loadLibrary call, which is part of the standard Android Java classes.

Complete applications can be compiled and installed using traditional development tools. The ADB debugger gives a root shell under the Android Emulator which allows native ARM, MIPS or x86 code to be uploaded and executed. Native code can be compiled using GCC on a standard PC. Running native code is complicated by Android's use of a non-standard C library (libc, known as Bionic). The graphics library that Android uses to arbitrate and control access to this device is called the Skia Graphics Library (SGL), and it has been released under an open source licence. Skia has backends for both Win32 and Unix, allowing the development of cross-platform applications, and it is the graphics engine underlying the Google Chrome web browser.
Unlike Java application development based on the Eclipse IDE, the NDK is based on command-line tools and requires invoking them manually to build, deploy and debug the apps. Several third-party tools allow integrating the NDK into Eclipse[20] and Visual Studio

Using the NDK

The Android framework provides two ways to use native code:

JNI libraries Write your application using the Android framework and use JNI to access the APIs provided by the Android NDK. This technique allows you to take advantage of the convenience of the Android framework, but still allows you to write native code when necessary. If you use this approach, your application must target specific, minimum Android platform levels, see Android platform compatibility for more information.

Native activities Write a native activity, which allows you to implement the lifecycle callbacks in native code. The Android SDK provides the NativeActivity class, which is a convenience class that notifies your native code of any activity lifecycle callbacks (onCreate(), onPause(), onResume(), etc). You can implement the callbacks in your native code to handle these events when they occur. Applications that use native activities must be run on Android 2.3 (API Level 9) or later.

You cannot access features such as Services and Content Providers natively, so if you want to use them or any other framework API, you can still write JNI code to do so.

4.7 Advantages for our purpose

Unlike many competitors, Android is built upon an open-source platform, and most of the Android code is released under the free software/open source Apache License. For developers who want to write very specialized applications, particularly applications which depend on functionality for which there are no libraries, this is a major advantage over other systems. Android applications are written in the Java programming language, which is a powerful, mature and very widely adopted language in the global development community. Android is a framework that live in a Linux environment. In this situation the Linux kernel can be expanded and/or modified, and by using the NDK is possible to exploit any kernel module or external library. Android OS is divided into layers and each layer provide different functionalities to the next one. The Application layer is the environment where an app can run, and where we will build our testbed application to manage the network and to see the behaviour in real time, Linux Kernel is the lowest layer, where all the modules that interact with the TCP/IP stack will run and cooperate with the our testbed by observing and modifying network parameters.

Capitolo 5

COGNET Testbed Environment

Many experiments was already conducted in this area, but found out that all of them was simulations or using static nodes. The aim of this project was to build an infrastructure to create a real testing environment with both mobile and static nodes. This app has to be configurable on-the-fly and show to the experimenter an immediate visual feedback of some network parameters and behaviour during the experiments. All the following modules were taking care of the multiple platforms where they have to live, except the Android app that cannot live in any other context. To cross-compile is to build on one platform a binary that will run on another platform. When speaking of cross-compilation, it is important to distinguish between the build platform on which the compilation is performed, and the host platform on which the resulting executable is expected to run. In our case all the libraries were compiled into a Linux machine 64bit with all the libraries to compile for Android, Linux and ALIX.

5.1 Setup

The very first phase before the real development was defining the environment where everything will be developed. In this section will described the major aspects that was defined in the project early stage.

5.1.1 Double network devices

Any Android device have a WiFi interface. This interface can be different from device to device and its drivers too. Furthermore those drivers usually are not open-source and neither editable, so is hard to inject code to develop a proper testbed. Due to those assumptions, in order to realise a proper testbed, a workaround was required. Three main reason drove us to the extension of the classical devices into a double WIFI device.

Ad-Hoc mode The first one was that the built-in driver for the Network interface in lot of devices does not allow Ad-hoc mode on Android devices, and that was a mandatory point of the project. There are mainly two way to let a device communicate in a wireless manner:

BSS Base Service Set is the standard way to communicate, there is a central node (the access point) that manage all the other connected node in a star topology.

IBSS Indipendent Base Service Set is known as Ad Hoc mode. In this mode each node manage its own connection without the need of a central controller, in this case the topology is a mesh that allow multi-hop communications.

Efficency Deploy the whole project through all the nodes was not practical to connect via USB cable one by one all of them. ADB [REF] via WiFi was a much more clean and efficient solution for long time project like this. But to do so different Network interfaces were used for for testing and for deploying.

COGNET Module This module is critical to the success of the project: it must be injected at kernel level and make it run on a stable way in order to collect all the needed data and setup parameters like congestion window.

By default the IBSS mode is disabled in all Android recent Android devices, this mode can be activated in two ways, depending on the device:

Android Framework by using the iw tool to force a connection to an ad hoc network

Linux Kernel by modifying the driver source to activate the ad-hoc network mode. This solution was used for the Nexus 7 with BCM4330 chipset and bcmdhd driver with the following modifications:

```
--> drivers/net/wireless/bcmdhd/wl_cfg80211.
- BIT(NL80211_IFTYPE_STATION)
- BIT(NL80211_IFTYPE_STATION) OR BIT(NL80211_IFTYPE_ADHOC)
```

5.1.2 Device to use

In order to provide a real mobile environment some fixed nodes, but for the sake of our project some mobile devices were mandatory. To choose the proper device a few aspects were considered:

- Operating system: this was the most important feature, an Andorid device were needed.
- Screen size: for the purpose of the project have a big screen was very helpful, due to the big amount of data do be displayed on charts and parameters managed via interface.
- Performance: To let the data be collected properly the testbed needed at least average performance, too low could deteriorate our experiments, and too high performance has the risk of not testing properly our system.
- Customizable: As any android device it must be customizable, but to improve some features and mainly to enable some drivers a switch to an unofficial version of Android was necessary: Cyanogen.

The platform chosen to develop our testbed was Nexus 7 from google, running Android. The classical Android distribution was lacking in some points. Some unofficial mods, like Cyanogenmod allowed major modifications for drivers and kernel. Cyanogen is the most common and more experimented Android mod. This firmware is open and permit a kernel recompilation without constraints. In order to take advantage of the COGNET module with the best performance possible, we had do inject it as kernel module. We flashed this device and we installed over it the Cyanogenmod with a customized Linux Kernel with our COGNET module inside.

5.2 COGNET Module

This module is configured as service in linux kernel. It harverst lot of data, mainly his role is tcp observation and modification of lot of parameters as described in table [**?**]. In this section we will describe all the parameters that this module intend to observe and modify. Parameters of interst was found at different layers of the protocol stack, furhermore also some parameters out of stack was observed.

5.2.1 Transport Layer

The access to all the needed data at this layer is granted by a kernel module made for this purpose. This choice was made because is not possible to read and write all the needed parameters via classical systems, debugfs grant only reading access to a subset of them. This module have the aim to observe the C struct tcp_sock.
Due to Linux architecture's clear distinction between User space and Kernel space a problem emerged: to read the value of the congestion window we had to copy from kernel space to user space, then we can be able to process this data and, when the new values from the Congestion avoidance algorithm are avaialble, we have to push them back to kernel space. The need of a full duplex channel between that two processes that resides in different spaces was fullfilled by the creation of am Inter-Process Communication created with a Netlink socket. The data gathered at this layer are the followingand they are readable inside the tcp_struct whenever the TCP congetion avoidance algorithm receive an acknowledgement:

- Congestion Window (CWND): value of the congestion window for the sender;
- Slow Start Threshold (SSTHR): the SSTHR value used by the algorithm;
- Packet outstanding: transmitted and not yet acked packets;
- Packed acked: number of acked packets in the last ack packet received;

- Packets in flight: outstanding + retransmitted packets. This value depend on CWND,if is too large the amount of packet in flight will increase and increase the probability of packet loss or timeout;
- Round Trip Time (RTT): measure in millseconds the time passed from the effective sending of the packet until the sender receive the relative ack;
- Smoothed Round Trip Time (SRRT): a weighted average of the past RTT. The calculation is made with the following: SRTT;
- RTO: the retransmission timeout, usually twice the SRTT and with values from 200 milliseconds and 2 seconds. If this settings is wrong a high packet dropping ration can occur;
- MSS: Maximum segment size;
- ACKN: the last acked packet number.

5.2.2 Data Link Layer

At Data Link Layer (DLL), the implementation is related to the specific device and related driver in use. The other paramters listed in table 5.1 are available only in some specific drivers. The parameter we will list here becamed observable thanks to some hooks found where was possible to gather those values. At this layer we will observe all the parameters related to the connection at MAC level, this include, for each device:

- Number of transmitted frames;
- Number of receiver frames;
- Bytes amount Transmitted;
- Bytes amount Received;
- Transmission power;
- Transmission channel;
- RSSi.

- Bitrate

The first 4 parameters are stored in the proc files relative to the analysed interface: /proc/net/dev. The transmission channel, the transmission power and the RSSi are observed with the driver (system call) Input/Output control named ioctl. Some other parameters are useful for the estimation of the Quality of Service (QoS) of 802.11e [1]. As long as other parameters, can be read only in some specific drivers, and only a subset of them can be also written or tuned from an external source. the parameters related to the QoS are:

- Arbitrarian Inter-Frame Spacing (AIFS);
- Content Window minimum value (CWmin);
- Content Window maximum value (CWmax).

the last two parameters indicates the Content Window range. Other parameters we decided to use for our testbed purpose are:

- # Fragments TX/RX: Number of transmitted and received fragments;
- # Frames RX dropped: Number of received frames that was dropped;
- # Frames TX retry count: Number of frames retransmitted;
- # Frames TX retry failed: Number of frames whose retransmission failed.

At this layer is importar the driver and the chipset used, for our purpose we used the following couples of chipsets and relative drivers:

- Broadcom bdcm4331 chipset running with b45 driver;
- Atheros AR5413 chipset running with ath5kl driver

5.2.3 Sensors

Thanks to the Android API we were able to observe the behaviour of our tablets in the physical world. Mainly for positioning and to infere the future position of the device we decided to extract some parameters, in order to have them available in a future evolution of this project. We supposed that, thanks to this

information, mixed with RSSi and all the other network parameters, a Cognitive Network can be able to predict where the device will be and act consequentially. For instance if we have 3 nodes: A, B and C; node A is close to B but is moving in direction of C, a classical network will wait until the connection with B will be crashed before react, on the other hand a Cognitive Network, with the right informations, can be able to react in time, and avoid any packet to be lost. The data we decided to extract is the following:

- accelerometer: it measure direction and acceleration of the devices in a 3 axes representation
- gyroscope: is used to measure the orientation of the devices. This is useful mainly for have a better understanding of the accelerometer data;
- GPS: the Global Position System can be helpful if the devices will be outdoor as it provide a good aproximation of the position of the node.
- Fused Sensors: is an abstraction of gyroscope and accelerometer mixed together. This is a very powerful tool made by the Android developers in order to grant a source of information about the devices that is clean and easier to read. Noise is extremely reduced and curves are way smoother than any other raw sensor.

This portion of the project was developed initially in San Diego, California. Lot of aspects changed radically during the evolution of the project, as the ideas and the need of more information about the connection evolved too in this period of time.

5.3 OLSRd

OLSR protocol is next to AODV protocol, one of the main two internet standards for mesh networks, and mainly for Mobile Ad-Hoc networks. Those standards are widely used and well tested. For our purpose OLSR as proactive protocol was identified as easier to be configured from the cognitive part of this project.

	Layer	Parameters	Laptop b43 bdcm4331	Alix 3d2 ath5kl ar5413	Tab-7 P6210 ath6kl ar6003	Nexus 7 bdcmhd bdcm4330
Observable and Writable	TCP	CWND	rw	rw	rw	rw
	TCP	SSTHR	rw	rw	rw	rw
	TCP	RTO	rw	rw	rw	rw
	MAC	CW min	rw	rw	NO	NO
	MAc	CW max	rw	rw	NO	NO
	MAC	AIFS	rw	rw	NO	NO
	MAC	TX power	rw	rw	r	rw
Observable	TCP	IP Address	r	r	r	r
	TCP	Port	rw	rw	rw	rw
	TCP	# Packets in flight	r	r	r	r
	TCP	# Packets outstanding	r	r	r	r
	TCP	# Packets acked	r	r	r	r
	TCP	# Packets lost	r	r	r	r
	TCP	Throughput	r	r	r	r
	TCP	RTT var	r	r	r	r
	TCP	SRTT	r	r	r	r
	MAC	Transmission Channel	r	r	r	r
	MAC	RSSI var	r	r	r	r
	MAC	Bytes RX	r	r	r	r
	MAC	# Frames RX	r	r	r	r
	MAC	# Frames RX duplicate	r	r	r	r
	MAC	# Frames RX fragments	r	r	r	r
	MAC	# Frames RX dropped	r	r	r	r
	MAC	Bytes TX	r	r	r	r
	MAC	# Frames TX	r	r	r	r
	MAC	# Fragments TX	r	r	r	r
	MAC	# Frames TX retry count	r	r	r	r
	MAC	# Frames TX retry failed	r	r	r	r
	MAC	Inactive stations	r	r	r	r

Tabella 5.1: Data managed by the COGNET module in different devices. r=readable, w=writable, TX=transmitted, RX=received, #=number, nv=to be implemented in the next version

