A Framework for the Evaluation of Protocols and Services in Ad-Hoc Networks

by

Pablo Martí-Gamboa, BSc (Hons)

A Dissertation submitted to the University of Dublin,

in partial fulfillment of the requirements for the degree of

M.Sc. Computer Science

2006

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Pablo Martí-Gamboa

September 11, 2006

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Pablo Martí-Gamboa

September 11, 2006

Acknowledgments

I would like to thank my supervisor Dr. Stefan Weber for his advice and invaluable guidance throughout the dissertation, without him this would not have been possible. To Berta, for her love and support during the last months of this work. Lastly, I would like to thank my family for their unconditional support over the course of my college years.

Pablo Martí-Gamboa

University of Dublin, Trinity College

September 2006

A Framework for the Evaluation of Protocols and Services in Ad-Hoc Networks

Pablo Martí-Gamboa,
University of Dublin, Trinity College, 2006

Supervisor: Dr. Stefan Weber

Current research into solutions for mobile ad-hoc networks (MANETs) has produced an abundance of protocols, applications and services. The performance of these solutions depends on a variety of parameters such as the number of nodes that intend to send at the same time, size of packets that are transferred between nodes, offered load and/or interference from other sources. Reports of experiments that are performed to evaluate proposed solutions need to include these parameters in order to facilitate repeatability and verification through the community. However, the facilitation of repeatability and the gathering of this information that is involved in experiments in real-world scenarios is very complex. This complication has resulted in evaluations of

solutions to be limited mostly to simulations. In this paper, we propose a framework for the evaluation of solutions in MANETs that aims to address the issues of extensibility, information collection and repeatability. We will present an analysis of a number of existing projects and proposed characteristics, this information will be used to design and implement a framework that incorporates this points.

Contents

Acknowledgments **iv**

Abstract **v**

List of Tables **xi**

List of Figures **xii**

Listings **xiii**

Chapter 1 Introduction **1**

1.1 Background 1

1.2 Credibility of MANET Simulations 2

1.3 Motivation 3

1.4 Aims 4

1.5 Dissertation Roadmap 4

Chapter 2 State of the Art **6**

2.1 The ideal testbed 6

2.2 APE 7

2.2.1 Measured Parameters 9

2.2.2 Issues 10

2.3 DAMON 11

2.3.1 Measured Parameters 13

2.3.2 Issues 13

2.4 ORBIT 14

2.4.1 Measured Parameters 16

2.4.2 Issues 18

2.5 Comparison 19

2.6 Scenarios 20

2.6.1 Scenario 1 20

2.6.2 Scenario 2 20

2.6.3 Scenario 3 21

2.7 Summary 22

Chapter 3 Design **23**

3.1 Design goals 23

3.2 Overall Architecture 25

3.3 Network layer 25

3.4 The Event Dispatching System 27

3.5 Plugin system 28
3.6 Security 29
3.7 Experiments 30
3.8 Logging System 32
3.9 Summary 32

Chapter 4 Implementation **33**

4.1 Implementation environment 33
4.2 The Network Layer 34
4.3 The Event Dispatching System 35
4.4 The Plugin System 37
4.5 Security 39
4.6 Experiments 40
4.7 Logging System 41
4.8 Statistics 43
4.9 Location Information 45
4.10 Traffic Generator 45
4.11 Summary 46

Chapter 5 Evaluation **47**

5.1 Test Setup 47
5.2 Experiments 48
5.3 Evaluation of Lycaon's features 49
5.4 Evaluation of network performance 49

5.5 Evaluation against existing systems 51
5.6 Summary 54

Chapter 6 Conclusions **55**

6.1 Summary 55
6.2 Completed Work 56
6.3 Future work 56
6.3.1 Traffic generator 56
6.3.2 Graphical interface 57
6.3.3 Distributed time synchronisation 57
6.3.4 Support for distributed mechanisms research 58
6.3.5 Plugin System 58
6.3.6 Minor Improvements 58

Chapter A Example APE choreography **60**

Chapter B Lycaon's BlindBroadcast Plugin **64**

Chapter C Experiment file used in evaluation **66**

Bibliography **68**

List of Tables

2.1 Comparison of parameters measured in each platform 19

5.1 Comparison of system capabilities 53

List of Figures

2.1 APE's architecture 8

2.2 Sinks and agents in operation 12

2.3 Orbit's hardware 14

2.4 Orbit's experiment support architecture 15

3.1 Lycaon's architecture 25

3.2 Lycaon's signals 28

5.1 Traffic distribution during experiment 50

5.2 Iperf's UDP result 51

5.3 Lycaon's TCP TG result 52

Listings

4.1 Event dispatching example 36
4.2 Distribution mechanism's hook 38
4.3 Sample intronspection session 40
4.4 Sample Lycaon experiment 42
4.5 python-wifi wrapper 44
A.1 Example APE choreography 60
B.1 BlindBroadcast plugin 64
C.1 Experiment file used in evaluation 66

Chapter 1

Introduction

1.1 Background

Current research into solutions for mobile ad-hoc networks (MANETs) has produced an abundance of protocols, applications and services [26, 20]. These solutions are evaluated to determine their characteristics and suitability to solve the problem they aim to address. The performance of proposed solutions for MANETs depends on a variety of parameters such as the number of nodes that intend to send at the same time, size of packets that are transferred between nodes, offered load or interference from other sources. These parameters and environmental factors need to be collected during the execution of an experiment. Reports of experiments that are performed to evaluate proposed solutions need to include these parameters in order to facilitate repeatability and verification through the community.

However, the facilitation of repeatability and the gathering of all information that

is involved in experiments in real-world scenarios is very complex. This complication has led to a situation where much of the research in ad-hoc networks has been limited to evaluations through simulation. The simulators used typically aim to represent the different software and hardware components within the system as well as the physical environment in which they operate. While this provides a useful method of validation for the first stages of the design or implementation of a novel protocol, the simplified assumptions made of the physical environment limit the scope of what can be achieved from them [4, 16]. For instance, the phenomena of gray-zones [18] were only discovered through experiments conducted in a real-world environment. It is therefore necessary to complement simulation studies with real-world experiments to identify phenomena that would otherwise go unnoticed within a simulator.

1.2 Credibility of MANET Simulations

The inaccuracy of MANET simulations is a well known problem in the MANET community that has produced dozens of publications about it [2, 17]. The inherent problem with MANET simulations is that it is very difficult to capture all the factors that may influence the performance and behaviour of a MANET experiment. In addition, even if the developers did come up with a model that accurately reflected all this factors in the result of the simulation, this simulation would be very resource intensive and expensive to compute. Thus all simulators do some generalisations about wireless propagation, protocol stack interaction, etc.

Obviously this generalisations affect the outcome of the experiment, yielding results

that may greatly differ from the results of running the same experiment in a real-world environment. Cavin et al. [4] compared three of the most popular simulators –NS2, Opnet and GoMoSiM– by implementing a simple flooding protocol in each one. If each simulation reflected reality, the results should have been identical. However, the results showed large differences between the simulators. Only one of these simulations could have been right because they gave three signicantly distinct results, if one is right, the other two must be wrong. The worrying part is that this differences suggest that all three are probably wrong.

1.3 Motivation

A number of recent projects have investigated the design and implementation of testbeds for the evaluation of MANET solutions in real-world scenarios. These projects have identified factors that need to be measured, methods to describe experiments, and approaches that facilitate repeatability. Factors such as the number of participating nodes, the topology of a network, etc. influence the setup and the environment of an experiment and need to be recorded during an experiment. Various projects [28, 23] have proposed methods to describe experiments in the form of choreographies. All these systems define additional aspects that are needed to provide repeatable experiments.

However, these systems focus on the investigation of new aspects of an evaluation while paying less attention to other aspects. This concentration on the investigation of new aspects has led to a large number of factors being identified but has failed to produce a system or collection of systems that enables researchers to describe experiments

in a form that supports repeatability and offers mechanisms to collect all information associated with an experiment.

The primary aim of this dissertation is to design and implement an extensible framework that allows MANET researchers to carry out repeatable experiments in the real-world. The design incorporates the relevant features of the systems described in chapter 2 and addresses the open issues that have been identified in these systems.

1.4 Aims

The aims of this dissertation are,

- To identify a set of desirable characteristics that a framework for the evaluation of protocols and services in ad-hoc networks should fulfil.
- To design and implement a framework which fulfils the requirements gathered in the above point and that allows to carry out repeatable MANET experiments in a real-world scenario.
- To evaluate the features of the framework against similar existing solutions.

1.5 Dissertation Roadmap

The rest of this dissertation report is laid out as follows:

Chapter 2 State of the art, provides a review and discussion of existing ad-hoc testbeds highlighting the strong and weak points of each one.

Chapter 3 Design, gives an overview of the design and structure of the proposed system.

Chapter 4 Implementation, this chapter will discuss the implementation of the components that together support Lycaon –the name of the framework.

Chapter 5 Evaluation, this chapter will discuss the experiments carried out and results. Further, a feature comparison against the existing solutions is presented.

Chapter 6 Conclusions, this chapter summarises the work presented and points to possible further extensions and improvements.

Chapter 2

State of the Art

Evaluation of ad-hoc networks is a lively field in the MANET community that has produced a variety of systems. In the following sections we will review a number of these systems, discuss the experiences that researchers have reported and highlight arguments for and against each one.

2.1 The ideal testbed

Kiess et al. [15] define their criteria that an ideal testbed should provide based on their experiences and reports from other research projects. Apart from desired characteristics such as modularity and portability, Kiess et al. identified three key requirements that a testbed should fulfil:

1. Reproducibility: Researchers should be able to reproduce published results based on the data provided by authors of a publication – an implicit assumption of

scientific work. Reproducibility represents a challenge for research conducted in the area of ad-hoc networks because the reproduction of the exact same topology and movement patterns during an experiment are difficult.

2. Comprehension: In order to be able to understand and explain the results of an experiment, it is necessary to record as much information about an experiment, its environment and other factors as possible.

3. Correctness: The results of an experiment can easily become useless because of broken tools, errors with the setup, and random errors while conducting the experiment. Kiess et al. argue that the perfect testbed should provide tools to verify that any given experiment has yield valid results.

2.2 APE

The Ad hoc Protocol Evaluation (APE) testbed [28] provides a system to design experiments based on scenarios, to measure data such as throughput, delay, etc, and to correlate and process these data after an experiment. The testbed consists of a Linux distribution that contains patches for the Linux kernel and a set of programs and scripts. It includes support for a number of routing protocols such as AODV [22], DSR [13], OLSR [12], LUNAR [27] and TORA [21]. Figure 2.1 shows a high-level overview of APE's architecture.

The focus of the APE testbed is the repeatability of experiments. In order to support repeatability, APE introduces the concept of choreography: A choreography

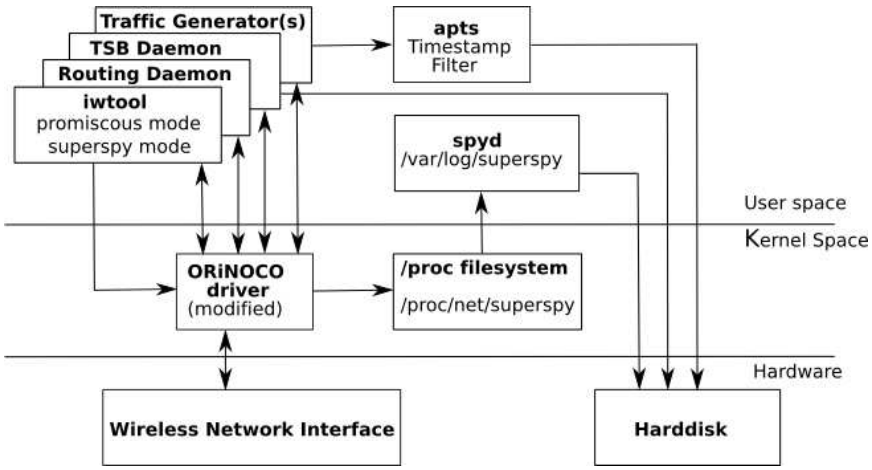

Figure 2.1: *APE's architecture*

describes the movement and action of individual nodes as instructions that are given to the user of a node. Appendix A contains a sample of an APE choreography. These instructions are defined before the start of an experiment and relayed to users as the experiment progresses. Individual choreographies can be reused and aim to ensure that topology changes are consistent throughout a set of experiments.

APE synchronises the clock of nodes participating in an experiment by issuing periodic broadcasts from designated nodes. This synchronised time is used to timestamp events.

APE records information at the level of 802.11 frames. The capture of this information is supported by a modified driver for Lucent ORINOCO adapters. This dependency limits the system to Lucent ORINOCO adapters at the time of writing and requires the development of additional modified drivers to support additional wireless adapters.

2.2.1 Measured Parameters

APE allows a number of parameters of an ad-hoc network to be measured: Virtual mobility, link change, connectivity, packet loss rate and hop count. The following paragraphs explain these parameters in more detail.

Virtual Mobility (VM): VM is based on signal changes in links between nodes. The measured signal quality of a transmission is computed with Q in dB $= \alpha - 33 * log(dist/\beta)$ [14]. Empirical experiments led to a loss model: Distance D between $node_j$ and $node_i$ in the range of 0.5 and 65 meters is computed as $D_j(node_i) = 4 * 10^{\frac{40-0.9*Q_j(node_i)}{33}}$. VM is then calculated as the difference of virtual distance between two consecutive time intervals. This model obviously only works with ORiNOCO cards.

VM is used to estimate the degree of similarity between two experiments and to provide a way to evaluate the effect of changes on other parameters such as packet loss, etc. This parameter represents changes between individual experiments in terms of mobility and movement, but also in terms of connectivity between nodes, since VM is dependent on inter-node connectivity.

Link Change: Link Change represents the number of link changes per time unit. The length of a time unit is dependent on the mobility during the experiment. The higher the mobility of nodes is, the smaller is the time unit. A lot of movement means a higher metric and a larger "temperature"[1] in the network.

[1]Temperature is a way to measure the mobility during an experiment. If a MANET network has a high degree of mobility, it is said that the temperature is high.

Connectivity: Connectivity in terms of neighbours is a metric that reflects the number of nodes a particular node is connected to.

Packet Loss: Packet loss occurs when one or more packets of data travelling across a network fail to reach their destination. Packet loss rate and its complement, reception rate, are given as a ratio that represents the correlation of successful and unsuccessful deliveries.

Hop Count: In ad-hoc networks, a hop represents one portion of the path between source and destination. Usually a packet will be relied several times before arriving to its destination. Generally speaking, the more hops data must traverse to reach their destination, the greater the transmission delay incurred. It must be noted however that lower hop counts does not necessarily means higher throughput.

2.2.2 Issues

A number of issues have been identified that have not been addressed by the APE system:

Scalability: The scalability of APE is limited because of the manual creation of choreographies. The setup of these choreographies is time consuming and cumbersome for large experimental setups.

Choreography Dependant: In order to carry out an experiment in APE, a choreography must be created before an experiment is undertaken. It is not possible to do

so without a choreography.

Hardware Dependencies: APE uses a patched Linux kernel that only works with Lucent's Orinoco cards. Thus, it has a strong hardware dependency that refrains APE from working in heterogeneous hardware/Linux kernels. This also affects handcrafted metrics such as VM, that are tested and developed with a particular hardware in mind, and only work with such hardware.

2.3 DAMON

The Distributed Architecture for MONitoring multi-hop mobile networks (DAMON) [23] has been developed at UC Santa Barbara. DAMON[2] is implemented in the programming language Perl and distributed as an installable software package for both Linux and Windows.

DAMON uses an agent-sink architecture to monitor mobile networks. A sink is a node with "unlimited" resources that collects statistics from nodes in its coverage are and generally remains in a static location or moves little compared to the mobility of other participating nodes in the experiment. Sinks advertise their presence through periodic beacons and are considered to be resilient to failure. Sink clients are called agents. These agents may be a resource-limited devices depending on the characteristics of the scenario. Figure 2.2 shows sink beaconing and agent-sink association in DAMON.

There are two types of nodes in DAMON: *fully-capable* nodes –nodes that have

[2]This review covers DAMONv1. DAMONv2 is under development and not available as of September 2006

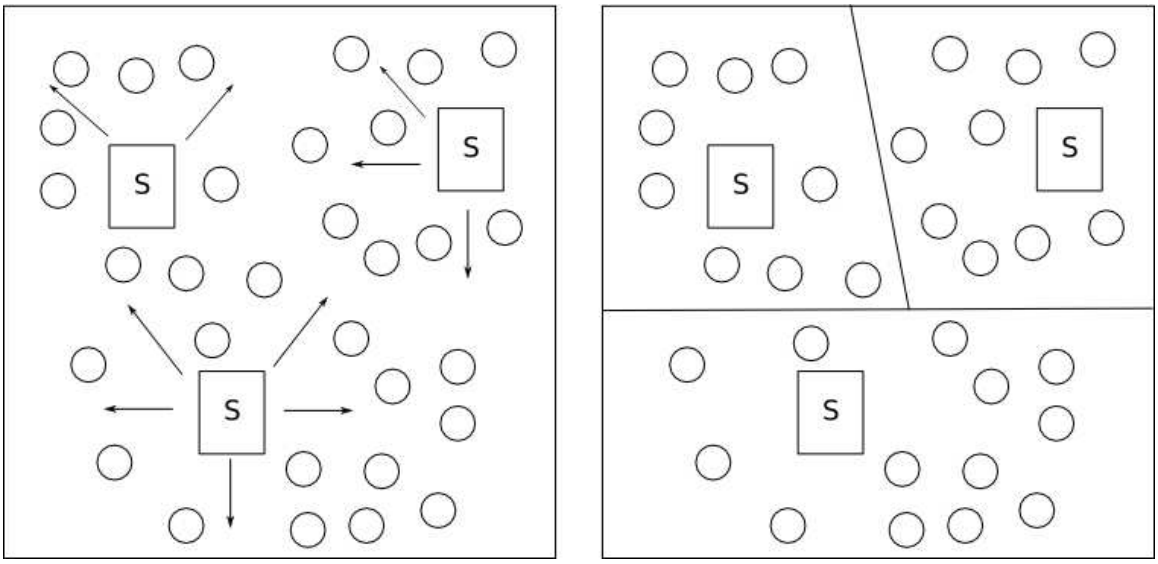

Figure 2.2: *Sinks and agents in operation*

"unlimited" disk space, battery and processing power– and nodes that cannot dedicate these resources like a PDA, but also participate in the experiment. DAMON supports both complete coverage, where all participating nodes are sinks and log traffic, and limited coverage scenarios –where only a limited number of nodes are sinks.

DAMON distinguishes between Time Dependant Digests (TDD) and Time Independent Digests (TID). TDDs have higher priority than their counterparts, and are sent in a best-effort fashion. They are real-time information like energy left, identity of the node, etc. Multiple TDDs can be aggregated and sent together. TIDs use TCP as transport mechanism and are the logs of traffic and the sequence numbers used to keep track of what has been successfully delivered to a sink.

2.3.1 Measured Parameters

Packet Delivery Ratio: Packet delivery ratio (PDR) is computed as the ratio of the number of data packets received by the destination nodes divided by the total number of data packets transmitted by the source nodes. Packet delivery ratio is a useful parameter to compare how two different routing protocols perform under, for example, mobility.

Route Discovery Latency: Route discovery latency (RDL) is the time that takes a routing protocol to discover a route to a given host. In reactive routing protocols such as AODV, RDL is quite high as AODV does not discover a route until a flow is initiated and may use an expanded-ring search, making it even worse.

Network Throughput: The rate at which a computer or network sends or receives data. Throughput is low in ad-hoc networks because of using omni-directional antennas –a node can forward only a single packet at a time resulting in poor spatial reuse. Throughput is useful to see how a routing protocol copes with mobility or bursts of data while delivering packets.

2.3.2 Issues

A number of issues have been identified that have not been addressed by DAMON:

Repeatability: DAMON does not support repeatability. It is not one of its design goals and because of this, the system does not, for example, provide the possibility to describe experiments through choreographies.