5.3.1 Impelmentation details

There are many implementations of OLSR protocol. Beyond all of them the more reliable seems to be OLSRd [**?**]. For our purpose is also convenient the fact that OLSR is released under a BSD license. It is easy to integrate into some projects thanks to this very liberal license.
The OLSR.org OLSR daemon is an implementation of the Optimized Link State Routing protocol. As described before, this routing protocol allows mesh routing, the purpose of the OLSR daemon is to be able to work for any network equipment. It runs on any wifi card that supports ad-hoc mode and of course on any ethernet device. This protocol is proactive, this means that is designed to be configured and to react to network fails before they occur. To do so is designed in a table-driven way and use a technique called Multipoint Relaying to optimize the message flooding, it implements ETT and optionally ETX to measure the quality of a link.
OLSR is running at Network Layer (layer 3) of the OSI stack, Thanks to this characteristic is highly portable and can be used through many systems, mainly is tested for the following systems:

- Windows (XP and Vista, Windows 7)
- Linux (i386, arm, alpha, mips, xscale)
- OS X (powerpc, intel, xscale, iPhone)
- VxWorks
- NetBSD
- FreeBSD
- OpenBSD
- Nokia N900
- Google phone (Android, G1)
- linux wifi phones
- the Intel Classmate

This implementation focused a lot on optimization, the result is a very fast and low consuming tool. This is vital when embedded and portable devices are involved, both for the CPU time and the valuable battery power. Thanks to his lightweight structure, OLSR is highly scalable and robust. Many project adopted this routing protocol on community mesh networks, we list here the major projects for size and importance in the development and analysis of OLSR:

- Athens Wireless Metropolitan Network (AWMN): up to 5000 nodes
- Guifi, mainly Spanish, deployed all over the world: 37500 nodes
- Berlin Leipzig Freifunk (FreiFunk.net): 600 nodes
- Vienna, Graz (FunkFeuer.net): 400 nodes

Due to the size and the importance of those project we can state that, as a deployed mesh routing protocol, OLSR can be considered stable and robust and regularly stressed and tested, despite what other competing protocols say. The community on the other hand keep it up-to date and the improvements are constant.
The implementation of OLSRd is RFC3626 compliant. It respect both the core functionalities and the auxiliary charateristics of OLSR as defined in the standard OLSRd is meant to be a well structured and well coded implementation that should be easy to maintain, expand and port to other platforms. It has a very flexible plugin architecture. OLSRd supports use of loadable plugins. These can be used to to handle and generate custom packettypes to be carried by OLSRs MPR flooding scheme or for any other desired functioning.

OLSRd

This is the basic version of OLSRd, in the early stage it was part of a master thesis project for Andreas TÃ¸nnesen at UniK - University Graduate Center. The project evolved a lot during the years, mainly after Thomas Lopatic joined the OLSRd prokect as a developer fall 2004. This version is higly plugin-oriented as there is only the minimum feature directly implemented in the core libraries. Putting the hands on the code we noticed that this version

was coming from a non-standard process, in fact there is not much structured and, in some feature, we still find some bugs. In order to let it work and compile through many different systems we had to work a lot, so is not so easy to deploy a clean version on different machines, if many plugins are needed.

Problems we found The code is fragmented and just in a few hours we can see that lot of people contributed in a manner not completely standardized. Only in our version we had 19 plugins running. The main problems we had was the total absence of a tool that can change some parameters during runtime, a plugin able to communicate remotely with the OLSRd daemon was not designed to run with Android, finally, the most important problem, was the absence of the ETT calculation, instead only ETT is available. It is currently developed and mantained by Henning Rogge and Markus Kittenberger, with the support of the whole OLSRd community

OLSRd2

This new version of OLSR is rewritten from scrath. It has clearly much more structure than his predecessor and lot of feature more. This version could be able to solve the 3 problems stated before. ETT and ETX are by default available to be read, is possible to access the daemon via telnet natively and finally is available a system to modify all the parameters while the daemon is still running. After crushing into the problems of version 1 we decided to try to import all our modifications in this new implementation.
Many of our previous modifcations were not necessary in this version, that currently implement telnet plugin with parameteres tuning whithout the need of restart the daemon.

There are mainly 2 portions that form this new version. The effective OLSR daemon and the Network Node definition. the OLSR portion uses the Node section as entities to reply on each entry of the routing tables.
the node entity is strictly connected to the compiling machine, or the library specified for the compiler for the network tasks.

Problems we found Unfortunately we realized that OLSRv2 is not yet enough mature to support multiplatform deployment, in fact for Android we had lot of trubles and we weren't able to deploy it in a stable manner nor with our modifcation nor clean versions. After many requests sent to the current mantainers we couldn't, together with them, find any immediate solution that allowed us to perform valid tests with this product, so we had to roll back to OLSRd v1, a more stable and mature version of the same algorithm, despite the fact this new version was way more reliable and well structured, lot of work have still to be done before it can be used without problems in other systems than Linux, in oour case Android.

5.3.2 Customizations

We worked with both of those systems. In the very beginning we started with OLSRd v1. We put lot of effort in this release but we noticed that was missing in many parts, like: ETX, telnet managment and some android features. During the development of our testbed, OLSRv2 reached a more mature state. We decided to spend a couple of week trying to import all our modifcations inside this new and better designed version of OLSRd. After lot of experiments we had to roll back to OLSRv1 due to the issues related to Android.
The major modifcations we made to this software are:

- Add logging to keep tables evolutions and ETT/ETX values with timestamps for post processing and comparison with other network values;
- fixed some minor bugs that caused misbehaviour in ad-hoc mode;
- added new commands to change dynamically some OLSR parameters;
- fixed OLSRd telnet plugin, in particular for Android.

5.3.3 Wireless Network Fix

The very first work that was done on OLSRd was fixing a bug regarding the way OLSRd recognize if a network device is switched on or off in case of Ad-Hoc networks. We discovered the problem by debugging the whole behaviour of OLSRd. The fix was quite simple, we needed just to remove a

check made on the network devices, that was not implemented on Android, thus any network device seems to be switched off.
When OLSRd was running no interface had both flags up and running. The bug was found in ifnet.c when it checks if the specified interface is available, in the functions:

```
chi_if_up
chi_if_changed function
```

The purposed solution is to check if the WiFi is up and running by using, in both the above functions, those conditions:

```
if((
    (ifs.int_flags & IFF_U) == 0) ||
    ((ifs.int_flags & IFF_RUNNING)==0))
```

Into:

```
if((ifs.int_flags & IFF_U) == 0)
```

This check works with ethernet because you can set ifs.int_flags with IFF_RUNNING. The same bug was found, and signaled, in OLSRdv2.

5.3.4 Logging

Our tested, in order to be complete also for post processing, had the need of logging capabilities in each aspect. Regarding OLSR we decided to store any data regarding the evolution of the routing tables in some files, and extract them later. The data extraction written for our purposes is capable of work within Android and Linux systems, and it store data in different locations depending on the system for wich it was compiled. Our data sources came from the information that OLSRd manage constantly in his table relative to:

- Neighbours
- 2-Hop Neighbous
- Links

This is the header that come before any section of the logging system, it is used to store all the files needed in a particular path, depending on the operatign

system currently in use. We decided to specify all the paths in the precompilation phase, so we will never have any chance of do something wrong if our testbed will be compilated in a certain Operating system we didn't manage. Simply, if this happen, no data will be stored.

```
  FILE *fp;
  char pathNeighbor[40];
#if defined _WIN32 || defined _WIN64
  //abuf_json_string(abuf, "os", "Windows");
#elif defined __gnu_linux__
  //abuf_json_string(abuf, "os", "GNU/Linux");
  strncpy(pathNeighbor, "/var/log/OLSRData/neighbors",
      sizeof(pathNeighbor));
#elif defined __ANDROID__
  //abuf_json_string(abuf, "os", "Android");
  strncpy(pathNeighbor, "/sdcard/OLSRData/neighbors",
      sizeof(pathNeighbor));
#elif defined __APPLE__
  //abuf_json_string(abuf, "os", "Mac OS X");
#elif defined __NetBSD__
  //abuf_json_string(abuf, "os", "NetBSD");
#elif defined __OpenBSD__
  //abuf_json_string(abuf, "os", "OpenBSD");
#elif defined __FreeBSD__ || defined __FreeBSD_kernel__
  //abuf_json_string(abuf, "os", "FreeBSD");
#else /* OS detection */
  //abuf_json_string(abuf, "os", "Undefined");
#endif /* OS detection */
  fp = fopen(pathNeighbor,"a+b"); /* open/create file for appending */
```

We will now describe the fields that we extract from each table. Each row is completed with an extra timestamp indicatind the absolute time and another one indicating the elapsed time since the beginning of the exepriment.

Neighbours

this table is one of the most important source of data. It provides data regarding the immediate neighbours of a certain node in each moment. The data we store is saved among a timestamp, in order to align all the nodes and try to understand what happened in the post-processing phase. For each node we store:

- IP address of the neighbour we are considering
- LQ: linq quality between our machine and the specified neighbour (useful in multi-interface configurations, where we can have many links with a neighbour)
- NLQ: the NLQ parameter shows how the neighbour sees us from 0 (very bad) to 1 (very good).
- SYM: this states whether the link to this neighbor is considered symmetric by OLSRd's link detection mechanism.
- MPR: indicates if this node acting like a MultiPoint Relays for my machine;
- MPRS: indicates if the node has selected us to act like an MPR for it
- will: the neighbour willingness. Nodes can advertise their willingness to be selected as MPR.

The file we modified is neighbor_table.c and the modification we put inside is in the point where the table is updated. The modifications we added are the following (plus the header specified before):

```
   fprintf(fp,"\n--- %02d:%02d:%02d.%02d -------------NEIGHBORS\n\n"
"%*s  LQ    NLQ     SYM   MPR   MPRS  will\n",
nowtm->tm_hour,
nowtm->tm_min,
nowtm->tm_sec,
(int)now.tv_usec / 10000,
iplen, "IP address");
   fprintf(fp,"%-*s %5.3f %s %s  %s  %d\n",
```

```
iplen,
OLSR_ip_to_string(&buf,
&neigh->neighbor_main_addr),
(double)lnk->L_link_quality,
neigh->status == SYM ? "YES " : "NO ",
neigh->is_mpr ? "YES " : "NO ",
OLSR_lookup_mprs_set(&neigh->neighbor_main_addr) == NULL ?
"NO  " : "YES ",
                                    neigh->willingness);
```

Link Set

The local link information base stores information about links to neighbors. A node records a set of "Link Tuples" (L_local_iface_addr, L_neighbor_iface_addr, L_SYM_time, L_ASYM_time, L_time).

- L_local_iface_addr is the interface address of the local node (i.e., one endpoint of the link),
- L_neighbor_iface_addr is the interface address of the neighbor node (i.e., the other endpoint of the link),
- L_SYM_time is the time until which the link is considered symmetric,
- L_ASYM_time is the time until which the neighbor interface is considered heard,
- L_time specifies the time at which this record expires and *MUST* be removed

When L_SYM_time and L_ASYM_time are expired, the link is considered lost. This information is used when declaring the neighbor interfaces in the HELLO messages.
The file where all this process ran is link_set.c and we modified it adding the following lines of codes:

```
    fprintf(fp,"\n--- %s ------------------------------ LINKS\n\n",
OLSR_wallclock_string());
```

```
   fprintf(fp, "%-*s %-6s %-14s %s\n",
addrsize,
"IP address",
"hyst",
"   LQ     ",
"ETX");
   fprintf(fp,"%-*s %5.3f %-14s %s\n",
addrsize,
OLSR_ip_to_string(&buf,
                                  &walker->neighbor_iface_addr),
                                  (double)walker->L_link_quality,
get_link_entry_text(walker,
           '/',
&lqbuffer1),
get_linkcost_text(walker->linkcost,
false,
&lqbuffer2));
```

Here is possible to access an interesting parameter: ETX, used to evaluate the quality of a link between two nodes.
hyst or Hysteresis: Link-hysteresis means that one, based on some function, "slow down" link-sensing. This is to make links robust against bursty loss or transient connectivity between nodes. This means that we are interested in making sure a newly registered link is not just a node passing by at high speed or a node that alternates between residing just outside and just inside radio range. In other words, hysteresis provides a more robust link-sensing at the cost of more delay before establishing links.
Unfortunately OLSRv1 is laking on the other very useful unit of measure of the quality of the signal: ETT, a metric available only in OLSRv2.

2-Hop Neighbours

The last table that has been decided to store, is the two hop neighbours, this table have all and only the 2-step neighbours for my node. Is possible to specify them as all the neighbours of my neighbours except my node and my neighbours. This table is very useful in particular for mobile networks (MANET

for our purpose). Only a minimal portion of information is needed from this table, in fact is possible to infere everything is needed from the other 2 tables already logged, and use this only as prediction support for nodes switching capabilities in a Cognitive Network.
The file to modify in this case is two_hop_neighbor_table.c and the logging tool do as follows:

```
  fprintf(fp,"%s two hop stuff here \n", OLSR_wallclock_string());
  fprintf(fp,"%-*s ",
ipwidth,
OLSR_ip_to_string(&buf,
          &neigh2->neighbor_2_addr));
  fprintf(fp,"            ");
  fprintf(fp,"%-*s %s\n", ipwidth,
OLSR_ip_to_string(&buf,
&entry->neighbor->neighbor_main_addr),
get_linkcost_text(entry->path_linkcost,
false,
&lqbuffer));
```

As is easy to see the procedure store only the next step and the overall link cost.

5.3.5 External commands

The goal here is to test the reaction of all the network parameters after the modification of some values in OLSRd configuration. In order to keep the node alive during the alteration of those values, a system that provide connection to OLSRd process and modification of some parameters was needed. We needed to create some custom commands with the purpose of modifying OLSR configuration during its execution. The best solution was to modify the parameters runtime without the need of restarting the daemon everytime, but also an hybrid solution was acceptable. A first implementation was a proxy that had the ability to receive external commands and execute some functions previously implemented. This wasn't a clear solution, but good enough for the first phases of the Testbed evolution. During the re-engineering of the whole testbed we took advantage of the telnet plugin in order to launch commands to

our OLSR daemon without interrupting it. This was possible thanks to some functions added to the original plugin and some fixing due to the fact that this plugin was not build for Android and our target is to set on the fly all the parameters by using our COGNET App. The chosen solution was the Telnet plugin. A new OLSRd plugin that let a user interact with the OLSR Deamon. This plugin was made to get some parameters and to set hna entries. Once the plugin is installed is possible to connect via telnet (or netcat) using the ip address of the machine where the deamon resides and the specified port (is possible to specify a particular port in the OLSRd configuration file).