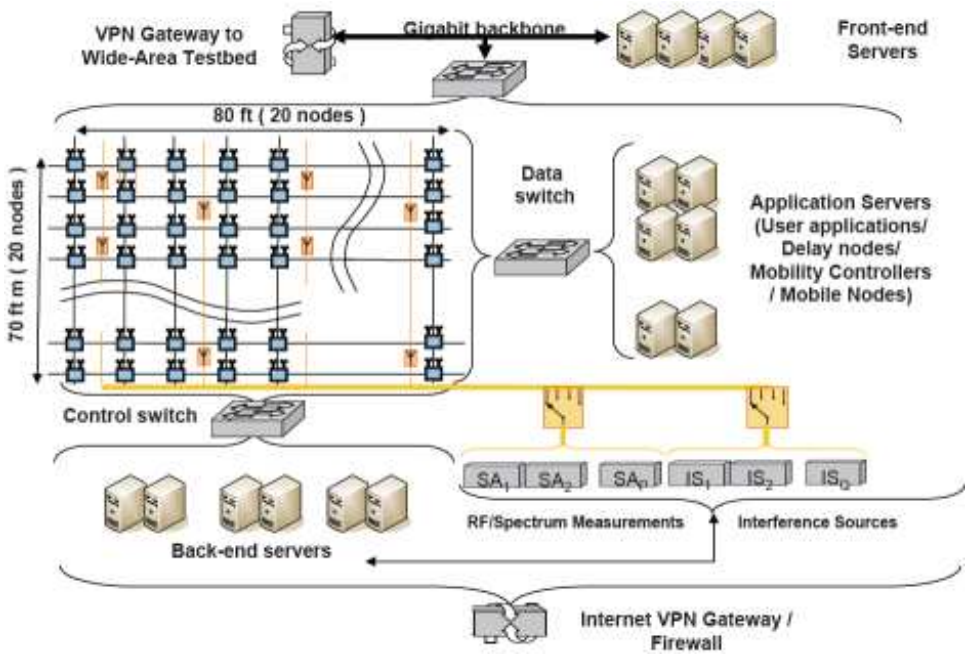

Figure 2.3: *Orbit's hardware*

Available Routing Protocols: The available version of DAMON is limited to the monitoring of networks that employ AODV as routing protocol.

2.4 ORBIT

ORBIT [24] is an indoor radio grid testbed designed for reproducibility of experiments. The project is a collaborative effort between several university research groups and industrial partners. ORBIT is being developed and operated by WINLAB, Rutgers University. ORBIT is open for research groups and institutions. Access to the testbed can be granted to run experiments on it both locally and remotely through a web interface.

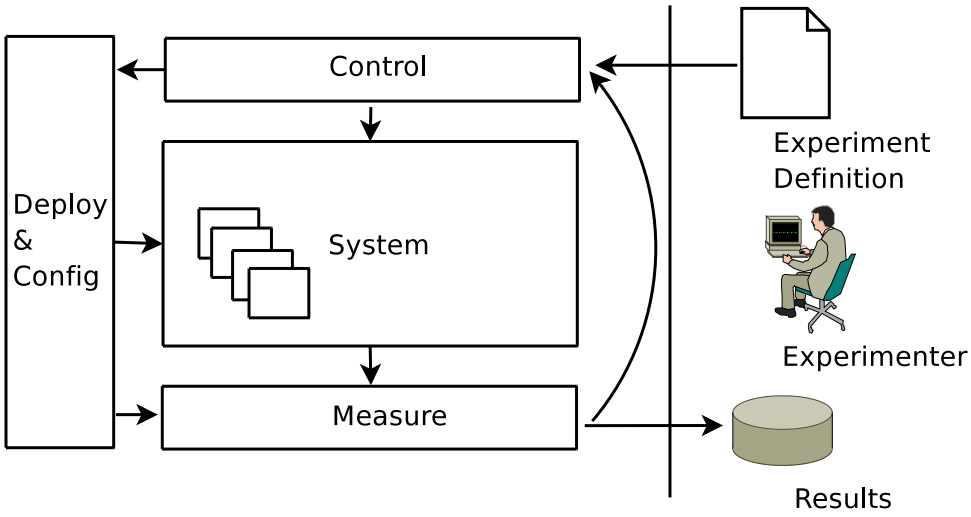

Figure 2.4: *Orbit's experiment support architecture*

ORBIT features a two-dimensional grid of 400 802.11 radio nodes as illustrated in figure 2.3 – extracted from ORBIT project's webpage [1] – which can be dynamically interconnected into specified topologies with reproducible wireless channel models. Each node is connected via multiple high-speed Ethernet links for transfer of applications, control and management information. By having dedicated ethernet links to each node, real-time monitoring of data parameters without affecting the experiment results is possible.

Experiments in ORBIT are written in the Ruby language. Different scripts define the topology and communication pattern between nodes, a list of applications needed, the role that each node will play during the experiment and also offers the possibility of changing some parameters dynamically in the middle of the experiment. Figure 2.4 shows the architecture that supports the experiment deployment.

ORBIT represents a very a complex evaluation system, the discussion of all features

is beyond the scope of this paper. We will focus on a few selected features that we consider important. The Orbit Measurement Library (OML) is a distributed client/server measurement framework. In OML, tuples of measurements are saved in a type safe format named XDR [25]. This data is serialised and sent to a collection server where it is inserted in a database for late analysis. Researchers can then access the results of the experiment through any software that speaks ODBC. ORBIT also features the Orbit Traffic Generator (OTG), a tool to generate traffic of data and test network performance. OTG works along with OML, reporting statistics to it. In addition, ORBIT uses NTP to synchronise time amongst nodes and Frisbee [11] to image disks.

Despite being a fixed testbed, mobility is supported in ORBIT through the use of virtual mobiles (VMs). A VM is a node off the grid but connected to it. The VMs emulate mobility by forwarding traffic with selected nodes, if the VM is supposed to be at location x, the mobility controller will encapsulate and forward its packets to node x which just decapsulates and transmits them. On the other hand, packets received by node x, will be encapsulated and forwarded to the VM. If after a period of time, VM "moves" to location y, the VM will use node y as described above.

2.4.1 Measured Parameters

Receive Signal Strength Indicator (RSSI): is an indicator of the strength of the received signal measured in dBm or mW, depending on the card model. The problem is that IEEE's RSSI definition is fuzzy. Vendors provide their own levels of accuracy, granularity, and range for the actual power and RSSI. On the other hand, other vendors do not provide an RSSI value and instead convert directly from dBm to

percent.

TX_Power: Transmission power –measured in dBm– at which a wireless NIC operates; the greater the power, the bigger the range at which the NIC can communicate. TX_Power is artificially reduced in some testbeds in order to simulate different topologies and to overcome spatial problems.

Noise: Background noise level present in the experiment. It is possible to dynamically modify the noise present in the experiment. This is useful for example in experiments at layer 2 to see how a MAC protocol performs under the presence of noise. In ORBIT noise is injected to emulate real-world wireless network topologies onto the testbed.

Network Throughput: The rate at which a computer or network sends or receives data. Throughput is low in ad-hoc networks because of using omni-directional antennas –a node can forward only a single packet at a time resulting in poor spatial reuse. Throughput is useful to see how a routing protocol copes with mobility or bursts of data while delivering packets.

Offered-Load: According to Duffield et al [5], offered-load is the total required bandwidth that clients would use if no constraints were present i.e. if limitless bandwidth would be available.

Number of Frame Retransmissions: Number of frame retransmissions is collected at layer 2 of the OSI stack. Using a tool called *athstats* along with OML, it is possible to measure for example the number of frames that were dropped due

to errors at the physical layer. This metric targets experiments that investigate issues at the data link layer.

2.4.2 Issues

A number of issues have been identified that have not been addressed by the ORBIT testbed:

Mobility: ORBIT is a static solution and emulates mobility through the use of VMs. This form of emulation brings with it similar complications to those exhibited by simulations. The delay in the communication between the nodes performing the experiment and the mobile nodes reporting their locations can interfere with the accuracy of the emulation and the nodes performing the experiment do not experience the same environment as the mobile nodes.

Real-world Issues: Being ORBIT a middle ground between reality and simulation, real-world issues such as gray-zones [18] may not be discovered through experiments in this environment.

Flexibility: The design and implementation of ORBIT is closely coupled to the hardware configuration and setup of the ORBIT installation. This dependency makes it difficult to replicate experiments in other locations and raises the effort that is necessary to adapt ORBIT's mechanisms to situations unforeseen in its design.

2.5 Comparison

The three platforms discussed in the previous sections have each their individual focus: APE aims to support repeatability in real-world measurements, DAMON aims to gather information about data transfers in ad-hoc networks, and ORBIT aims to provide an environment for repeatable experiments for wireless communication. Table 2.1 gives an overview of the parameters that can be measured with the individual platforms. As we can see, only few parameters are measured by all platforms. Two platforms measure the network throughput, one of the most obvious to measure. Packet loss and number of frame retransmissions are related, but while the former operates at layer 3, the later measures at layer 2.

	APE	DAMON	ORBIT
Virtual mobility	x	-	-
Link change	x	-	-
Connectivity	x	-	-
Packet loss	x	-	-
Hop count	x	-	-
Packet delivery ratio	-	x	-
Route Discovery Latency	-	x	-
Network throughput	-	x	x
RSSI	-	-	x
TXPower	-	-	x
Noise	-	-	x
Offered load	-	-	x
Number of frame retransmissions	-	-	x

Table 2.1: *Comparison of parameters measured in each platform*

2.6 Scenarios

In the following paragraphs we will describe a number of scenarios that we consider representative for the use of testbeds for the evaluation of MANET solutions. These scenarios are then used to discuss the characteristics and features of the architecture of the framework.

2.6.1 Scenario 1

Bob would like to monitor the traffic of an ad-hoc network during a conference on ad-hoc networks, as described by Ramachandran et al. [23]. The participants will install Lycaon on their laptops or PDAs. The parameters that are to be collected are the maximum and average throughput, number of control packets, etc. Because of the characteristics of the scenario, only one sink is used. Bob specifies everything in an experiment file and runs it. The experiment is signed with Bob's private key transparently. The participating nodes will be sure that the experiment actually comes from Bob. Once the experiment is finished, he analyses the resulting logs with the aid of scripts.

2.6.2 Scenario 2

Alice is interested in performing an experiment that involves computing a new metric that is not present in Lycaon, such as APE's link change metric. To do so, she writes a new plugin that computes link change metric using Lycaon's plugin API. This plugin is deployed on all participating nodes seamlessly using the distribution plugin when

the experiment begins. Alice specified in the experiment that nodes should log the information in a database, she invokes a script that handles the processing of the different logs and outputs a single report of the experiment. After the experiment has finished, Alice uses a visualisation plugin that will plot the link change metric against time.