Socket commands

This version of code was basical and buggy, in fact the passage of between Java and C, using JNI, is not completely trustworthy. This draft was made only for testing purpose and is reported here mainly to track the evolution of the whole project. A Socket was running inside OLSRd, waiting for commands coming to a specific port, all the commands were hardcoded and not easily configurable nor customizable, in fact each modification was quite hard to be done. Another socket were running inside our Testbed, with the purpose of sending commands to our OLSRd instance. The communication between the two sockets was not properly managed, and this caused some random crash hard to debug. Those commands, if not properly formatted, were causing the crash of OLSRd and Testbed too. We decided to search for another solution more flexible and easier to expand for future work. A robust solution were mandatory for the next stages of the project.

Telnet commands

As mentioned, OLSRdv2 was not exploitable for this project, this version had all needed feature directly implemented in the original version. The parameters modification was running via telnet commands, and a whole netcat/telnet interface is implemented with lot of commands already configured. After some trials to build a brand new telnet plugin. it was preferred to move to another solution, a non-approved plugin for OLSRv1. The only trace of this plugin was in some comment in OLSRd mailing list, but as in OLSRv2 the solution of the

same problem was telnet driven, the same approach was used in this project and fix this plugin for Android.
The main fixes made on the telnet plugin found was in the compiling stage. Some flags were missing or misplaced, also along the code found, some pre-compiling conditions that removed a lot of features from any non-Linux system. In particular all the Foreign commands were forbidden for all the systems except Linux, this caused the program to run everything as expected, but no custom commands was loaded in the commands structure related to this plugin.

As feature 8 commands were implemented exploiting this plugin. Other commands were already available in the Telnet plugin.

Basic commands There are some commands already coming with the plugin chosen to use. The main functions of those commands will be briefly described.

HNA This command can be used to add, remove and list HNA entries.

hna (add|del) <address>/<netmask> From here is possible to manage the HNA entries;

hna list this command list all the HNA entries.

Interface This command enables/disables, adds/removes or lists interfaces to/from OLSRd. Newly added interfaces inherit their configuration (includeing LQ multipliers) from interface defaults. If an interface gets removed and re-added all special settings (LQ multipliers, timings,...) get lost.

interface (enable|disable) <name> To preserve the configuration the disable flag may be used, if this is used, a disabled interface can be re-enbaled using enable.

interface (add|del) <name> This command add or remove a particular interface from the settings of OLSRd.

interface status <name> The status command prints current addresses and some other information for a specific interface.

interface list this cocommand prints all interfaces and their current status.

LQMULT This command alters LQ multipliers for an given neighbor on one or all interfaces.

lqmult (add|update|set) (<interface>|*) <neigbor> <value> Add inserts a new multiplier (error if exists); Update changes an existing multiplier (error if not exists); Sets adds or updates a multiplier.

lqmult del (<interface>|*) <neigbor> only deletes one multiplier whereas

lqmult get (<interface>|*) <neigbor> can be used to retrieve the current multiplier value(s) of a given neighbor.

lqmult flush <interface> removes all multipliers from a given or from all interfaces.

lqmult list <interface> prints all multipliers by interface.

Terminate This command terminates OLSRd. To prevent accidental invocation this command must be acknowledged.

Custom Commands The main need was to be able to edit the Hello messages and the Topology Control messages timing, it was added also the HNA timing for completeness. We will list here a piece of code to explain how was made the implementation of those functions, very similar one to another.

```
DEFINE_TELNET_CMD(cmd_hellointerval_struct,
                  "hellointerval", handle_hellointerval,
                  "modify interval for hello messages and reset timer",
                  " hellointerval 5 (s) -> set value\n\r"
                  " hellointerval -> get value");

static telnet_cmd_function handle_hellointerval(int c, int argc, char*
    argv[])
{
  if(argc > 2) {
    telnet_print_usage(c, cmd_hellointerval_struct);
    return NULL;
```

```
  }
  struct interface *ifs, *oldifs;
  struct OLSRd_config *cnnf = OLSR_cnf;
  struct OLSR_if *in = cnnf->interfaces;
  /* do something useful or cool or at least funny ;) */
  if(argc==2) {

    for (ifs = ifnet; ifs != NULL; ifs = ifs->int_next){
      ifs->hello_etime = me_to_reltime(atoi(argv[1]));
      signal(SIGHUP, OLSR_reconfigure);

    }
    struct if_config_options *cnf;
    for (in = cnnf->interfaces; in != NULL; in = in->next){
      cnf = in->cnf;
      cnf->hello_params.emission_interval=(double)atoi(argv[1]);
    }
    signal(SIGHUP, OLSR_reconfigure);
    return NULL;
  }else{
    struct if_config_options *cnf;
    for (in = cnnf->interfaces; in != NULL; in = in->next){
      cnf = in->cnf;
      telnet_client_printf(c,
          "%0.2f\n\r",(double)cnf->hello_params.emission_interval);
    }
    return NULL;
  }

  telnet_print_usage(c, cmd_hellointerval_struct);
  return NULL;
}
```

This function, and similarly all the others, will act as follow:

- If called with no parameters it will return the value of the function called at the current moment;

- If called with one parameter, integer, it will set the corresponding value to in the OLSRd settings, in seconds;
- If the number of paraemters exceed one, it will print an explanation of how this function works.

After any parameter setup, a signal is sent to the daemon in order to let it restart his timers with the new settings, but this action will not interfere with the current evolution of the algorithm, so the goal of not turning off OLSRd is achieved.
The signal command call an OLSR functionality that reconfigure all the interfaces. This functionality is usually called only for a configuration file modification. But also calling it in this way its doing his job.
All the functions are declared after the telnet plugin is loaded in main.c:

```
//load all the needed plugins
OLSR\_load\_plugins();

//add our functions to the telnet plugin

//hello functions
telnet_cmd_add(&cmd_hellointerval_struct);
telnet_cmd_add(&cmd_hellovalidity_struct);

//TC functions
telnet_cmd_add(&cmd_tcinterval_struct);
telnet_cmd_add(&cmd_tcvalidity_struct);

//hna functions
telnet_cmd_add(&cmd_hnainterval_struct);
telnet_cmd_add(&cmd_hnavalidity_struct);
```

Hello Interval Hello messages one of the basis of this protocol, they check who is alive at the frequency set by this parameter. This function have the objective to read or set a new value for the timer that decide when to send the next Hello message. If this value is properly set the behaviour of the network may change. If is set very high the result will be a smaller amount of packet

sent to flood the network for pure routing purposes, and is possible to deduce that, by sending a smaller number of messages through the network, the risk is to loose some links without noticing it in time, the overall effect will be a network less reactive in the update of the topology and link failures. If this value is too low the resulting network will be a congestioned network extremely responsive, but with a high overhead.

Hello Validity The validity of a message means the amount of time that can pass between the creation and the effective arrival of this message to his destination. A high value risk to be non effective, considering valid message that are starving in the network, so the routing performace will be degradated. A low value on the other hand can be too severe and set as unvalid messages that have only a little delay, this can cause a wrong routing tables and uneffective routing.

Topology Control Interval Topology control messages are extremely useful in the calculation of the routing tables, they differ from the Hello messages for the content. This message carry all the routing information for each node and is flooded through all the network by the MPRs. Is clear that they are heavier than the Hello messages, and they carry an amount of information more complex to be interpreted. Then the average Interval time is higher than the Hello messages Interval time, but still not too much in order to avoid non updated tables

Topology Control Validity The topology information change less often than other information usually, this value is usually set as 3 times the TC-Interval value. Flooding can degradate the network performance if this value is too high.

HNA Interval Host and Network Association messages are useful only for nodes with more than one interface, they may act like gateways to associated hosts and networks by reordering those Association Tuples. The principle and the values are often similar to TC-Messages. Interval can be set also smaller than TC-messages, in fact here only nodes that have this capabilities will send this kind of messages.

HNA Interval The interval is managed in the same way as TC-messages.

Practical usage The interface between those commands are made via telnet calls. During the loading of an external interface all the parameter values are loaded by using un-parameterized commands run directly via shell commands as follows:

```
echo \"hellointerval 00\" | telnet localhost 50023
```

This command will return the hello interval value in seconds. If the command is lauched with a parameter it will set the value of Hello Interval as specified:

```
echo \"hellointerval 50.0 00\" | telnet localhost 50023
```

This interface system made the development very modular and pretty easy and fast to expand to new features and parameters. Due to the ease of develop in this way, it was decided to add even more functions to read the current state of any parameter and create a predisposition to update also other parameters.

Capitolo 6

Android Testbed

This chapter will describe the schema of the application, the services and the JNI libraries implemented for the COGNET Android testbed. The initial purpose of this application was to became a testbed just for project. During the development, thanks to a modular approach, it was possible to add features easily in the latest phase of the development. Now is possible to recycle this base for other routing protocols than OLSRd, and other network parameters than the ones explained in the previous paragraphs.

6.1 Schema

The application is organised as flat as possible. It was decided to evolve in this direction because too many nested activities could became less comfortable and cause loss of time for the experimenter. All the services can be started or stopped from the main activity, and all the JNI calls/methods are completely transparent to the developer and are accessible via simple calls. The schema 6.1 describe at a very high level the structure of our testbed. As is possible to see the User Interface (UI) is divided into a few number of interfaces, with lot of functionalities, this was done in order to let the user have all the status every time under control without switching from activity to activity.
The activities are the following and the next sections will describe each of them with their connection with the relative services, threads, JNI functions and shell scripts:

Main Activity : is the root of the project, it contains all the data necessary for the bootstrap of our testbed, from this section is possible to initialise the whole project and launch all the manager or monitoring activities. This activity almost fully connected with all the services and libraries written for this testbed;

Host Manager : this activity list all the nodes with IP:MAC couples and let the experimenter switch them on of off. This is done by some iptables rules;

Experiment Manager : From this activity is possible to start-up a whole experiment session from a single device to all the IPs specified. It work thanks to a JNI interface and a Socket pair running in each device;

OLSRd Settings : this activity was created only for the purpose of a dynamic reconfiguration of the OLSRd parameters. It works thanks to come shell scripts that interact with the telnet interface;

TCP graphics : This activity have the purpose of showing the behaviour of the network at the TCP layer, the graphics are divided in groups of 3 or 4 data source per tab. It works thanks to a JNI call forwarded to the COGNET module;

MAC graphics : This activity, similarly to the TCP Inspector, show the behaviour at MAC layer. The data came from a Socket connected to the COGNET module, and is divided into n-second intervals, with n parametric;

Sensors : Here is possible to grab information about the movement of the device and store them into some logfiles to analyse and cross this datas with other information coming from the OSI layers.

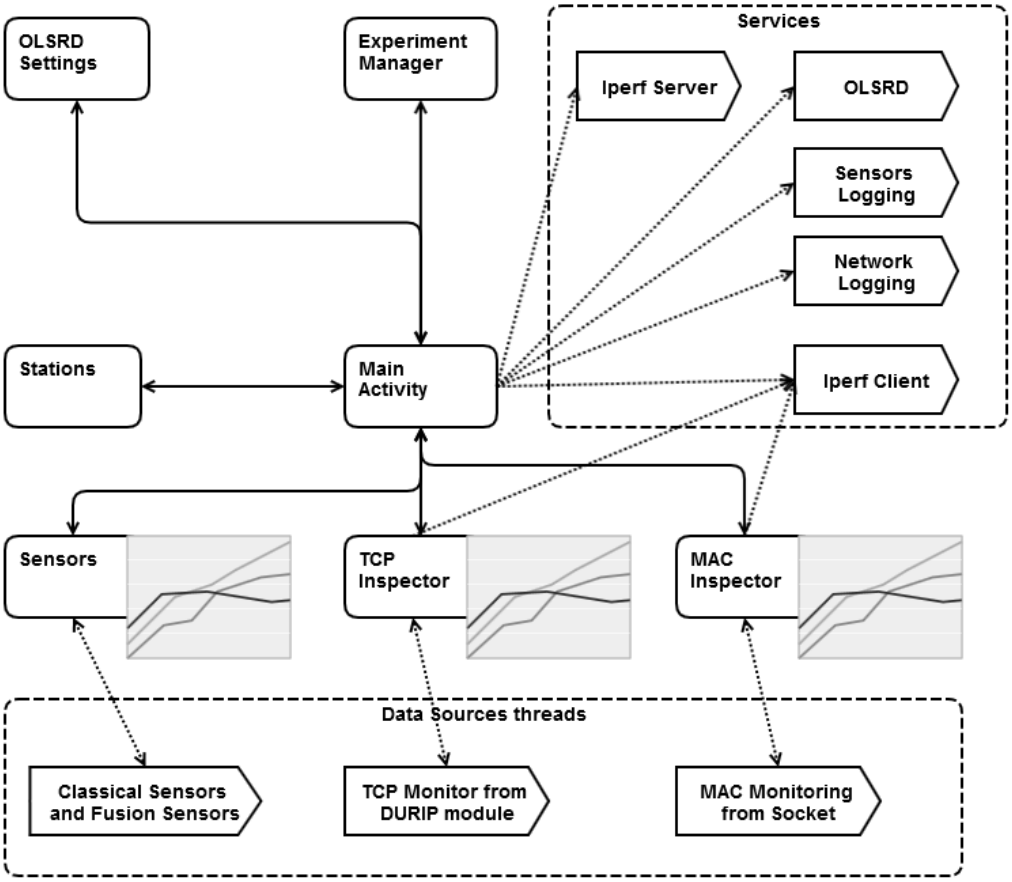

Figura 6.1: Schema of the testbed

6.2 Main Activity

A configuration portion was placed in the first activity of our testbed. Here is possible to specify a lot of parameters and launch some functionalities that will described in the next chapters. The application allows a bunch of setup before start the real execution. In the Main Activity is available the whole ifconfig and iwconfig setup. Plus here is the point where is possible to initialise the COGNET module with all the specified parameters once the network interfaces are setup. The settings available in this activity is:

- IP:to specify the IP that the device will use after the configuration;
- PORT: this port is referred to the COGNET module, it will use this port

+/- 1 to communicate with other process through sockets;

- Logging filename: This field specify the log file to be used in this experiment;
- Netmask: the netmask for the network that will be joined;
- Subnet: the subnet specification of the network for the tests
- Logging Type: Here is possible to specify with kind of logging:
  1. Files,
  2. Graphical,
  3. Both,
  4. None;
- Timeout: The amount of time to wait before launch the setup, usually 0, but for centralised experiments can be setup differently

Those commands works through a socket interface. The goal of this interface is just to forward all the commands coming to the COGNET module, and collect any answer available. It is then possible to test this configuration in 2 ways. The first is just by checking the interface state and check if the Cell is associated. A more reliable technique to check if a node is effectively in the same network is by a simply Ping command, also available through our UI. From this activity is also possible to launch many other features:

- OLSRd: is possible to control the OLSR daemon, to start and to stop it;
- IPERF client: is possible to setup and launch an IPERF client to a certain server, as will be described in the next sections;
- IPERF Server: is possible to enable an IPERF server in each device;
- Logging: it turn on or off the logging features through all the system;
- MAC graphic launcher;
- TCP graphic launcher;
- Sensors graphic launcher;

- Devices manager: To modify iptables in order to enable or disable nodes from my device;
- OLSRd Configuration: to launch the activity that manage the dynamic reconfiguration of OLSRd;
- extra features: ping, network state and other graphics too heavy to work properly;

6.3 Graphics activities

One of the main feature of this testbed is the ability to represent the behaviour of the network while the experiment is still in progress. This can be useful to infer information about the network and act consequentially without the need of study all the data in a post-processing phase.
There are mainly three graphs in this Application. All of them are dynamic and the data source is disconnected from the activity that manage the effective plotting of the graphs.
The schema 6.2 describe at a very high level the structure of our testbed. This

Figura 6.2: Screenshot of the graph for MAC

was developed by using an external library: http://androidplot.com/docs/ in order to make more understandable the interpretation of the informations, this activity needed to be able to show many graph with many curves at the same time, this need is caused by the amount of data that must be displayed, in particular for TCP are showed data regarding slow start and fast recovery functionalities that an experimenter will probably wish to understand and analyse in this Mobile Ad-Hoc Network:

- CWND: Congestion Window,
- AW: Advertised Window,
- SSHTHRESH: Slow Start Threshold,
- RTT: Round Trip Time,
- SRTT: Smoothed Round trip time,
- SRTTJTU: Smoothed Round trip time,
- RTO: Retransmission timeouts,
- FLIGHT: Flight time,
- PACKETOUT:Number of packet outgoing,
- PACKETLOST: Number of packet lost,
- PACKETRET: Number of retry,
- PRRDEV: Proportional Rate Reduction [13],
- PRROUT: Proportional Rate Reduction outgoing,
- TOTRET: Total number of retries in the experiment,
- BYTESACK: Amount of Bytes acked since the beginning off the observation,
- PKTACKED: Number of acked packets, useful to control along the amount to understand if the throughput is going fine or not,

- MSS: Maximum Segment Size, it specify the largest amount of data that this network and its components can handle.

For MAC the following measures are represented:

- MAC: MAC address of the destination node that is under analysis,
- RSSi: RSSI is a Radio-Frequency (RF) term and stands for Received Signal Strength Indicator. It is a measure of the power level that a RF device, useful to infer information about distances between nodes, the higher the RSSi the better the quality and speed of the communication through the radio segment,
- RXB: Number of received Bytes,
- RXDROP: Number of dropped received packets,
- RXDUPL: Number of duplicated received packets,
- RXFRAG: Number of received fragments,
- RXPACKS: Number of received packets,
- TXB: Amount of data sent in Bytes,
- TXFILT: Number of transmitted filtered packets,
- TXFRAG: Number of transmitted fragments,
- TXPACKS: Number of transmitted packets,
- TXRETRYCOUNT:Number of retransmitted packets,
- TXRETRYF: Number of retransmitted fails,

Finally in the Sensor graphic is represented this information:

- Accelerometer: useful to infer information about the movement currently in action, it return a 3 dimensional object with X, Y and Z all measured in m/s^2;
- Gyroscope: this information is at the lowest level, it measure the acceleration in a raw mode. in radians per seconds rad/s^2;

- Orientation: This tool uses the earth magnetic field to try to understand the orientation of the device;
- Linear Acceleration: this is a Fused sensor, it infer informations from the other three sensors and merge them together in order to obtain a much more clean source of data. This representation of the information is way smoother than the other, in fact the white noise and all the micro vibrations are removed and the data follow some manipulation;
- Gravity: Fused sensor useful to understand the real orientation of the device in the world in a much more clean way than all the other sensor. It works by mixing Orientation, Rotation Vector and Gyroscope in a differential equation;
- Rotation Vector: this vector is useful mainly to understand the current static situation of the device, it does not measure acceleration but only the current inclination on three axis: roll pitch and yaw.

For the Sensor graph was easy to develop, in fact the source is directly available from the Android Framework and to develop it it was sufficient to pick the right data and print it as graph. For the other two graphics to come up with a good solution was more complicated. The needs were many:

Readable data It is not possible to print all that lines in a single graph, it will became totally unreadable and with incoherent data. Plus the order of greatness of many of those entry are totally different one to another, this may result in a graph that seems to be flat for many entries that are just limited by a short range. No more than 4 information per graph was a good trade off between readability and feasibility.

Performances The first trial was to use more activities or one thread but it was clear that the device can stand up to this effort for a very short amount of time before crash in some points. The amount of data, if more than one data source is opened, is critically high for a mobile device, and must be reduced to the minimum.

Usability The app have the main goal to simplify the work of an experimenter. The usability is a key feature of this module. A non usable tool will be

simply ignored. In order to let this tool easy to be used, the hierarchy has to be kept as flat as possible and avoid a continuous switching between activities. It was decided to use Fragments and swipe between all the graphs instead of close and open all of them one by one.

To achieve those goals it was necessary to divide the source from the Interface, and then place an Adapter between those two objects in order to let the UI run with no limitations coming from the Source Thread. This granted a lightweight solution and a smoother interface, as well as the ability to start and stop the Data Source Thread with no problems related to the interfaces, as it is an independent object.

6.3.1 Data Source

This class have the purpose of collecting all the needed data from the TCP/MAC COGNET source. To do so it implements Runnable and it act like an independent process (thread).
It was chosen this approach after some trials without runnable interfaces, but was not possible to detach completely the waiting phase and the data gathering phase from the User Interface. This resulted in a really poor user experience and a buggy Application that crashed continuously. This thread have implemented some extra features to let it start and stop properly, without the risk of keep the memory occupied when is not needed.
After the setup of the class, it start the Socket in the MAC portions, and a shell script in the TCP graph portion.

MAC Socket The data coming from the COGNET module in this case flow through a socket, this choice was done for the nature of this data source, very constant, a high amount of data and with a high ratio. To receive the information coming from the MAC module the best solution found was to open a Socket, besides, the information will flow anyway once the experiment received the START command from the experiment manager. The macREADCOGNET is a kernel module responsible to this and many other tasks. Once the module is running is possible to open a Socket.

```
ServerCmd = "macREADCOGNET 1:"+paramSleep+" "+ paramPort +" phy1
    wlan1", ////+paramVerbose+":"+(paramPort/1000),
```

```
Runtime r = Runtime.getRuntime();
try {
            p = r.exec(ServerCmd);
    ...
    sc = new Socket("127.0.0.1", (paramPort+2));
    out = new PrintWriter(sc.getOutputStream(), true);
    mac = new BufferedReader(new InputStreamReader(
                              sc.getInputStream()));
    ...
```

In the next portion of code the Testbed keep reading any line printed by the socket, and when a line match our needs it is added in the displayed data. All the rows pass through a regular expression to extract all the needed data, those data are then fetched and pushed into our structure connected to the real draw activity.

TCP Script This data source is different from the MAC source, as the TCP module section provides information only when some traffic is effectively running through the analysed network connection, otherwise no data is sent at all.
It was decided that there is no need to let run a module for gathering data while there is no data to analyse. The module in the beginning was similar to macREADDURIIP but was changed it in favour of a classical shell script that is executed via the Runtime technique. This script is simple, and its temporization is based on the kernel clock used to write on the proc files relative to TCP connections.
When the activity is shut down there is no need for this script to keep on working, in fact he does not add any particular functionality except for reading the proc file created by the COGNET module for our purposes. The script listed here was used to extract those data:

```
#!/usr/bin/bash
A="1"

while [ $A -lt "2" ] ; do
       cat /proc/tcp_output_COGNET
       sleep $1
```

```
done
```

The rest of the activity is similar to MAC Data source, except the real source of the data, in the MAC situation it keep on working also when the Activity is not running. The COGNET module keep write on those proc files eveytime he see a packet, but if no one is watching at those files the information will be lost.

6.3.2 Graph Adapter

This class is connected to an observable object and implements the Parceable object in order to let it communicate with its utilizator, the Activity. This class is needed as bridge between The Data source thread and the User Interface (Activity) in order to don't let the activity got stucked permanently while the thread keep update the data source.
This class is also useful to simplify the development of future graphs, in order to expand this testbed for future works. Mainly is possible to state that this is an abstraction layer made ad-hoc as bridge between the source and the destination. It allows to be instanced along with an Observable object and this was our choice. This object is connected via an Observable object that have the ability to issue the Observer (that reside in the User Interface) every time the Data source have deployed a new set ready to be printed in the graphics.
This class was introduced only in a late stage of the project, when the concept of fluent User Interface became more important and also the Usability and the Stability became crucial for the project.

6.3.3 Graph Activity

This activity is divided in fragments as described before. Those fragments load the data coming from the Adapter simultaneously as long as all the data for this activities are stored in a single matrix with fixed dimensions. It was decided to act in such a manner to let the memory be free as soon as possible, but due to the massive amount of data running in this section and all the computations necessary to print it out as graph. Both approaches were tried, dynamic and fixed data structures and the performances dramatically boosted up with the static version.

Basically every time the observer receive an update, the activity read the corresponding data and push them into the our matrix. This matrix have one line per each parameter read and one column for each time interval in the graph. As there is a maximum of 300 time intervals * 13 MAC data, and as float they occupy 4 bytes each. In this scenario the app will occupy 15kB only for this purpose but the app will cut all other side activities that came from the Dynamic approach. In the Dynamic approach missing data were calculated and all the redrawing of the graphics were based on sequential requests instead of single request. This results in less memory occupied but much more interactions between the app and the underlying modules.

This data structure is shifted each time the observer send a new data set, the last portion is deleted and the new information are pushed in the head of the each row. This rotation have cost 1 thanks to the Dynamic Vector used, so the performance are not affected by that.

The last step that occur when an issue came from the Observer is the effective printing of the Graphics on the current Fragment. All the other Fragments are not updated until the focus are back over them, this is useful to maintain a low memory consumption and reduce useless operations. It is also possible to do so thanks to the static matrix of values described before. If the app have to redraw a whole graph from scratch it is possible to pick up all the information from this structure by choosing only the needed columns.

This class uses an Observer connected to the Observable in the Data source class, in order to know when to update the UI and start the process described in the previous paragraphs. Considering that a whole screen can take up to 30 seconds to fill completely with data, will be annoying to have to wait every time the user change graph. So it was decided to use a Fragment structure for this portion of the testbed.

The Fragment structure is divided into many graphs that share the same Data Source. This approach keep the memory and CPU consumption as low as possible, having more than one data source cause a sudden crash of the whole testbed. Is possible to swipe from a Fragment to another without loosing the alignment of the graph with the data collected until that moment.

6.4 Logging features

Each level of our testbed have logging capabilities. Combined together they can form a complete image of what is happened in a certain communication through all the layers and protocols involved. Logging features are implemented in those modules:

COGNET TCP TCP information comes directly from a script that interact with the COGNET module, clearly as the subject is TCP information, if no data travel in the network, the result of this logger will be empty;

COGNET MAC Those data come from a Socket placed inside the COGNET module, this module have the main rule to inspect and log data, and in this case to send those data to our testbed, contrary to COGNET TCP, this module is time driven, and it reports data also in case of network with no traffic;

OLSRd This logger is implemented among the original OLSRd source code, in order to let us process the data and the evolution of the routing tables at the end of the communication, and to try some information crossing with other layers;

Sensors Here the app will store all the data coming from the sensors of movement.

We had the needing also of some post-processing information. To achieve that a log system was set.

6.5 Commands Service

This interface to the shell scripting was made in order to let the commands be completely asynchronous from any other activity. The only way to stop them is to store the ID or to find it back by using a 'pgrep' command with the name of the created process. Any thread launched in this way will not interfere with the UI, and the app is able to stop them with a simple kill command.