2.6.3 Scenario 3

Charlie would like to carry out an experiment in a network without the knowledge of its topology. To do so, Charlie issues a discovery command that returns the identification and position of each node in the network that is running Lycaon. With this information, Charlie creates a choreography scenario for an experiment. In the scenario, Charlie specifies the time, position, and movement of nodes, the actions they perform, the collection of the data etc. When the scenario is complete, the description is distributed to the participating nodes. Charlie chose a *blind broadcast scheme* as distribution plugin which distributes all commands to the participating nodes using broadcast. Charlie specified that instead of logging to a database, nodes and sinks will log files. After the files have been transferred to a central server, sorting and merging scripts produce a single file with the results of the experiment. Charlie can either analyse these results with the aid of an existing visualisation plugin or create a new one that is adapted to his needs.

2.7 Summary

A set of desirable features that a MANET testbed should provide has been presented. This chapter has also introduced a sound review of the state of the art in MANET testbeds highlighting the interesting features of each one. This information and the requirements extracted from the three presented scenarios will be incorporated into the design presented in next chapter.

Chapter 3

Design

This chapter is going to present a set of goals that have been identified as desirable features for our proposed framework followed by a description of a design that satisfies this requirements.

3.1 Design goals

The design goals of Lycaon follow the characteristics described by Kiess et al. [15]. They aim to incorporate the relevant features of the systems described in Chapter 2 and address the open issues that have been identified in these systems.

1. Distributed, Modular Architecture: Lycaon aims to evaluate protocols and applications in one of the most distributed environments: MANETs. Thus, Lycaon's architecture needs to consist of elements that can be distributed among the nodes that participate in an experiment. The individual elements should consist of

components that can be distributed throughout the network by an adequate distribution mechanism and that can be assembled depending on the experiment that is to be performed.

2. Data Gathering Capabilities: The provision of data gathering capabilities is one of Lycaon's main aspects. The framework should provide functionality to capture as much data as possible about an experiment and the environment it takes place in. In order to do so, the framework will have to be extensible to allow the implementation of new functionality to capture classes of data that was not anticipated during the design of the framework. Lycaon will also need to provide mechanisms that allow to interface with legacy software in order to incorporate existing tools that are used to gather data in experiments for wireless communication.

3. Scripting Capabilities: Experiments that are to be carried out within the framework should be described in a form that allows them to be repeated with minimum of effort. Also, the description of experiments should facilitate the comparison of results that were produced by a set of experiments. The form of these descriptions should be human readable and free from requirements for specific editors.

4. Portability & Heterogeneity: The nodes that make up an ad-hoc network may consist of variety hardware and operating systems. The framework needs to take this into account and be constructed so that experiments can be adapted to available hardware and software. The implementation of the framework needs to be easily portable to new platforms and existing descriptions of experiments should be applicable to a variety of platforms.

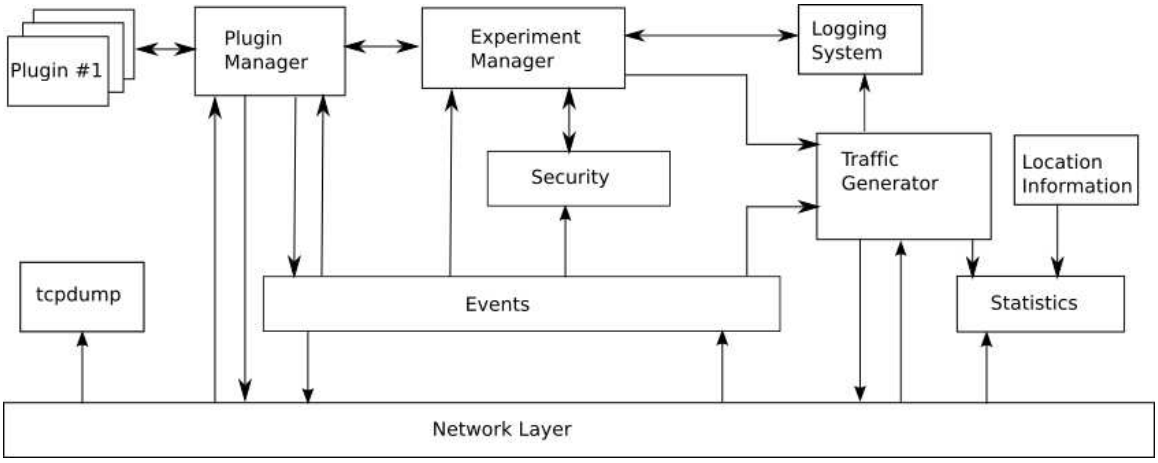

Figure 3.1: *Lycaon's architecture*

3.2 Overall Architecture

The system will be made of a number of individual modules each with a single specific role. A diagram detailing the relationships is provided in Figure 3.1. Each module is discussed in turn below.

3.3 Network layer

The network layer (NL) sits at the bottom of Lycaon's stack. Communication between Lycaon nodes is over UDP. The reason behind using UDP as communication protocol instead of TCP is that because of the nature of ad-hoc networks, TCP is nearly broken [10]. Lycaon is meant to be used in large multi-hop networks where vanilla TCP just does not work that well. Even though UDP is an unreliable protocol, if a message is corrupted or is lost on its way to a node, chances are that another node will have

received and forwarded it. TCP is only used in two places right now: uploading results of an experiment (FTP) and while using traffic generators (TG), either an iperf-like TG, or Lycaon's own python TG.

Upon reception of a packet, the NL will forward the packet to the loaded distribution plugin, the distribution plugin decides if the packet should be processed or not –in case we already saw this packet, more on this later– and whether to forward the packet or not –depending on the plugin's algorithm. In order to be able to decide if a packet should be processed or not, each node maintains two hash maps that keep track of the "open" and "closed" packets. Closed keeps the ids of the packets seen in the last ten seconds, this avoids forwarding already forwarded packets. If an action requires a response it will register the packet id in the open list. If an instance of that packet is received, the callback registered with it will be fired with the packet data as arguments and the entry in the open list will be deleted. The entries in the closed list are kept for 10 seconds, this will avoid having a closed list that grows and grows as time passes.

The distribution plugin may or may not forward the packet –again all depends on the plugin's algorithm, but it will always instruct the NL whether a packet must be processed or not. The NL will process the packet with the appropriate handler for the packet type. There are several types of packets in Lycaon, with their respective handlers. The type of the packets range from simple beacons announcing position and local time –the only type of packets that is not forwarded by default in Lycaon– to experiments, plugin requests/replies, etc. Despite their differences, each handler operates similarly. If the NL needs to communicate with other module –e.g. when an experiment is received– it will signal the other module through the signal dispatcher

mechanism.

3.4 The Event Dispatching System

Lycaon uses the well-known *Louie* dispatcher to send signals between modules. Figure 3.2 shows how the signals in Lycaon interact, and what modules communicate through them.

For example, consider that the NL receives an experiment and is not running an experiment previously. The EM will process the experiment and check that the dependencies required for that experiment are solved. Once it has checked that has all the plugins required to start it will send an *EXP_IAM_OK* signal that will make the NL to send an *IAM_OK* packet and will wait for an *ALL_OK* signal. Now consider that a *PLUG_REQ* packet is received from a neighbour node at the NL. The node needs plugin X and after being checked this plugin is present in this node's plugin folder. The node will reply with a *PLUG_REP* packet with the plugin data as payload. In the requester side, once the plugin is received, this will provoke a *PLUGIN_REC* signal that will in turn make the EM check if it has all the necessary plugins to start, if that is the case it will send an *IAM_OK* packet.

The first node that figures out that the rest of the nodes are ready to start will signal the NL to send an *ALL_OK* packet that will signal the rest of the nodes to start the given experiment in Y seconds. When the experiment is complete, the node will send an *EXP_FIN* signal that will make all the listening modules – like tcpdump – to stop working and process its data.

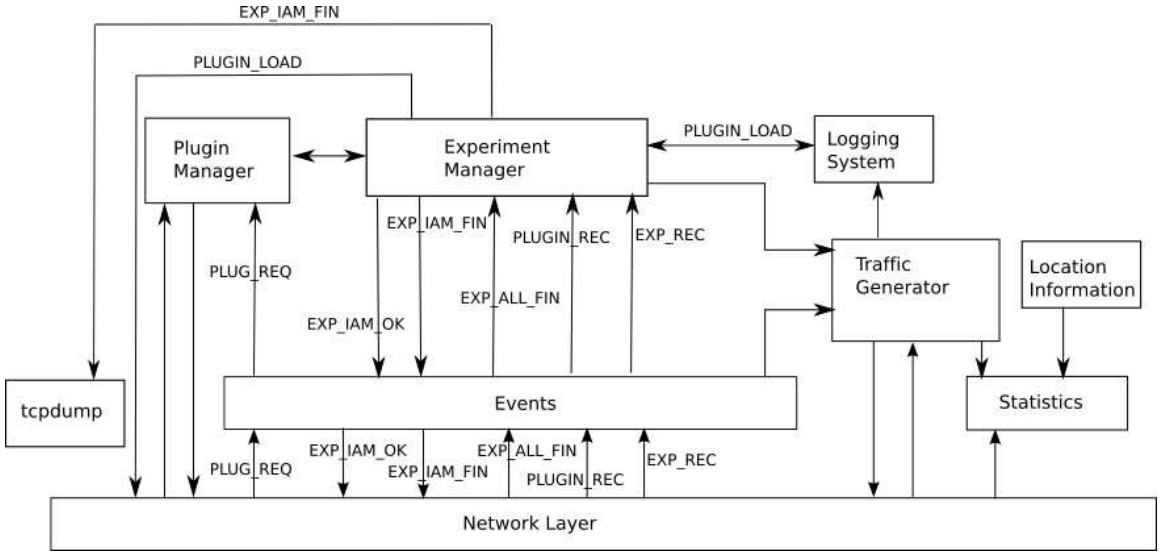

Figure 3.2: *Lycaon's signals*

3.5 Plugin system

The plugin system is composed of one class, the PluginManager (PM). The PM takes care of loading plugins, registering and unregistering plugins, etc. PM is the only entry point in Lycaon to access the plugin infrastructure. Plugins in Lycaon are identified by name and version in order to support repeatability of experiments.