6.5.1 Asynchronous calls

Perform asynchronous calls, as described in the previous sections, is definitely a need of this project. To act in a completely independent manner compared, to the rest of the application, it was necessary to use a separate class

```
    private class JniCall extends AsyncTask<String, Void, String> {
        @Override
        protected String doInBackground(String... params) {

            mainJNI(4,params);
            return "Executed";
        }
        @Override
        protected void onPostExecute(String result) {
            TextView txt = (TextView) findViewById(R.id.output);
            txt.setText(result);
        }
    }
```

6.5.2 JNI functions

Some functions must be executed at the lowest level, or interact with the kernel modules made, or the kernel native modules needed. To perform this operations some low level libraries were necessary. Many JNI functions and libraries were developed, they act like bridge between other libraries or simply perform low level operations in a separate context than the testbed itself.
To declare a function inside a class the first step is to import the library needed, both Linux libraries and our JNI libraries. Once the needed libraries are loaded the next step is to declare the functions that will be used with the parameters that they will accept as shown in the example:

```
    static {
            System.loadLibrary("nl-genl-3"); // "libmyjni.so" in
                Unixes
            System.loadLibrary("MACReadServer");
}
```

```
public native int mainJNI(int n , String []s);
```

6.6 Iperf Client Service

Iperf is a very common tool in the networking community. Is used for load testing purposes in experimental networks and in production network to understand the behaviour of the infrastructure during a massive workload. The mechanism is simple, once specified the destination, the interval between requests, the duration of the test and some other optional parameters the tool start to interact with an instance of Iperf running in Server mode at the destination node. During the test, the client, send as many data as possible to the specified destination. Those burst have a specified duration and between all of them the client wait an 'interval' amount of time, speficied as well as the other parameters. To test properly the throughput it was implemented a client-server iperf service on each device. Is then possible to test the device with workload or in standby mode.
The setup permits also to setup all the parameters to start both an Iperf Client and Server. the following parameters are used to setup Iperf client:

-ip Destination IP: the target of all the Iperf communications,

-i Interval: the interval between a burst and the next,

-t Duration: The duration of each burst.

The Iperf Client run as an Android Service. Here it wasn't found any need or special issue or need coming from external libraries or detaching root commands. This task was managed by calling the client in a very simply way, by ignoring the outuput and by associating it to a Runtime object. The background service keep running until an explicit command kill it.

6.7 Iperf Server Service

Each device have the ability to launch an Iperf Server in order to receive data from any other client in the network. This thread can also be killed whenever the experimenter decide to do so. This permits some experiments about the

behaviour of the network and eventual Cognitive algorithm, when a heavy communication suddenly crash, also if the crash is only partial. This thread is asynchronously launched from the main activity and can be also stopped from there.
This thread is asynchronous and detached from the Interface. Is also very lightweight so is possible to wit for incoming connections without problems, on the other hand if the logging system is active, the amount of data for each connection can grow a lot in a short amount of time and this can cause problems. As per the Client, also the Server run as a simple Android service, the classical system for managing background threads in Andorid.

6.8 Experiment Manager Activity

The last part of the App development roadmap was a tool to manage remotely all the device involved into an experiment. The purpose of this section is to explain how was developed and what is the goal of this module.
From the interface is possible to specify some fields, basically:

- List of Ips to initialize to the experiment: it is also possible to enable and disable previously saved IPs, once started the experiment they are saved into a text file divided in StartIPs and StopIPs;
- Delay: time to wait before start the experiment, useful to sync all the devices;
- Port Number: the port where all the communications will be done through the devices, this convention was created to work with one port on many commands:
  - #Port to start;
  - #Port to stop;
  - #Port to communicate.

The liraries imported for this activity

```
static {
        System.loadLibrary("nl-3"); // "libmyjni.so" in Unixes
```

```
            System.loadLibrary("nl-genl-3"); // "libmyjni.so" in
               Unixes
            System.loadLibrary("ExperimentManager"); //
}
    public native int startExperiment(int n, String []s);
    public native int stopExperiment(int n, String []s);
```

The only commands available in this section are the same coming from the JNI functions declared:

- Start Experiment: Here all the specified IPs will be stored inside a text file and a call to the JNI function made for this purpose will be done;
- Stop Experiment: In this function all the IPs loaded in the list are currently running. Is possible to de-select some of them, that will stay active, all the selected IPs in the list will shut down once this button is pressed. A separate file will be created: StoppedIPs, in order to avoid the loss of inserted data (as insert data through touch screen is a bit annoying).

It was used the technique of storing information into files in order to preserve them for next experiments. This is also easier to manage compared to volatile data structure that need ports, jni bridging, socket or other techniques in order to work properly. Once the data is stored into the file, the JNI function plus the C library will load the values without any supplementary effort.
Currently this tool is in development to add features, but the structure is highly modular so is easy to expand it with new features as IPERF or OLSRd settings for each node of the network. As the telnet commands can take an IP as parameter, that usually is 'localhost', is possible to remotely control all the parameters on each device in the experiment.

6.9 Olrsd Reconfiguration Activity

The present testbed is capable of calling the some commands added on OLSRd and accessible via Netcat or Telnet [24]. This tool was added in the very final phases of the project. The ability of customize and tune OLSRd is a key feature

for trying to improve its routing capabilities and, more than else, the network discover and the topology management.
If the message rating and validity is modified by prediting a good or a bad situation, is possible to get new connection before the old ones disappear, or to maintain the old ones if the situation seems to be stable. In the first case is possible to increase the message generation rating, and obtain a more responsive node, that will find other routes in a faster way compared to a node with less HELLO messages. On the other hand a stable node, with stable neighbours, need only the minimum amount of HELLO messages, since he does not need to find urgently a new route as the communication with the current neighbour set will be constant in the next future, for this scenario a reduced messages generation rating is sufficient to provide a good connectivity with an optimized throughput. In fact the more messages are generated to flood the network, the less information can effectively be transported. To achieve this capability it was used the interfaces created in OLSRd, described in the previous chapter. The User interface is as easy as possible, with 6 bars indicating interval and validity for HELLO, Topology Control and HNA messages.
Those bars are loaded by using the commands implemented without parameters. When the experimenter modify that value, another call is made through the same functions with the speficied value. A command can be thrown by using a Runtime and a shell command. In the studied case a command like this shall be used.

```
Runtime.getRuntime().exec("echo \"hellointerval 35.0 00\" | telnet
    localhost 50023");
```

In the case of parameter reading it is sufficient to attach a BufferedReader to our Runtime instance in order to read the results.

6.10 Extra Features

During the development it was created some useful tools to help the fast testing and management of the experiments and the network settings. Mainly those tools were created for avoid wasting of time and reduce the console interaction as much as possible. The most useful tools were:

- Ping to a certain client: to check if the connection is active or not;
- Host managment: to switch on or off some hosts via Iptables rules;
- Network status: it check the current status of the connection and the cell associated to the device;
- Olsrd switch: it activate or deactivate the OLSR daemon, useful to check what happen when a node crash partially but is still connected to the network;

6.10.1 Ping Command

This was one of the first implemented commands, it was made to test the connectivity of the devices. This feature is really basic and useful mostly when something is going wrong with the experiment, if some node is not working as expected or my node is not entering int the routing tables of other nodes.
It works thanks to a Runtime mechanism where is possible to interface Java to the device shell, and execute real shell scripts with limited scope or permissions. Is possible to run also root scripts but is not always working as expected.

6.10.2 Hosts management Activity

This activity was created to manage the connectivity between nodes. The goal is to achieve some routing modifications via Iptables rules.
Here is possible to see all the devices connected to the network and see if they are active for the node currently analysed. In the list of nodes is possible to see both the IP and the MAC addresses and decide if switch that node ON or OFF simply by clicking on it. A rule is added or deleted depending on the previous situation of that node. A node active will have no rules, and an un-active node will have a rule, in the Iptables table of the analised node.
This rule will state to REJECT all the nodes coming from the selected node. A DROP will have similar effects, but slower, probably more realistic than the REJECT rule that will say an answer, also if negative, regarding the incoming packets. This schema 6.3 represent the functionality of this technique. As is possible to see:

1. in the first portion all the nodes are connected and no rules is added in any node;
2. if node B is selected, the effect will be to drop all the packets coming from node B, he disappear from this neighbours;
3. the network adjust itself by rewriting the routing tables in order to create a new connection to node C through node D and F, that previously was avoided because of the cost;
4. by removing the rule the network will slowly restore its previous status.

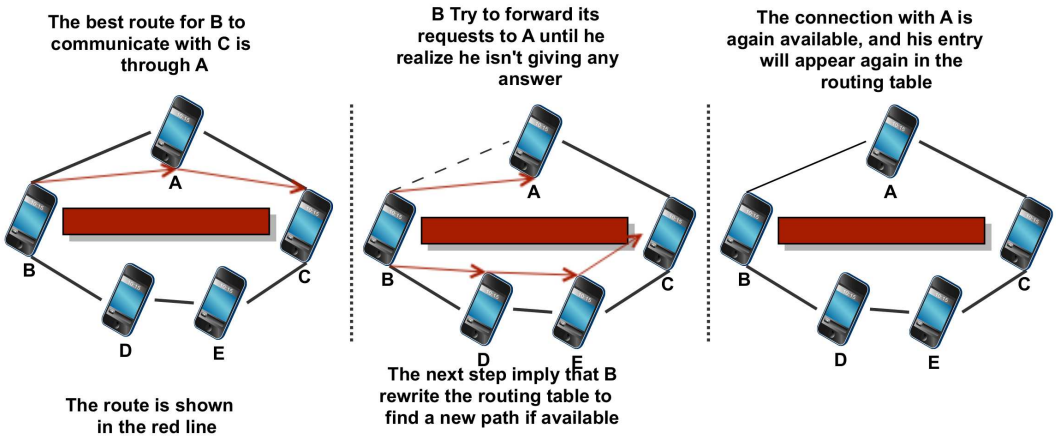

Figura 6.3: Schema of the Iptables mechanism

To do such operations some JNI libraries were used also in this portion of the testbed. The following were the most used functions:

```
    static {
            System.loadLibrary("hostsLib");
}
    public native String[] activeHosts(int n);
    public native String[] unactiveHosts(int n);
    public native int hostOn(String s);
    public native int hostOff(String s);
    public native int flush();
    public native int fill();
```

The functionality coming from each of those functions is:

- activeHosts: list all the active hosts
- unactiveHosts: list all the host that the experimenter decided to exclude from the network
- hostOn: activate a host if was deactivated before
- hostOff: deactivate a host if was not previously deactivated
- flush: flush all the iptables entries, the result will be Activate all hosts
- fill: fill iptables with all the entry, the result will be this node excluded from the network

6.10.3 Network status Command

This command is useful only to test if the setup was successful or not. It check the network configurations over the specified interface. It also return the CELL to whom my device is connected. Other information coming from this tool can be useful to get quality of the signal clues. But the main information is the CELL, that let us know if the node will be able to communicate with all the other nodes in the Ad-Hoc nodes.

6.11 JNI Libraries

In order to perform some special activities some ad-hoc libraries were implemented. Those libraries had the goal to communicate with lower level commands and some functions from our COGNET module and other libraries made for this project. The methods in jni have a strict to be declared and used. The same library cannot be shared through multiple classes, in fact, part of the name of any method in the library, must be the name of the class itself.
The structure of the header must follow this form:

```
JNIEXPORT jint JNICALL
   Java_it_COGNET_1app_ExperimentMasterActivity_mainJNI(JNIEnv *env,
   jobject thisObj ,
              jint n, jobjectArray stringArray){
```

By specifying the package, the activity in which it will work and the name of the function that are going to be created.
If some parameters need to be passed back and fourth from JAVA to C and vice-versa, the parameters have to be specified into a JNI structure in order to let JAVA read/write them. For the C side, the difference is not so big compared to the rest of the code. Also for the parameters the syntax is quite strict. The following example will describe how was possible to to extract data from the calls coming from the COGNET Testbed:

```
    int stringCount = (*env)->GetArrayLength(env, stringArray);
argv = (char **) malloc(sizeof( char *) * stringCount);
    for(ii=0; ii < stringCount; ii++) {
            jstring element = (*env)->GetObjectArrayElement(env,
                stringArray, ii);
            const char * rawstring = (*env)->GetStringUTFChars(env,
                element, NULL);
            ...
```

The main JNI libraries created are the following:

- Device Switch: to manage iptables in order to enable or disable hosts;
- OLSR: interface for switching OLSRd ON or OFF;
- Experiment Manager: tool for starting and stopping an experiment over set of IPs;
- COGNET Socket interface: this module contain the interface to startup the device modules and its interface related to tour tests.

Device Switch The need of this library came from the unavailability to edit the iptables entries when you are in user mode. Android have different policies than other Linux systems, than is not possible to force this rule and act like root whenever you want. Busybox [REFERENCE] may help for issues like this, but is not a clean solution and the results rely on an external tool.
The modifications made by our commands are minimal. Flushing IP tables is a basic command, add or delete one entry is the same command except for a flag

-A / -D and add all the entries is just the same command replied many time. The command used for this purpose is:

```
char sx[80];
sprintf(sx, "su -c \"iptables -A INPUT -m mac --mac-source %s
   -j REJECT\"", ss);
system(sx);
```

The only other command that need some description is the listing of the un-active nodes. There is a file where all the entries are saved and returned as array to the Java APP. This mechanism is a bit twisted, in fact it will be reported here, as is the only example of returning a complex object from JNI to JAVA. The String structure must be built following the JAVA syntax, so the C library have to create a variable base don the parameter 'env', representing the current environment, and create a memory location that follow the rules of a JAVA array of strings.

Once this step is made is possible to extract each IP with a Regular Expression from each line returned from the Iptables command and analyse them one by one and, eventually, return them to the calling function.

```
sprintf(ss, "su -c \"iptables -L INPUT\"");
//check if modules are up or not
jclass sclass = (*env)->FindClass(env, "java/lang/String");
jobjectArray ret = (*env)->NewObjectArray(env, (int)n, sclass,
   NULL);
regex_t compiled;
regcomp(&compiled, "REJECT.*MAC\\s(([[:xdigit:]]{2}[-:
   ]){5}[[:xdigit:]]{2}).*", REG_EXTENDED);
...
while (fgets(line, sizeof(line)-1, fp) != NULL) {
       regexec(&compiled, line, ngroups, groups, 0);
       tmp = (*env)->NewStringUTF(env,line);
       (*env)->SetObjectArrayElement(env,ret,i++,tmp);
       (*env)->DeleteLocalRef(env,tmp);
}
```

OLSR Switch The other module that is not acting only as a command forwarder is the OLSR manager. It allow the app to act like root and turn olsrd ON or switch it off. The switch off came only from a pgrep function that return the correct pid to kill and the kill function. A more interesting command is the starting one. It was needed a detached process to run completely independent.To achieve this target it was found out that a system call will wait for the input until it ends. So it was decided to forward all the output coming from olsr to /dev/null.
We also had the need of let this process run also if our activity will shut down, or the entire COGNET testbed will stop, OLSRd must keep running without event noticing this, as is a routing protocol and if stopped without our command it may ruin the experiment also on other devices. This issue was solved thanks to the nohup command, to prevent any SIGNHUP or other signal that may shut down the process. Note that nohupping backgrounded jobs is typically used to avoid terminating them when logging off from a remote SSH session, in our case the command is launched in a virtual session that immediately shut down.

```
sprintf(ss, "su -c \"nohup /system/bin/olsrd -f
    /sdcard/COGNET_TESTBED/CONFIG/olsrdTABLET.conf > /dev/null
    &\"");
system(ss);
```

Capitolo 7

Results

This chapter will show the analysis the result of the real exploitation of our testbed. First of all will be evaluated the effectiveness of the analysis in real time, with some example and some screenshot of the graphics representing the behaviour of the network. Secondly will be presented a report of an experiment made with three mobile nodes with the related data and the possible cross-layer information we could infer. In the last portion of this chapter will be explained a portion of the future works planned for this testbed to improve its features or add new ones.

7.1 Testbed experiments

During the experiments is possible to setup the device in four manner in order to obtain different kind of results. The three options are:

- log files
- Graphics
- log files and Graphics
- none

To obtain the graphics that are going to be explained now, is necessary to set the graphics source on, otherwise nothing will be showed or, sometimes, the graphics will stuck while waiting for data. The behaviour of some network parameters and the variations that some movement caused to them will be now

analysed. The Sensors will be skipped as they are not representative, the movement were made in different area of the same floor of the Department of Information Engineering, University of Padua. The following set of experiments are useful only to prove the effectiveness of the testbed.

7.1.1 MAC Layer

At this layer different tests were made, to check the effectiveness of the testbed during some live analysis. The experiments was divided into many phases, where some variables were changed constantly as: the topology of the network, the load on the network and the Iptables of each node.

RSSi

This metric indicates the power of the signal to another physical radio antenna. It may be extremely useful to infer the current position and to foresee the movement of the devices, in order to take action before a loss of packet occur. The live experiments are useful to notice variations on this parameter, and most of all to determine when the signal is going to disappear or to overcome a relay's RSSi.
In the figure 7.1 is clearly possible to see that the devices are moving away from each other. The experiments started with the two device placed in the same room. Immediately the analysed devices was moved into another room as the RSSi value state, in the last phase of the experiment the devices was carried in another floor and the signal power dramatically decreased. This graphics was useful in the next phase of the experiments, to understand when two devices ran out of sight and coordinate movement circumscribe exactly the area within which the experiment can be conduced or, alternatively, to go out of range voluntarily but in a controlled manner.

Trasmission in movement

In this experiment the tables were initially far, in order to force a 2 hop communication between the source and the destination. Once the IPERF started the distance between the two devices increased as the transmission retry count indicate, the amount of packet not acked by the destination grow as the

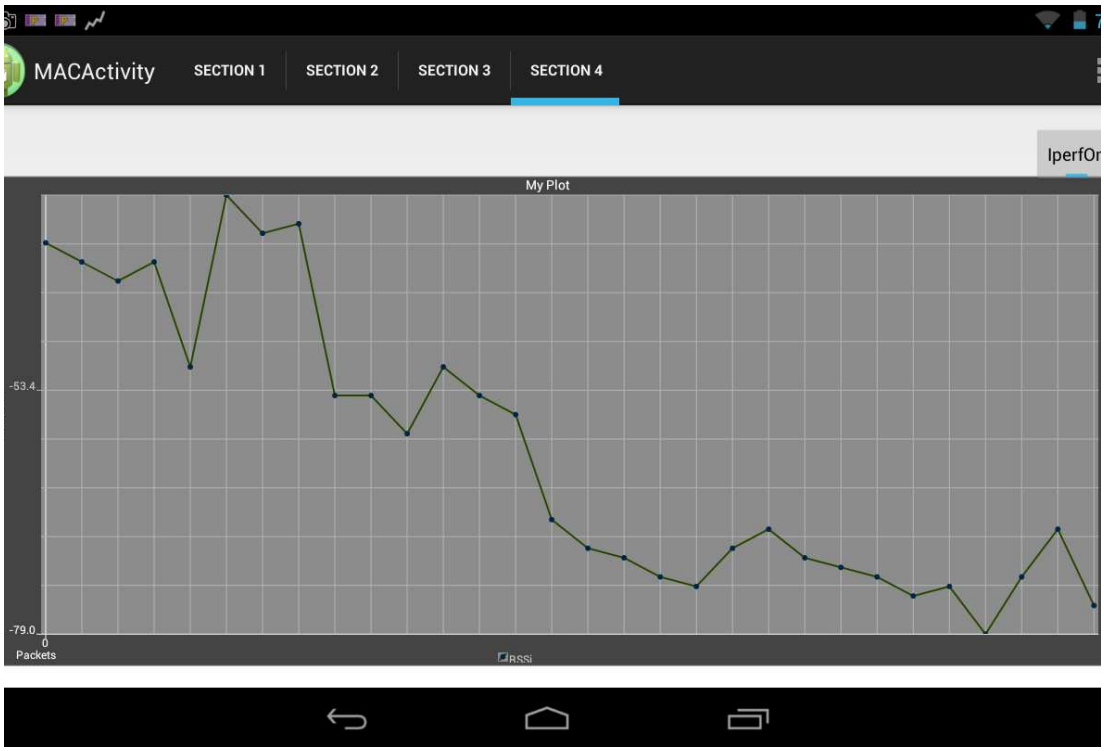

Figura 7.1: RSSi values of two nodes in dBm, each tick is approximately one second

network becomes less stable. As is possible to see in figure 7.2 the transmission is initially limited by the topology of the network and a certain amount of packet has to be retransmitted. Once the distance overcome the coverage of the radio antenna, this value double.

Transmission flow

This tool test the amount of data currently flowing through the network. Here is possible to detect immediately if the load tests are working properly and how this flow is affected by the changes induced in the network. As shown in figure 7.3 the IPERF command launched from another device to the analysed device is clearly visible as long as the ack flow related to the this request burst. The command was shut down after few seconds to test the reactivity of the tool.

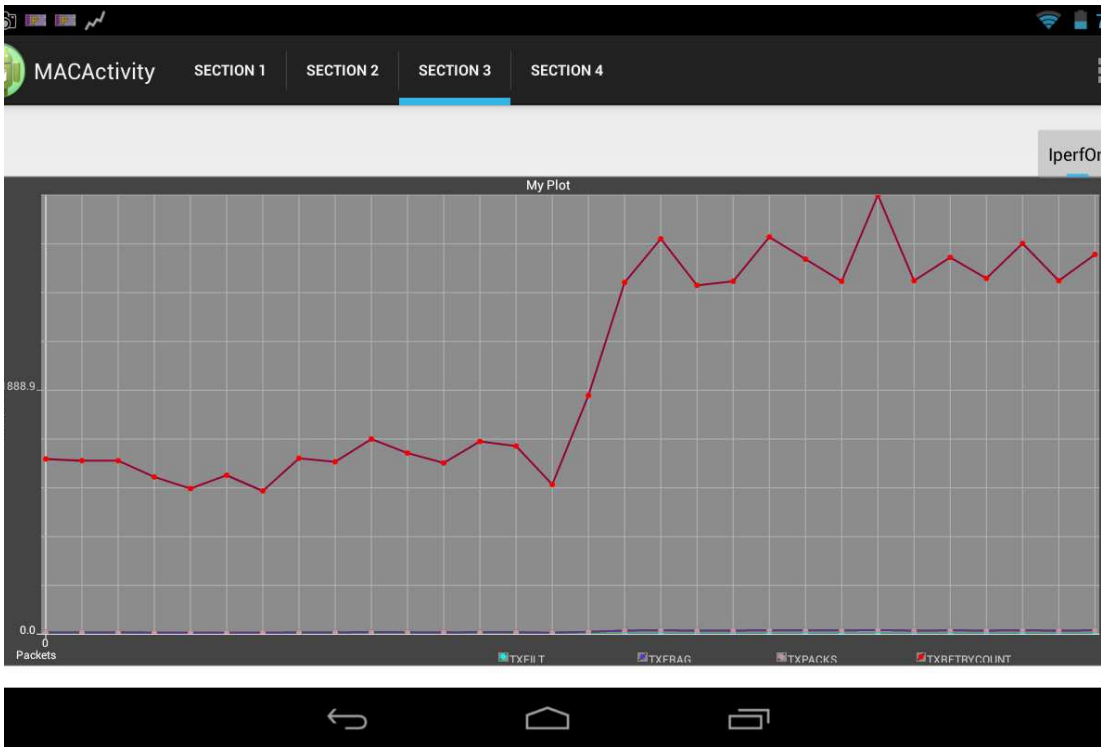

Figura 7.2: Packet count, Transmitted retry count

7.1.2 TCP Layer

At this layer the observator is the only entity able to see some data. The relay and the destination does not observe the traffic generate in this mode.

CWND

This value indicate how the communication is going at TCP layer. As this value can be forced from our COGNET testbed is useful to analyse its behaviour both in live mode and in post-processing. The Slow Start Threshold value is strictly related to the congestion window as if is bigger than CWND than the slow start mechanism is triggered. Otherwise the Congestion avoidance mode in the Fast Recovery mechanism. The following experiment show classical transmission with the behaviour of the two values. As the testbed induce a data loss with an Iptables modification, the algorithm react by dramatically reduce the Congestion Window. Picture 7.4 show how the CWND grow during the first phase, followed by the Slow Start threshold. In the end of the experiment a device was disconnected and the value of the current CWND dramatically

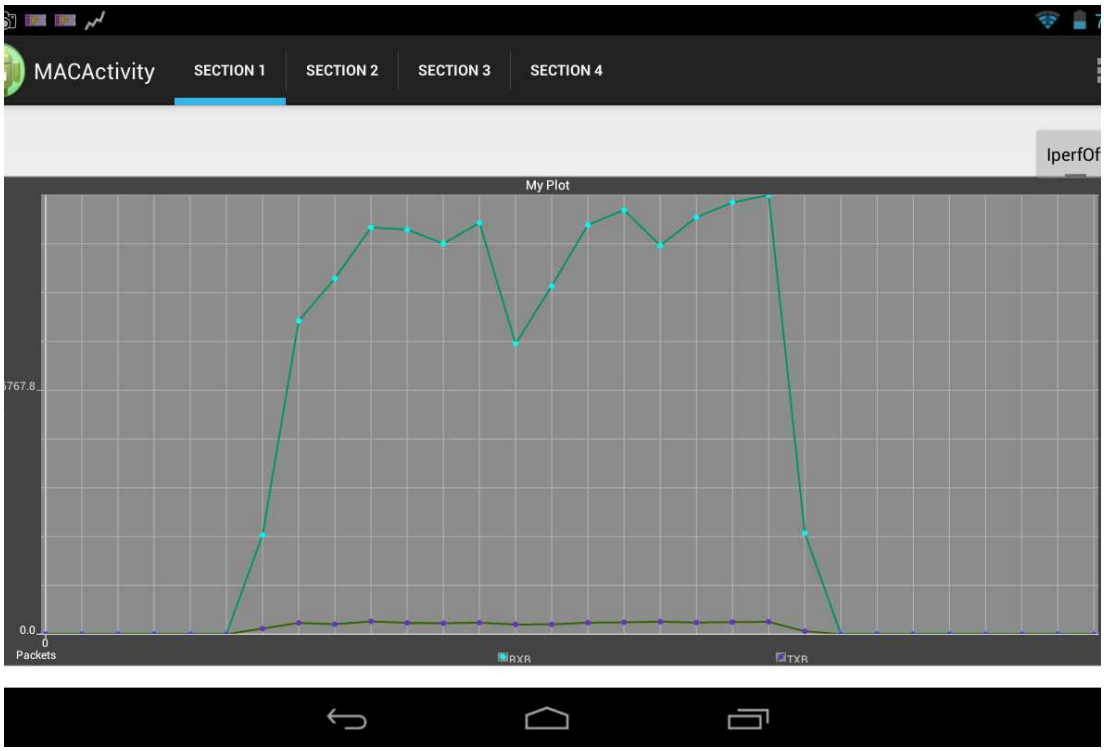

Figura 7.3: Bytes amount during an IPERF short command

decrease to 0. This is caused by a data loss, thanks to the fast recovery the value grow up immediately also if the data loss is still going on, as shown in figure 6.2.