An implicit requirement for the plugin system is that it should feature an automatic distribution mechanism capable of transparently deploying plugins to nodes that require one or more plugins necessary for an experiment. The reasoning behind this requirement is that in networks with, say, more than 5 nodes having to manually deploy the required plugins to each node would be a lengthy process.

If an experiment uses a given plugin, the ExperimentManager will query the PM if it is present in the plugin folder. If the plugin is present it will just return an instance

of it to the requester, otherwise it will request it from the network. Missing plugins are requested through a *PLUG_REQ* packet with the name and version of the plugin as payload. If a node receives the packet and has the desired plugin it will reply with a *PLUG_REP* and the plugin data as payload.

The plugin architecture allows Lycaon's users to extend it at runtime. It must be said however, that developing an application completely extensible is a difficult and not straightforward task. Lycaon is extensible but to some extent, the following list summarises the points where Lycaon can be extended:

- Distribution plugin: Researchers can write their own distribution plugin, allowing them to see, for example, the behaviour of a new broadcast algorithm.
- Result parser plugin: Researchers can write their own data processing plugin, allowing them to specify what variables to use and how to plot them.
- Storage plugin: By writting a new storage plugin a Lycaon user can add a new database/format for saving experiment results.
- Traffic Generator: Researchers can write their own traffic generator plugin in pure Python.

3.6 Security

Lycaon was designed to run as a daemon service in MANET testbeds such as WAND [3], this means that Lycaon could be running for weeks or months waiting to receive the next experiment. A Lycaon experiment contains a choreography section where

potentially hazardous instructions to be executed are specified, thus it is necessary that when an experiment is received, the node should be sure that the given experiment has originated from the network "owner" and not from a malicious peer. Experiments then are cryptographically signed in order to ensure the authenticity of the experiment itself. Seting up this cryptographic measures is relatively easy and only requires a file to be present in the Lycaon folder, which is the public RSA key of the experimenter. In addition, Lycaon must open some devices as a privileged user in order to be able to change some wireless settings, this means that Lycaon must be run as root and must drop privileges as soon as posible.

3.7 Experiments

The choreography of distributed experiments modeled as activities in a distributed workflow is a very complex problem. It is quite difficult to design a format that is able to represent the ammount of data necessary to describe an experiment. One of Lycaon's main goals is that experiments should be human-readable and should be free from requirements for specific editors. This requirement lead to quickly dismiss a XML-based experiment format.

An experiment must contain three sections:

- a *metadata* section where things like dependencies, experiment information, etc. are specified
- a *choreography* section where actions to be run and the time that they are meant to be run are specified

- a signature field that contains the cryptographic signature of both the metadata and the choreography sections.

An inherent problem with choreographies in distributed systems is how to synchronise actions amongst nodes. Lycaon nodes' clocks are not synchronised, clock synchronisation in a distributed environment is a difficult problem that was out of the scope of this work. To tackle this problem, the time of the actions specified in the choreography is relative to the start of the experiment. This means that instead of executing action A at "1157378883.956636", action A will be executed 15 seconds after the start of the experiment.

Another problem to address is that, in contrast to Orbit, Lycaon is going to be used in networks where separate control channels to coordinate actions are not present. This makes necessary to devise a mechanism to coordinate the start of the experiment amongst the nodes. The solution is simple, each node while parsing the choreography of the experiment is able to see which nodes have actions specified for the given experiment. Thus, each node keeps a list of the participating nodes that will be updated as IAM_OK packets are received. The first node to have the participant list empty will signal to the rest that everybody is ready to start with a ALL_OK packet. This packet has a payload the time at when the experiment will start, again this time is relative in order to avoid synchronisation problems.

3.8 Logging System

The logging system in Lycaon is pluggable and allows to write a new plugin to save experiment results in the desired medium or database. The only entry to the logging system is through the *_log_results* hook in the experiment module. This will in turn call the loaded plugin's *log_results* method. If a Lycaon user wants to use an unsuported database, say PostgreSQL, is a matter of writting a plugin that implements the *IStorage* interface and copy it to the plugin folder.

Additionally, it also must be created another plugin that implements the *IStorageIterator* interface in order to be able to analyse results saved in the new format. This interface only has a method *iterresults* that returns an iterator to iterate over the results.

3.9 Summary

The design of the system presented in chapter is highly modular and all the important concerns are separated into individual modules. Some critical parts such as extensibility and authentication have been identified and addressed in the presented design.

Chapter 4

Implementation

This chapter describes the work carried out during the implementation phase of the project. The core activity during this phase was the implementation of the architecture described in chapter 3. This chapter describes the details of the implementation only. Evaluation of this work is contained in chapter 5.

4.1 Implementation environment

Lycaon has been developed in the Python programming language. Python is a well known dynamic object oriented language with a clear syntax that is very easy to read and pick up. The reasoning behind choosing Python is simple: Lycaon is a relatively complex system that had to be developed in a short time frame. A Python programmer can be up to 5-10 times more productive than a Java programmer according to some comparisons [6, 7, 8]. Being high performance not a requirement for this project, the choice of a dynamic language that interfaces very well with C –thus with Linux/Unix–

was obvious.

Apart from other well-known open-source projects such as *gpsd* or *Louie*, Lycaon uses the *Twisted* [9] framework for network communication. Twisted is an event-driven asynchronous framework that imposes a novel programming style for developers used to multi-threaded applications, but one capable of great efficiency under heavy loads. The key while programming with Twisted is that all blocking operations should return a *deferred* –also known as *promise* or *future* in languages like E [19]– this deferred will have one or more callbacks registered that will be executed with the response of the blocking operation as argument.

4.2 The Network Layer

While designing Lycaon, we wanted to follow a design similar to DAMON's sink-agent architecture. Then while the implementation phase it become clear that even though the sink-agent architecture looks like a nice solution to the problem of supporting resource-limited devices, uploading data during an experiment would affect the result of the experiment itself, making its results useless. So the agent-sink architecture was dropped and instead the model used in APE was used. It is then assumed that all the nodes are resource-unlimited devices –i.e. laptops– and each node logs its own traffic, this log will be uploaded once the experiment is finished to the FTP server specified in the experiment file –usually the same laptop that launched the experiment.

The NL is composed of two basic classes, *SinkBase* and *SinkNode*. SinkBase provides some of the basic infrastructure necessary for communication, such as the open

and closed hashmaps, methods to build and send packets, garbage collection of the closed list, etc. On the other hand, SinkNode extends SinkBase providing a pluggable distribution mechanism, the handlers for the different types of packets in Lycaon and also connects the appropriate methods to their respective signal handlers for the signal dispatching system.

As Lycaon nodes communicate through UDP broadcasts, one of the first problems that aroused was how to generate packet ids in a distributed multi-hop network that can be comprised of several hundreds of nodes. The classical model followed in TCP is to use sequential packet ids –up to 2^{32}-1, unfortunately this is not possible to apply in Lycaon as there could be a possibility that two packets sent by different nodes were sharing the same packet id, thus provoking a collision. At the moment Lycaon is using as packet id the time since epoch, this could be changed in the future to a combination of the IP address of the node plus a sequential packet ID *ala* TCP.

Another interesting problem is in situations where a node sends a request, for example a plugin request, how to register a callback for a response –packet id– that we do not know beforehand. To overcome this, in a plugin request it is included the packet id that the respondant should use in case it has the given plugin.

4.3 The Event Dispatching System

The event dispatching system used in Lycaon is *Louie*. Louie is the only python dispatching system that comes with support for Twisted. The different signals are declared in the *consts.py* module as strings. A method then, is connected to a given

signal through Louie's *connect* method. If during runtime the signal is sent from any other module, this will execute the registered method(s) with the arguments passed from the invoking *send* method. If the invoker needs a response, the send command returns a tuple of a wrapper and a deferred. A callback can be registered with this deferred as usual.

Listing 4.1 shows the handler for an experiment received. A signal, *EXP_RECV*, is sent through Louie and will be received at the ExperimentManager. If the EM accepts the experiment it will reply with the id of the experiment, otherwise with None –that is equivalent to Java's null.

```
def _process_experiment(self, pkt_id, host, payload):
        logmsg('processing experiment %s from %s' % (pkt_id, host))
        d = louie.send(K.EXP_RECV, None, payload, pkt_id)[0][1]
        def set_exp_running(resp):
            if resp:
                assert self.state == IDLE
                self.state = BUSY
                self.exp_id = resp
        d.addCallback(set_exp_running)
```

Listing 4.1: *Event dispatching example*

4.4 The Plugin System

The plugin system is implemented on top of Twisted's plugin system. Twisted's plugin system enables developers to write extensible programs that can be enhanced in a way that is loosely coupled: only the plugin API is required to remain stable. The only entry point to Twisted's plugin system is through the *getPlugins* method, this method accepts two parameters: interface and package. Interface specifies the interface that we are interested in, and package is a python module that specifies the path where plugins will be searched. This method returns an iterator of plugins that the developer will use accordingly. Appendix B shows one of the simplest distribution plugins you can write, a "blind" broadcast plugin –i.e. forward all packets unless we have already seen an instance of the packet.

The problem with Twisted's plugin system is that in Lycaon we are usually interested in just one plugin instead of a collection of plugins. Not only that, but also the fact that plugins in Lycaon had to be identified by a tuple of name and version. The solution was to create a layer on top of Twisted's plugin system with some methods that allow to retrieve plugins identified by name and version, register a plugin received from the network or get the data of a given plugin. All this is contained in the *PluginManager* class.

When all the plugin requirements necessary for an experiment are satisfied, a *LOAD_PLUGIN* signal is sent through the signal dispatcher. This signal has as argument a Python dictionary where all the plugins to load are specified. Interested receivers – extensible parts in Lycaon – will receive it and load the specified plugin,

avoiding a signal for each plugin to load.

```
# Send the packet to the network plugin
# if its the first time we see this packet it will return 1
if self._plugin.process_packet(data, (host, port), self):
   # find what handler to use
   handler = getattr(self, '_process_%s' % pkt_type, None)
   if handler:
      try:
         handler(pkt_id, host, payload)
      except LycaonException, why:
         logerr('Exception received: %s' % why)
   else:
      logerr('Unknown packet %r I will not forward it' % data)
```

Listing 4.2: *Distribution mechanism's hook*

As we previously stated in section 3.5, creating a truly extensible program needs a great deal of careful design and engineering that unfortunately could not be devoted to Lycaon. Lycaon is then extensible but only to some extent. Lycaon has a number of *hooks* where the loaded plugin's method(s) will be called to process some data, the list of possible plugins has already been reviewed in section 3.5. For example, listing 4.2 shows the hook present in the *datagramReceived* method of the SinkNode class. The plugin will extract the packet id out of the data and will decide whether the data should be forwarded or not and whether the packet should be processed or not by the

NL.