7.2 Data analysis

In the very ending phases of the project this testbed was used to extract some data from some real experiments. Mainly mobility experiment were done, with an approximate duration of 10 minutes each. After the experiments all the log files were gathered in order to analyse them and show the correlation between all the layers, and how can be possible to predict some network behaviour with all those data available in a virtual layer able to see all the TCP/IP stack parameters and the routing protocol values too. Before start the analysis must be known that the OLSR timers started earlier than the experiment itself, roughly 80 seconds before. All the values regarding OLSR have to be shifted right by 80 seconds. This experiment consist in three replication with the same path: 10 minutes of experiments with two movements in an area where the source and the destination cannot see each other. The experiment were made

Figura 7.4: Congestion window and Slow-Start Threshold during communication with forced packet loss

outdoor this time, in order to cover real distances and have realistic data. A graph of how the experiment was conduced is shown in figure 7.5. the wall block the signal between Source and Destination nodes so, when Destination is hidden by the wall, the routing table change as the link quality will result degraded. RSSi value is also useful to predict this behaviour before the reaction of OLSRd.
By analysing the data the most interesting results extracted by the testbed comes from the information regarding CWND and SSthr 7.6, OLSRd parameters regarding the cost and the quality metrics 7.8, the MAC layer value representing the power of the signal between the source and the other nodes involved in the communication 7.9 and finally the packet in flight 7.10.
As we can see in the detailed images 7.7 the behaviour of the nodes are standard during the experiment, as the values grow until a congestion signal arrives CWND is divided in two and the threshold take the value of CWND, thanks to the TCP Fast Recovery the congestion window grow faster. The Fast Recovery algorithm is active, which uses fast retransmit followed by Congestion Avoidance. In the Fast Recovery algorithm, during Congestion Avoidance

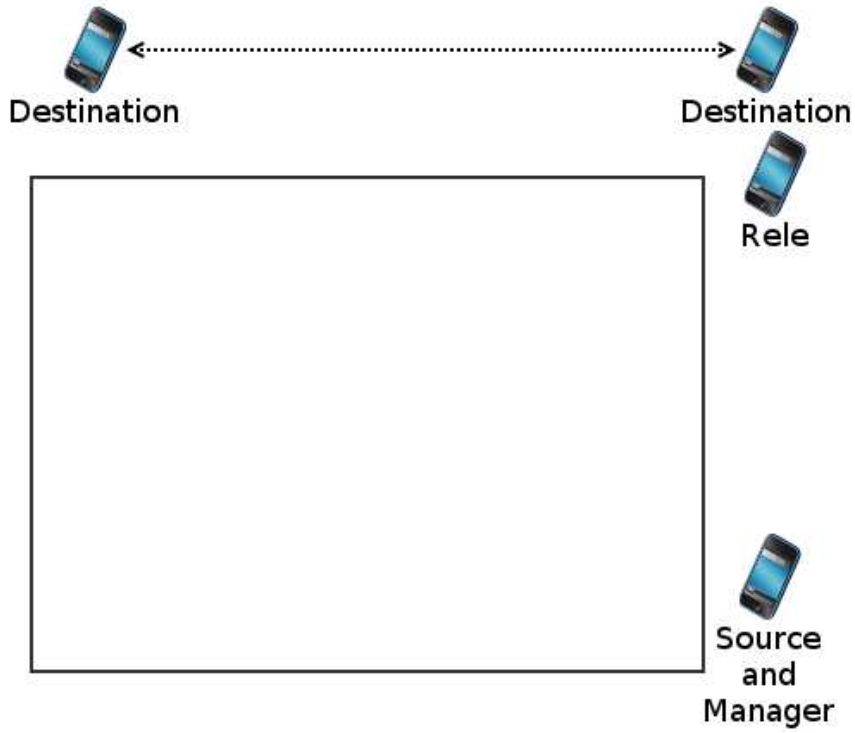

Figura 7.5: The three nodes involved in the experiment, Destination node keep moving for the whole experiment with 1 minute of stop each time it reaches his place

mode, when packets (detected through 3 duplicate ACKs) are not received, the congestion window size is reduced to the slow-start threshold, rather than the smaller initial value.

On each experiment it was launched an IPERF Client on the source and an IPERF Server on the destination nodes. Os possible to see how the Throughput

change when the network have to switch from one hop to two hop, if the routing table will be able to change before the data loss, the overall performances will increase every time the topology change in not convenient manner. On the other hand, a proactive routing protocol, as is OLSR, may have the capabilities to be forced to new routes by using network parameters coming from the whole TCP/IP stack.

From the above data is possible to see how fast is the reaction at the lowest level 7.9 at time 120 when the RSSi value get stucked as the destination disappear from the line of sight of the source. After some time, roughly after 180 since the beginning of the data gathering at MAC layer, the RSSi value goes to infinite, to indicate that this device is no more reachable. The graph at TCP layer started the data gathering earlier, the 340 value is roughly placed at the same time as the 120 at MAC layer. Is possible to see how both the CWN and the amount of packets in flight dramatically drop down then some data loss occur. To optimize the throughput, a Cognitive Network algorithm should be able to avoid those situations and reduce the CWND before the data loss occur. On the OLSR informations at time 200 is possible to see a peack that indicates how the ETX of the destination grow when the connection is lost. At this time the device disappeared from the line of sight of the source and, for a certain amount of time, at time 130 for the RSSi, it disappeared completely from the network. The cost increased as well, but in a more controlled way. The extension of this peak is 180 seconds for OLSR, so it took a while to restore the initial routing table. On the other hand RSSi had only 100 seconds of downtime regarding the communication between source and destination.

The second phase of the experiment had a similar path. The device this time never disappeared completely from the network, in fact, is possible to see that the behaviour is a bit more standard both for RSSi and OLSRd, by viewing the data, also CWND and packet in Flights suffered less data loss as the movement was slower and the destination a couple of meter closer than in the first phase.

The reaction coming from the TCP layer is of course faster than the routing protocol reaction, but the hole of performances lasted for 130 seconds. Furthermore the initial reaction caused some data loss.

A protocol able to exploit information coming from all the layers to reconfigure parameters in order to optimize and align those behaviours may be able to

improve the throughput of the whole network.

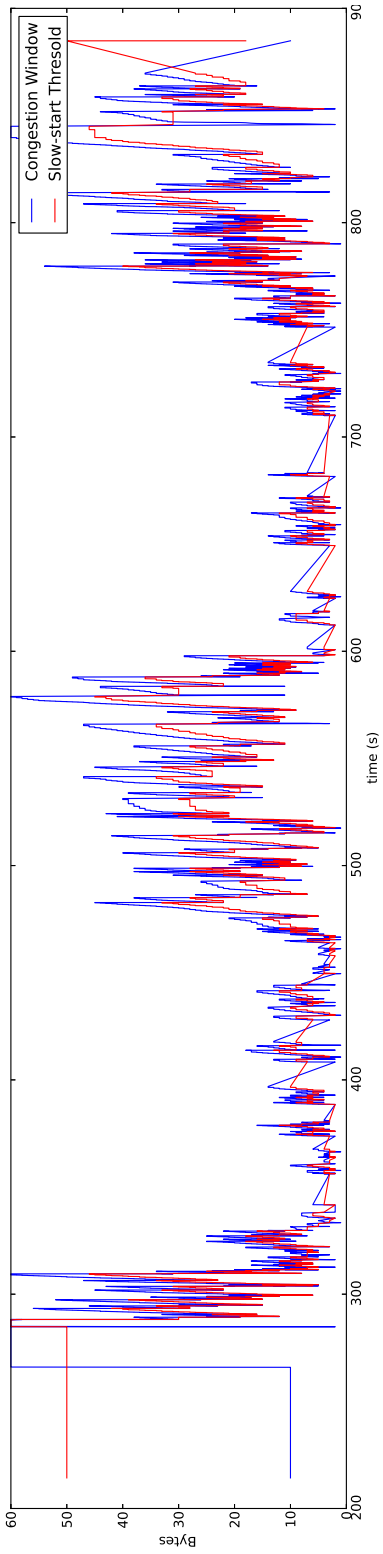

Figura 7.6: Congestion Window and Slow Start threshold values

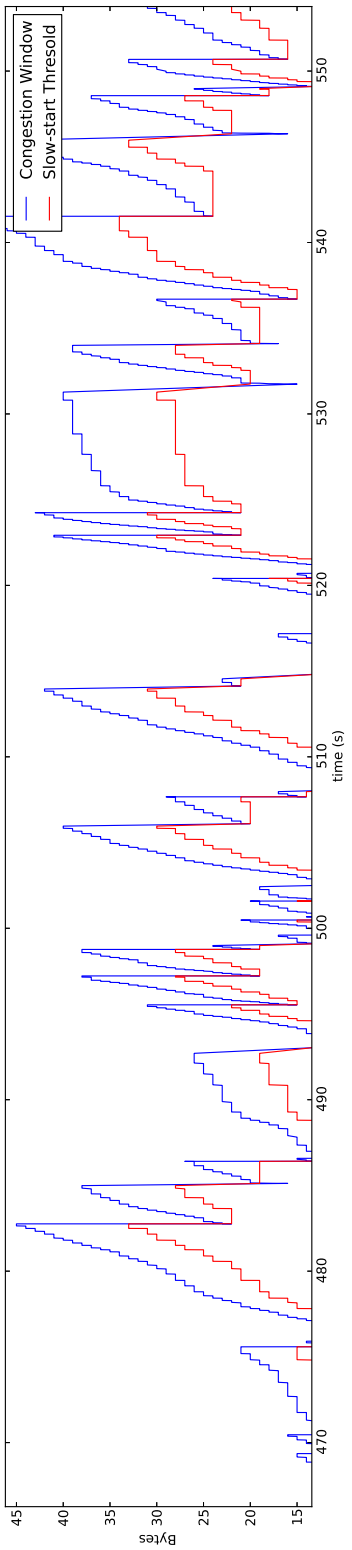

Figura 7.7: Congestion Window and Slow Start threshold values when one hop routing is restored