4.5 Security

Lycaon uses public-key authentication in two places: while signing or checking the signature of an experiment and to grant access to the python console that allows to inspect each Lycaon module during an experiment – more on this later. In both places, the public RSA key of the owner is used to authenticate either the experiment or the ssh connection to the manhole. This key should be deployed in the *.lycaon* folder of each node running the experiment, if the key is not present Lycaon will not start.

Lycaon must be run as root in order to access the network devices with root permissions –a mandatory requisite for the statistics module that needs to be able to modify wireless settings. It is a well-known fact that running applications with unlimited privileges should be avoided in Unix systems as this represents a big security risk. Lycaon drops privileges as soon as it binds to a port. Further, when *tcpdump* is executed, it also drops privileges as soon as it has opened the network interface in capture mode.

The security module also provides an interesting feature that was not planned during the design. The Twisted API provides the infrastructure necessary to build a service that can be seamlessly integrated with a Twisted application, and provides a Python administrative shell that can be accessed through telnet/ssh. From this shell, it is possible to inspect inside the components of the application like if it was a local python shell running a script. The access to this service in Lycaon is through ssh and only accepts connections from authenticated users –by the public RSA key. It

is possible to see what packet ids are in the closed and open lists, interact with the statistics module, etc. Listing 4.3 shows a snippet of an intronspection session.

```
$ ssh 192.168.0.1 -p 6022
...
>>> dir()
['K', 'LycaonProtocol', 'Node', 'NodeFactory', 'SinkNode',
'TwistedDispatchPlugin', '__builtins__', '__doc__', '__file__', '
   __name__', '_version', 'get_mh_factory', 'logmsg', 'louie', '
   make_application', 'protocol']
>>> SinkNode.closed
{'1157562027.8940899': '1157562027.8941', '1157562037.8947201': '
   1157562037.8947289', '1157562007.8967841': '1157562007.8967941',
   '1157562017.894099': '1157562017.8941081', '1157562047.8951459':
   '1157562047.8951559'}
```

Listing 4.3: *Sample intronspection session*

4.6 Experiments

As stated in section 3.7, Lycaon experiments have two main parts: metadata and choreography. For the metadata section, Python's *ConfigParser* was used. This module implements a basic configuration file parser language which provides a structure similar to that found in Microsoft's Windows INI files. The choreography section was

implemented in a handcrafted language relatively easy to read. Listing 4.4 shows a sample Lycaon experiment. Line 2 instantiates a new Experiment object, an Experiment object just has three instance variables –metadata, choreography and signature– which are plain Python strings. The metadata section is pretty straightforward to read and understand, there are sections –denoted by [section]– and each section has pairs of *name: value* entries. Line 11 is the reportback section, where the ip address, username and password to be used for uploading the results of the experiment are specified. Line 17 is the beginning of the choreography. Choreography entries always follow this format: *when who what arguments*. For example, line 18 states that five seconds after the start of the experiment, node 192.168.0.1 will execute iperf with the "-u -c 192.168.0.2 -t 10 -i 1 -b 11m" arguments –that is connect via UDP to 192.168.0.2 for 10 seconds, reporting each second and with an offered load of 11Mbits. The *execute* directive will execute the arguments as command:arguments of command. The *tg-tcp* directive expects an ip address as argument, the ip the TG will connect to. Finally the *instr* directive prints a choreography message in the screen of the node *ala* APE.

4.7 Logging System

Lycaon's loggin system is compound of two different modules, one deals with logging the results of the experiment and the other with logging the traffic of an experiment. Although they are two different subsystems, they are included in the same section.

```
1  from lycaon.experiment import Experiment
   experiment = Experiment()
3  experiment.metadata = """
   [info]
5  id: sampleexp
   author: Pablo Marti
7  version: 0.1
   [dependencies]
9  distmech: gossip3-50
   storage: sqlitedb-41
11 [reportback]
   host: 192.168.1.1
13 user: huno
   passwd: lycaon
15 [logging]
   tcpdump: yes"""
17 experiment.choreography = """
   005 192.168.0.1 execute iperf:-u -c 192.168.0.2 -t 10 -i 1 -b 11m
19 015 192.168.0.2 execute iperf:-u -c 192.168.0.1 -t 15 -i 1 -b 8m -N
   020 192.168.0.1 instr Move 30 meters to the west
21 070 192.168.0.1 tg-tcp 192.168.0.2"""
```

Listing 4.4: *Sample Lycaon experiment*

The logging system comes with two different plugins to save results, the *SQLiteDB* plugin –interface to SQLite– and the *TextDB* plugin –a text database. The plugin to be used will be specified in the experiment and no logging plugin will be enabled at the start by default. SQLite should be preferred as its dependencies are minimal and its features make it a worthy choice. For example, SQLite database files can be freely shared between machines with different byte orders. Compare this with ORBIT, where they use the XDR format to overcome this problem. It is our opinion that the approach taken by ORBIT is over-engineered and this solution is simpler.

The other module included here is *tcpdump.py* which is not pluggable because it did not make sense to make it. Tcpdump is present in almost every Linux/Unix machine and libcap is the *facto* standard for logging network traffic. Tcpdump will only be started if specified in the experiment; this makes it useful for scenarios where we are just interested in the traffic distribution of the network, or for scenarios where we are not interested in the tcpdump trace.

4.8 Statistics

The module *statistics.py* provides a dictionary-like –in Python jargon– interface to layer 2/3 information during the experiment as well as GPS –if running– data. The access to layer 2/3 is provided through two Python wrappers around the Linux wireless stack. The current functionality is limited to that provided by the wrappers. The only potentially blocking operations are those related to GPS, as the wrappers around the wireless stack are not blocking. In the case of GPS, the *GPSStats* class updates

itself each 5 seconds, thus caching the information for 5 seconds. When some GPS information is requested it is possible that is 5 seconds old, thus it should be tuned to a lower number in scenarios with high mobility.

```
class PyWifiStats:
    def __init__(self, iface):
        self.iface = Wireless(iface)
        self._getters = {'statistics': 'getStatistics',
                         'encryption': 'getEncryption',
                         'fragmentation': 'getFragmentation',
                         'powermanagement': 'getPowermanagement',
                         'rts': 'getRTS', 'retrylimit': 'Retrylimit'
   ,
                         'sensitivity': 'getSensitivity'}
    def __getitem__(self, key):
        if self._getters.has_key(key):
            return getattr(self.iface, self._getters[key])()
        raise KeyError
    def provides(self):
        return self._getters.keys(), None
```

Listing 4.5: *python-wifi wrapper*

Despite the fact that adding a new class to the statistics module is not as easy as plugging a plugin, the module has defined an interface that new classes should

implement if they want to be added to the module. For example, listing 4.5 shows the *PyWifiStats*, a wrapper around the *python-wifi* package. The instance variable *getters* is a dictionary that maps the parameters that this class announces as available to the respective methods in python-wifi's *Wireless* class. The method that must be implemented is *provides*, that returns a tuple with the "getters" and "setters" that the class provides. Adding a new wrapper around, say, a Galileo receiver is a matter of implementing this interface and adding the necessary lines of code in the *Stats* class.

4.9 Location Information

In the original design, LI was going to be made completely pluggable. Then it become clear that it was not worth the effort as the only LI hardware widely available is GPS. The package used to access GPS information is *gpsd*. As mentioned in section 4.8, the class *GPSStats* provides a dictionary-like interface to access GPS information. As accessing this information may block, the class updates itself every 5 seconds, thus caching the information for 5 seconds.

4.10 Traffic Generator

Lycaon's TG is contained in the *trafficg.py* module. The TG system is compound of three classes:

- *TGProtocol*: The protocol represents the server side of the TG.
- *TGFactory*: The factory for the protocol.

- *TGProducer*: The client part, where the algorithm to produce data is defined.

The TG system was originally designed to be pluggable, however because of lack of time this feature was dropped. At the moment the traffic generator only works with TCP connections, as Lycaon uses Twisted's Producer/Consumer API and is limited to TCP connections. The only method that should be changed in a pluggable implementation is *resumeProducing*.

4.11 Summary

The design presented in chapter 3 has been implemented with encouraging results. The system is modular and extensible in some parts and provides features similar to the systems presented in chapter 2 plus some of its own. This will be discussed in deep in chapter 5. A CD-rom with Lycaon's source code, API documentation and some misc documentation produced durante Lycaon's development can be found attached at the back of this work.

Chapter 5

Evaluation

This chapter will outline the experiments carried out, their goals, results and conclusions that can be drawn from them, firstly the equipment used and configuration is outlined.

5.1 Test Setup

The experiments where conducted using heterogeneous equipment with the same conguration:

- 1 Dell Latitude D410, 2GHz Pentium M Processor, 256MB of RAM
- 1 Dell Latitude D400, 1.3GHz Pentium M Processor, 256MB of RAM
- 1 Fujitsu B Series Lifebook, 500MHz Intel Celeron, 256MB of RAM
- 1 WAND Node, 933MHz Low Power Mobile Pentium III, 128MB of RAM

- All machines run a vanilla Linux 2.6.12 kernel
- Cisco Aironet 352 Cards in the D410 and the WAND node
- Orinoco gold cards in the D400 and the lifebook

These machines were were positioned such that there were all an equal distance apart. 802.11 channel 7 was used as this frequency was not used by any other 802.11 devices in the proximity. The environment where the evaluation was conducted was in the DSG lab in the second floor of the Lloyd Institute, Trinity College Dublin. This is a relatively noisy environment with dozens of other 802.11 sources, from both infrastructure based and ad-hoc networks.