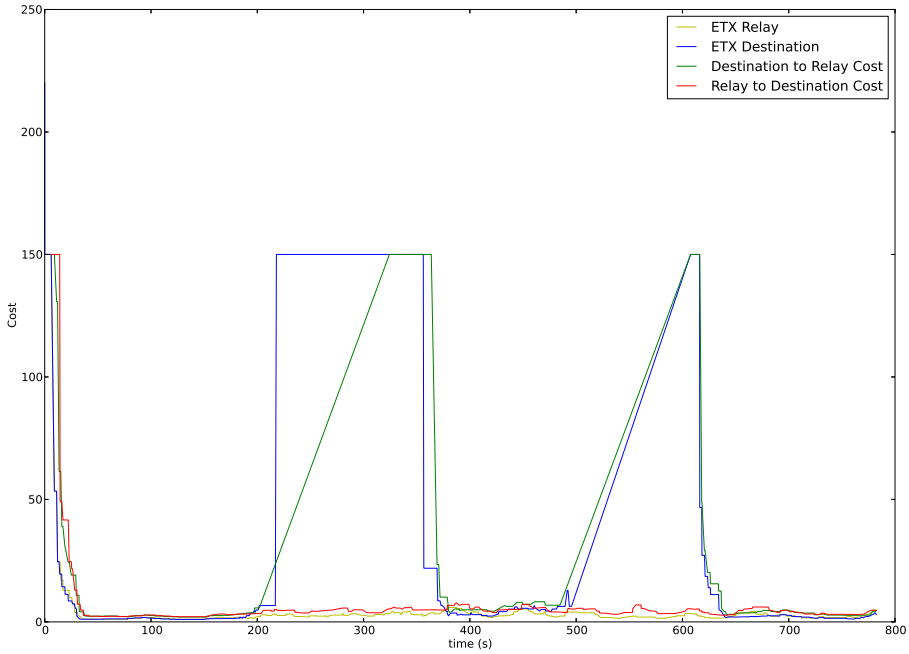

Figura 7.8: ETX and Cost to reach Destination and Relay

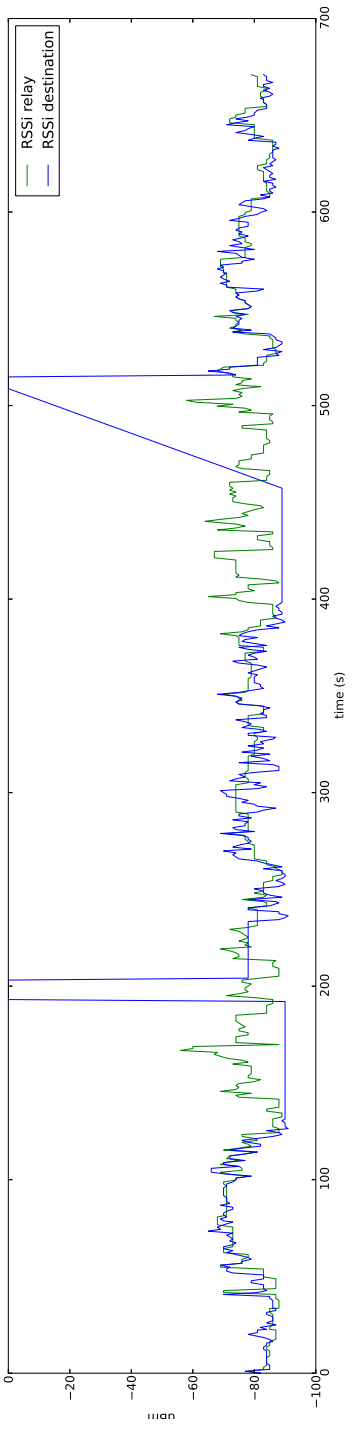

Figura 7.9: RSSi Values from the point of view of the sender

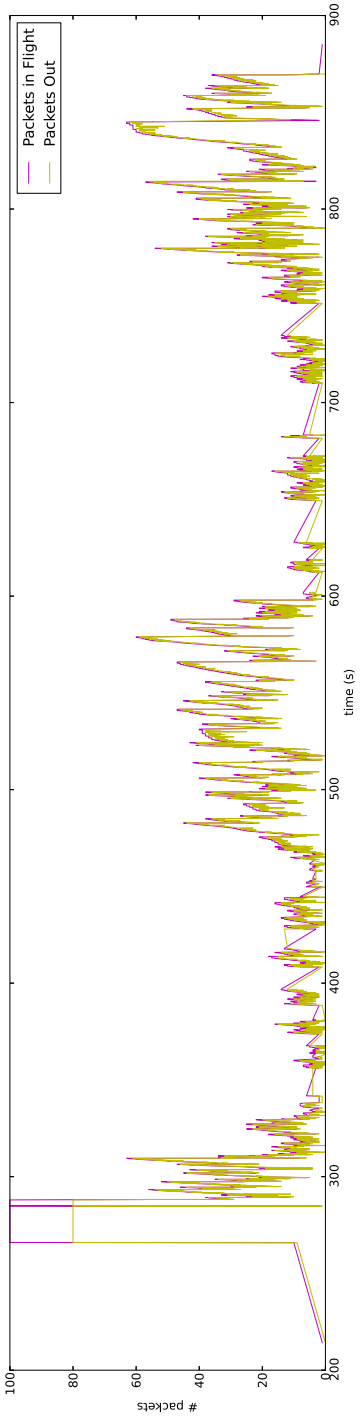

Figura 7.10: Number of packets in flights compare to the packet sent

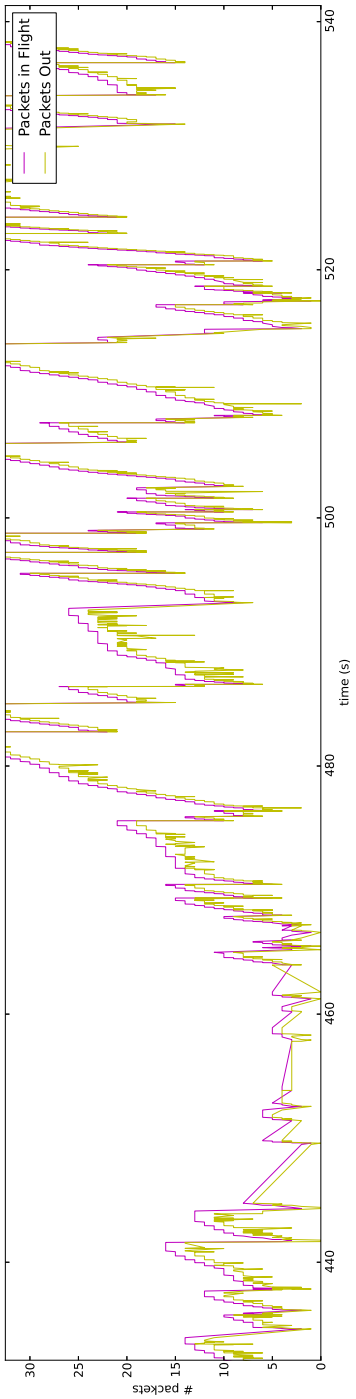

Figura 7.11: Packets in flight detail when the signal of the destination is again available

Capitolo 8

Conclusion

This project had the goal to provide a MANET network made with some mobile nodes, and develop onto it a platform able to setup, manage and analyse the network behaviour. The testbed developed is multiplatform as is currently deployed in a heterogeneous network, and was already used in some projects regarding Routing protocol analysis for Mobile Ad Hoc Networks and it will be used as a support to create a Cognitive Network structure.

Once all the needs that can come from a Cognitive network study was understood, the development of the COGNET testbed was achieved by a modular and layered design pattern that meet the requirements of the project. The Library and the kernel module related to it are able to crate and exploit an Ad-Hoc network over Android devices or mixed network. The integrated routing protocol is OLSRd, that can be activated or deactivated. Currently most of the goal of the project are achieved and the project is ready to be expanded for new needs. As a surplus the testbed developed had the ability to set some network parameters and display live the behaviour of the network in some charts. Those ability are very time saving when a whole experiment have to be setup on many nodes.

The testbed was successfully exploited for some indoor tests as well as some outdoor experiments. The main differences between those two experiments was mainly the lack of fixed stations to check the current situation and the amount of mobility in the outdoor experiments. The result were promising and show the correlation between

8.1 Future work

Currently this testbed was already exploited for another couple of project, and the results were promising for the future of this project. This is the first stage of this project, and can be considered as a working draft for future improvement and fixes. A lot of new features has to be imported in order to make this testbed as complete as possible:

- New routing protocols;
- Routing protocol massive switching;
- Real data flow instead of IPERF;
- OLSRd dynamic parameters flooding instead of node per node editing;

8.1.1 Routing Protocols

This testbed was built to be as modular and flexible as possible. Is possible to add as many routing protocols as desired just by cloning the OLSR class and adapt to the new needs in order to manage, start and stop the routing protocol. This features can follow the same flow of OLSRd and will probably need only some minimal graphical changes in order to maintain a clean interface.

8.1.2 Routing protocol massive switching

All the network, or a portion of it, must be able to switch from one routing protocol to another using the experiment manager tool. This ability can be extremely useful to test the dynamic evolution of networks over different routing protocols. This ability can be eventually useful to Cognitive Network protocols that are going to be developed.

8.1.3 Real data streaming

The current situation use IPERF as tool to load the network and make tests in two conditions: relaxed network and stressed network. The cons with this approach is that the data can't be verified and the flow is not natural. A video streaming client-server architecture was planned to be developed, by using the

devices camera or stored files. As the project had other crucial milestones in its early stage, was not possible to include this portion and it has been postponed to a next development stage.

8.1.4 OLSRd parameters

Unfortunately the data collected via the dynamic reconfiguration of OLSRd parameters were not promising. With some analysis was possible to discover that OLSRd ignore those modifications, unless they happens through all the nodes in the same time. This patch must be part of a future work as the continuous parameters adjustment of each layer involved is a key feature for a Cognitive Network protocol. This problem was found only in the very end of the project, during the data analysis. On each node the parameters were changed but the timers were not aligned due to the constraint explained above.

Bibliografia

[1] Ieee standard for information technology–local and metropolitan area networks–specific requirements–part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications - amendment 8: Medium access control (mac) quality of service enhancements. *IEEE Std 802.11e-2005 (Amendment to IEEE Std 802.11, 1999 Edition (Reaff 2003)*, pages 1–212, Nov 2005.

[2] Daniel Aguayo, John Bicket, Sanjit Biswas, Douglas SJ De Couto, and Robert Morris. Mit roofnet implementation. 2003.

[3] Ian F. Akyildiz, Won-Yeol Lee, Mehmet C. Vuran, and Shantidev Mohanty. Next generation/dynamic spectrum access/cognitive radio wireless networks: A survey. *Computer Networks*, 50(13):2127 – 2159, 2006.

[4] Todd R. Andel and Alec Yasinsac. On the credibility of manet simulations. *IEEE Comp*, pages 48–54, 2006.

[5] Andrea Bardella, Matteo Danieletto, Emanuele Menegatti, Andrea Zanella, Alberto Pretto, and Pietro Zanuttigh. Autonomous robot exploration in smart environments exploiting wireless sensors and visual features. *annals of telecommunications - annales des tÃ©lÃ©communications*, 67(7-8):297–311, 2012.

[6] L. Barolli, M. Ikeda, G. De Marco, A. Durresi, and F. Xhafa. Performance analysis of olsr and batman protocols considering link quality parameter. In *Advanced Information Networking and Applications, 2009. AINA '09. International Conference on*, pages 307–314, May 2009.

[7] John Bicket, Sanjit Biswas, Daniel Aguayo, and Robert Morris. Architecture and evaluation of the mit roofnet mesh network.

[8] K.R. Chowdhury and T. Melodia. Platforms and testbeds for experimental evaluation of cognitive ad hoc networks. *Communications Magazine, IEEE*, 48(9):96–104, Sept 2010.

[9] T. Clausen and P. Jacquet. Optimized Link State Routing Protocol (OLSR). RFC 3626, The Internet Society, October 2003.

[10] Federal Communication Commission. Spectrum policy task force. *Report ET Docket No. 02-135*, 2002.

[11] S. Corson and J. Macker. Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations. RFC 2501, The Internet Society, January 1999.

[12] M. Danieletto, G. Quer, R.R. Rao, and M. Zorzi. On the exploitation of the android os for the design of a wireless mesh network testbed. In *Military Communications Conference, MILCOM 2013 - 2013 IEEE*, pages 1032–1038, Nov 2013.

[13] Nandita Dukkipati, Matt Mathis, Yuchung Cheng, and Monia Ghobadi. Proportional rate reduction for tcp. In *Proceedings of the 11th ACM SIGCOMM Conference on Internet Measurement 2011, Berlin, Germany - November 2-4, 2011*, 2011.

[14] David Ehringer. The dalvik virtual machine architecture. *Techn. report (March 2010)*, 2010.

[15] C. Gomez, D. Garcia, and J. Paradells. Improving performance of a real ad-hoc network by tuning olsr parameters. In *Computers and Communications, 2005. ISCC 2005. Proceedings. 10th IEEE Symposium on*, pages 16–21, June 2005.

[16] Simon Haykin. Cognitive radio: brain-empowered wireless communications. *Selected Areas in Communications, IEEE Journal on*, 23(2):201–220, Feb 2005.

[17] Friedrich K. Jondral. Software-defined radio-basic and evolution to cognitive radio. *EURASIP Journal on Wireless Communication and Networking*, 2005.

[18] Tianji Li, DouglasJ. Leith, Venkataramana Badarla, David Malone, and Qizhi Cao. Achieving end-to-end fairness in 802.11e based wireless multi-hop mesh networks without coordination. *Mobile Networks and Applications*, 16(1):17–34, 2011.

[19] Riccardo Manfrin, Andrea Zanella, and Michele Zorzi. Crabss: Calradio-based advanced spectrum scanner for cognitive networks. *Wireless Communications and Mobile Computing*, 10(12):1682–1695, 2010.

[20] S.M. Mishra, D. Cabric, C. Chang, D. Willkomm, B. van Schewick, A. Wolisz, and R.W. Brodersen. A real time cognitive radio testbed for physical and link layer experiments. In *New Frontiers in Dynamic Spectrum Access Networks, 2005. DySPAN 2005. 2005 First IEEE International Symposium on*, pages 562–567, Nov 2005.

[21] J. Mitola and Jr. Maguire, G.Q. Cognitive radio: making software radios more personal. *Personal Communications, IEEE*, 6(4):13–18, Aug 1999.

[22] T.R. Newman and T. Bose. A cognitive radio network testbed for wireless communication and signal processing education. In *Digital Signal Processing Workshop and 5th IEEE Signal Processing Education Workshop, 2009. DSP/SPE 2009. IEEE 13th*, pages 757–761, Jan 2009.

[23] T.R. Newman, A. He, J. Gaeddert, B. Hilburn, T. Bose, and J.H. Reed. Virginia tech cognitive radio network testbed and open source cognitive radio framework. In *Testbeds and Research Infrastructures for the Development of Networks Communities and Workshops, 2009. TridentCom 2009. 5th International Conference on*, pages 1–3, April 2009.

[24] Jon B. Postel. Telnet protocol specification. Internet RFC 854, May 1983.

[25] G. Quer, H. Meenakshisundaram, B.R. Tamma, B. S. Manoj, R. Rao, and M. Zorzi. Using bayesian networks for cognitive control of multi-hop wireless networks. In *MILITARY COMMUNICATIONS CONFERENCE, 2010 - MILCOM 2010*, pages 201–206, Oct 2010.

[26] D. Raychaudhuri, I. Seskar, M. Ott, S. Ganu, K. Ramachandran, H. Kremo, R. Siracusa, H. Liu, and M. Singh. Overview of the orbit radio

grid testbed for evaluation of next-generation wireless network protocols. In *Wireless Communications and Networking Conference, 2005 IEEE*, volume 3, pages 1664–1669 Vol. 3, March 2005.

[27] P.D. Sutton, J. Lotze, H. Lahlou, S.A. Fahmy, K.E. Nolan, B. Ozgul, T.W. Rondeau, J. Noguera, and L.E. Doyle. Iris: an architecture for cognitive radio networking testbeds. *Communications Magazine, IEEE*, 48(9):114–122, Sept 2010.

[28] R.W. Thomas, L.A. DaSilva, and A.B. MacKenzie. Cognitive networks. In *New Frontiers in Dynamic Spectrum Access Networks, 2005. DySPAN 2005. 2005 First IEEE International Symposium on*, pages 352–360, Nov 2005.

[29] R.W. Thomas, D.H. Friend, L.A. DaSilva, and A.B. MacKenzie. Cognitive networks: adaptation and learning to achieve end-to-end performance objectives. *Communications Magazine, IEEE*, 44(12):51–57, Dec 2006.

Acknowledgements

First of all thanks to my family for the support and the comprehension for this period where my working time was doubled.
Many thanks goes to Matteo, as this project took a lot of effort and time from his other projects. The collaboration was complex sometimes because of my work and his other tasks but the final result was definitely satisfactory.
Along with him I would like to thank all the Signet members for they help and the knowledge they shared with me. Their support was very appreciated and extremely useful in many occasions.
A big thank goes to Paolo at Beenz for his support, his python skills and more than anything else the extra time he has left to me to finish this document.
Thanks to Joulia to have tried to lovely kill me a couple of time for my working night times. A thought also goes to my flatmates that endured my lack of effort in house cleaning in the last months.