5.2 Experiments

A series of experiments were carried out to evaluate the various components that make up Lycaon. The experiment file used to evaluate Lycaon is shown in appendix C. Although a single experiment file was used, it allowed us to evaluate Lycaon in two areas:

- Evaluation of Lycaon features
- Evaluation of network performance through traffic generators

5.3 Evaluation of Lycaon's features

While Lycaon was being developed, the ad-hoc network where it was tested was a two node network. With the hardware loaned from the DSG laboratory, this experiments tested Lycaon with a 4 laptop network in order to see if features such as experiment synchronisation, plugin distribution, etc. worked as expected with more nodes. The first experiment was to try the plugin distribution system, for this purpose the plugin *iperf3-62* was specified as a dependency in the experiment. Only one node, the D400, had the plugin – which was just a gossip3 plugin with an updated version number. The test worked out satisfactorily, and the plugin was transparently deployed to the rest of the nodes. When all the nodes had the dependencies satisfied, the experiment started simultaneously in all the nodes at the expected time. When all the nodes were finished, the experiment logs were uploaded to the specified FTP server with no problems whatsoever.

Another feature to test was the traffic distribution analyser. The tcpdump trace of the D410 during the experiment was analysed with the *tethereal.py* script to see the traffic distribution during the experiment. The result is shown in figure 5.1.

5.4 Evaluation of network performance

In this experiment we compared the results of the iperf traffic generator against the results yielded by Lycaon's own traffic generator. As stated in section 4.10, Lycaon only works with TCP, in addition we could only make iperf work with UDP traffic, TCP traffic in ad-hoc networks with iperf appeared to be broken never getting output

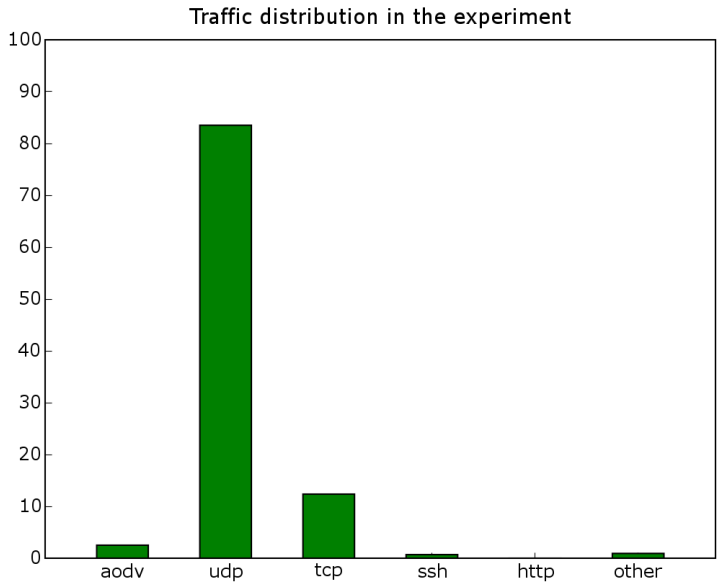

Figure 5.1: *Traffic distribution during experiment*

from the command execution. So this section is actually comparing iperf's UDP results against Lycaon's TCP TG results. Despite the fact that we are comparing different transport protocols, it is still interesting to see that the average of iperf command shown in figure 5.2 –3.8Mb/s– is similar to the average of Lycaon's TCP TG shown in figure 5.3 –3.4Mb/s. The difference can be explained due to several factors, first and most important TCP has congestion control mechanisms that make it a bit slower than UDP traffic. It could also be argued that because iperf is implemented in C++ and Lycaon's TG in Python, this makes iperf a bit faster than Lycaon's TG. This however we do not believe it to be true as while the experiment was running CPU usage by Python was never more than 20%, so the CPU usage –thus the overhead of using a dynamic language– was not the bottleneck.

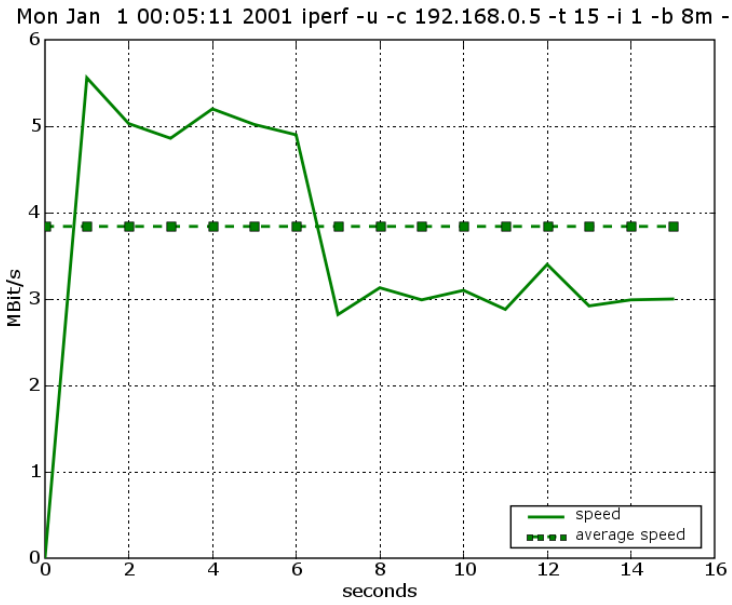

Figure 5.2: *Iperf's UDP result*

5.5 Evaluation against existing systems

Perhaps the features tested in the below evalution looks limited, but we argue that a great deal of the effort put in Lycaon has gone into intangible assets such as extensibility, security, reusability, etc. Lycaon may be undervaluated when compared to other systems' tangible assets such as more features or functionalities. This section is going to show what Lycaon, despite its young age, has already achieved when compared to similar systems. Table 5.1 shows a feature comparison of Lycaon against the three testbeds reviewed in chapter 2.

The first two features compared are APE specific, and despite the fact that we have already explained why having metrics that depend on hardware is bad in section 2.2.2,

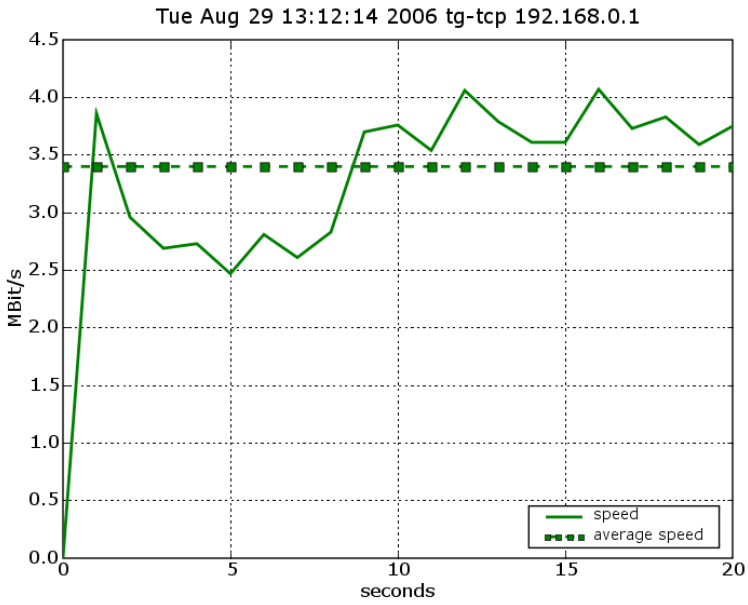

Figure 5.3: *Lycaon's TCP TG result*

they appear in the table for the record. Connectivity and Hop count are two metrics that could be extracted from the tcpdump trace, while the infrastructure to record the traffic log of an experiment exists, this feature is not currently present in *tethereal.py*. It should not be very difficult to implement this two metrics into the tethereal module. Packet loss can be extracted from the pywifi wrapper around the linux wireless stack, in section 2.2.1 we already shown that packet delivery ratio (PDR) is the complement of packet loss, so even though PDR is not measured by python-wifi, is a metric really straightforward to compute. On the other hand, route discovery latency is a more tricky metric to compute that Lycaon does not supports, and does not plan to support. Lycaon is already capable of reading and modifying in real-time metrics such as RSSI, TXPower, Noise, Offered load and Number of frame retransmissions thanks to the

	Lycaon	APE	DAMON	ORBIT
Virtual mobility	-	x	-	-
Link change	-	x	-	-
Connectivity	-	x	-	-
Packet loss	x	x	-	-
Hop count	-	x	-	-
Packet delivery ratio	x	-	x	-
Route Discovery Latency	-	-	x	-
Network throughput	x	-	x	x
RSSI	x	-	-	x
TXPower	x	-	-	x
Noise	x	-	-	x
Offered load	x	-	-	x
Number of frame retransmissions	x	-	-	x
Plugin architecture	x	-	-	-
Intronspection capabilities	x	-	-	-
Experiment authentication	x	-	-	-
Run as non root	x	-	-	?
Run as ad-hoc application	-	x	x	-
Run as a 24/7 service	x	-	-	x

Table 5.1: *Comparison of system capabilities*

wrappers around the linux wireless stack.

In addition, Lycaon has some unique features not present in other systems. For example, Lycaon is extensible in several parts thanks to a modular design and a plugin architecture. This plugin architecture can transparently locate and download missing plugins. Another nice feature of Lycaon is its introspection capabilities that allow a researcher to connect to Lycaon through ssh and get an administrative python shell from where she can inspect some parameters of Lycaon during an experiment.

Some of the other system, such as APE and DAMON, are "ad-hoc" applications

that its intended usage is run them as root, carry out several experiments and finish. In contrast, Lycaon was designed to run as a daemon service that could be weeks or months waiting to receive an experiment, thus Lycaon had to be made secure enough so that a malicious peer could not launch an experiment with potentially harmful instructions. Lycaon is the only system out of the four testbeds presented in this work that introduces security in its design from the beginning.

5.6 Summary

This chapter has presented an evaluation of Lycaon's features and a feature comparison with the testbeds introduced in chapter 2. Lycaon features match those present in the other testbeds and has introduced some desirable and innovative features not present in other solutions.

Chapter 6

Conclusions

This chapter will outline the work presented in this thesis. A great number of the goals and requirements identified in chapter 3 are satisfied by the implementation presented in chapter 4. Unfortunately, not everything that was planned could be implemented because of lack of time, this chapter will provide some insights for people interested in expanding Lycaon.

6.1 Summary

This work has introduced a review of the state of the art in MANET testbeds. Each presented system has been analysed and the arguments for and against each one highlighted. A set of desirable features not present in any platform and a design that satisfies it has been outlined. This design has been implemented and evaluated with satisfactorily results. Lycaon features not only matches most of the features present in the reviewed system but also introduces some features of its own.

6.2 Completed Work

An extensible MANET testbed has been devised, implemented and evaluated with encouraging results. Lycaon features a modular pluggable architecture that allows to extend its core at runtime through plugins. Plugin dependencies are solved transparently throughout the network. Apart from matching features present in similar solutions, Lycaon's design tackles problems such as authentication and security. Lycaon provides an infrastructure that researchers can use and extend for MANET experiments in the real-world.

6.3 Future work

There are many interesting directions in which Lycaon's core could be extended. We had several other features in mind that had to be cut because of lack of time, this section summarises some of them.

6.3.1 Traffic generator

The traffic generator that Lycaon features is a simple TCP TG that sends data until a hard limit in bytes is reached. The reason because the TG is limited to TCP is because the implementation uses Twisted's consumer/producer API and is limited to TCP connections. The original plan was to make TG pluggable so one could write a new TG plugin and test it, this feature had to be deferred because of lack of time. An interesting feature to add to Lycaon would be a new, revamped pluggable TG system

able to work with both UDP and TCP. Its output could be made similar to that of iperf, so the *iperfparser.py* module could be used to analyse both.

6.3.2 Graphical interface

Lycaon's current interface is command-line based, while this is not a problem for the node part, it would be nice to have an integrated environment for the experimenter that provides a graphical interface for both creating experiments and analysing the experiments.

6.3.3 Distributed time synchronisation

Time synchronisation is a well known problem in distributed systems, that has generated dozens of publications about it. Even if time is synchronised through NTP amongst the participant nodes just before the experiment -and the experiment is long enough, when the experiment is finished, all nodes will have a different time. This is because the clocks that are embedded in computers are not terrible precise. The drift is influenced by several factors: machine load, temperature, quality of the clock itself, etc. At the moment each Lycaon node keeps a list with the remote time of the neighbouring nodes learnt from beacons received. It would be very valuable if someone implemented a distributed time synchronisation mechanism for Lycaon.

6.3.4 Support for distributed mechanisms research

Lycaon's extensible architecture allows to plug a new distribution mechanism contained in a plugin. It would be interesting to have some sort of support for logging in each node what packet ids have been received. This would enable researchers to empirically see how a given distribution mechanism performs in the real-world.

6.3.5 Plugin System

Lycaon's plugin system is implemented as a small layer on top of Twisted's plugin system. This has served well for the purpose of the project, but there is room for improvement. For example, the algorithmic complexity of getPlugins is $O(N)$, this is not necessarily a problem as the number of plugins in the plugin folder will usually be below 20-30. Nonetheless this could be improved by implementing a $O(1)$ interface to retrieve the plugins, this would mean however getting rid of Twisted's plugin system and revamping Lycaon's plugin system. In addition, Lycaon is currently only extensible in some sections of the system, an interesting future work would be to identify sections that could be interesting to make extensible, for example location information.

6.3.6 Minor Improvements

- Sorting and merging scripts for the traffic logs: Currently traffic logs are uploaded to the FTP server specified in the experiment. This logs are not merged and when analysed only yield the results of one particular node, instead of the set of all the traffic generated by the participant nodes in the experiment.

- Experiment synchronisation: The algorithm that controls the experiment synchronisation does not works if there are no actions specified in the choreography. This is because to signal the start of the experiment all the participant nodes must signal an IAM_OK packet, when all are ready the experiment starts. If there are no actions specified then the nodes have no way to know if everybody is ready to start. A new algorithm must be devised to tackle this problem. A choreographyless experiment is useful in cases where we just want to monitor the packet distribution during an experiment and nothing more.

Appendix A

Example APE choreography

```
# generic setup/teardown instructions
choreography.scenario.title=Relay Swap (TCP)
choreography.total.nodes=4
choreography.startup.command.0=startup
choreography.startup.command.1=tcpdump -i $IFNAME -s 200 -w /var/log
   /tcpdump.apelog &
choreography.shutdown.command.0=killproc iperf
choreography.shutdown.command.1=killproc tcpdump
choreography.shutdown.command.2=copy_files
choreography.shutdown.command.3=pack_files
# node 0
node.0.ip=192.168.5.40
node.0.ipmask=255.255.255.0
```

```
node.0.action.0.msg=This is destination node. Place at position A.
node.0.action.0.duration=30
node.0.action.0.command=my_iperf -s -f m
node.0.action.1.msg=Starting test and spyd logging!
node.0.action.1.command=start_spyd
node.0.action.1.duration=0
# above we wait 1 second less than other in order to start iperf
    earlier.
node.0.action.2.msg=Stay here whole test and wait for requests...
node.0.action.2.duration=66
node.0.action.3.msg=Test soon finished.
node.0.action.3.duration=10
node.0.action.4.command=exit
# node 1
node.1.ip=192.168.5.41
node.1.ipmask=255.255.255.0
node.1.action.0.msg=This is a relay node. Go to position B.
node.1.action.0.duration=30
node.1.action.1.msg=Starting test and spyd logging!
node.1.action.1.command=start_spyd
node.1.action.1.duration=1
node.1.action.2.msg=STAY here for 20 seconds (before moving)
node.1.action.2.duration=20
```

```
node.1.action.3.msg=Now, perform the SWAP (go to position C).
node.1.action.3.duration=25
node.1.action.4.msg=STAY here for the rest of the test.
node.1.action.4.duration=20
node.1.action.5.msg=Just 10 seconds left!
node.1.action.5.duration=10
node.1.action.6.command=exit
# node 2
node.2.ip=192.168.5.42
node.2.ipmask=255.255.255.0
node.2.action.0.msg=This is a relay node. Go to postion C.
node.2.action.0.duration=30
node.2.action.1.msg=Starting test and spyd logging!
node.2.action.1.command=start_spyd
node.2.action.1.duration=1
node.2.action.2.msg=STAY here for 20 seconds (before moving)
node.2.action.2.duration=20
node.2.action.3.msg=Now, perform the SWAP (go to position B).
node.2.action.3.duration=25
node.2.action.4.msg=STAY here for the rest of the test.
node.2.action.4.duration=20
node.2.action.5.msg=Just 10 seconds left!
node.2.action.5.duration=10
```

```
node.2.action.6.command=exit
# node 3
node.3.ip=192.168.5.43
node.3.ipmask=255.255.255.0
node.3.action.0.msg=This is an end node. Place at position D.
node.3.action.0.duration=30
node.3.action.1.msg=Starting test and spyd logging!
node.3.action.1.command=start_spyd
node.3.action.1.duration=1
node.3.action.2.msg=Stay here whole test. Now sending data to node
   0.
node.3.action.2.command=my_iperf -c 0 -t 65 -REPEAT 1 -SLEEP 0
node.3.action.2.duration=65
node.3.action.3.msg=Test soon finished.
node.3.action.3.duration=10
node.3.action.4.command=exit
```

Listing A.1: *Example APE choreography*

Appendix B

Lycaon's BlindBroadcast Plugin

```
# twisted imports
from twisted.plugin import IPlugin
# zope imports
from zope.interface import implements
# python imports
from time import time
# lycaon imports
from lycaon import interfaces
class BlindBroadcastPlugin(object):
    implements(IPlugin, interfaces.IDistributionPlugin)
    _name = 'blindbroadcast'
    _version = '$Id: blindbroadcast.py 49 2006-08-25 13:30:13Z huno
   $'.split()[2]
```

```
    def process_packet(self, data, (host, port), ref):
        pkt_id = data.split(':')[0]
        if ref.closed.has_key(pkt_id):
            # ignore seen packets
            return 0
        if ref._fire_callback(pkt_id, data):
            # Callback fired, log time reception
            # in the close list
            ref.closed[pkt_id] = repr(time())
        else:
            # add it to the close list
            ref.closed[pkt_id] = repr(time())
            ref.broadcast_data(data)
        return 1
# this line is for the plugin system, don't touch
broad = BlindBroadcastPlugin()
```

Listing B.1: *BlindBroadcast plugin*

Appendix C

Experiment file used in evaluation

```
from lycaon.experiment import Experiment

experiment = Experiment()

experiment.metadata =

"""

[info]

id: evalexper

author: Pablo Marti

version: 0.1

[dependencies]

distmech: gossip3-62

storage: sqlitedb-41

[reportback]

host: 192.168.0.2
```

```
user: huno
passwd: lycaon
[logging]
tcpdump: yes
"""
experiment.choreography =
"""
005 192.168.0.1 execute iperf:-u -c 192.168.0.2 -t 10 -i 1 -b 7m
015 192.168.0.2 execute iperf:-u -c 192.168.0.1 -t 10 -i 1 -b 7m
015 192.168.0.4 execute iperf:-u -c 192.168.0.5 -t 10 -i 1 -b 7m
015 192.168.0.5 execute iperf:-u -c 192.168.0.4 -t 10 -i 1 -b 7m
017 192.168.0.1 execute iperf:-u -c 192.168.0.4 -t 20 -i 1 -b 7m -N
040 192.168.0.2 execute iperf:-u -c 192.168.0.1 -t 15 -i 1 -b 7m -N
046 192.168.0.4 execute iperf:-u -c 192.168.0.5 -t 20 -i 1 -b 7m -N
066 192.168.0.1 tg-tcp 192.168.0.2
066 192.168.0.5 tg-tcp 192.168.0.4
"""
```

Listing C.1: *Experiment file used in evaluation*

Bibliography

[1] Orbit project webpage. `http://www.orbit-lab.org`, Sep 2006.

[2] T. R. Andel and A. Yasinsac. On the credibility of manet simulations. *IEEE Computer*, pages 48–54, July 2006.

[3] P. Barron, S. Weber, S. Clarke, and V. Cahill. Experiences deploying an ad-hoc network in an urban environment. In *IEEE ICPS Workshop on Multi-hop Ad hoc Networks: from theory to reality*, pages 103–110, Santorini, Greece, July 2005.

[4] D. Cavin, Y. Sasson, and A. Schiper. On the accuracy of manet simulators. In *Proceedings of the Workshop on Principles of Mobile Computing (POMC'02)*, pages 38–43, Toulous, France, Oct. 2002.

[5] N. G. Duffield, W. A. Massey, and W. Whitt. A nonstationary offered-load model for packet networks. *Telecommunication Systems*, 16(3-4):271–296, 2001.

[6] S. D. et al. *Python Success Stories, Volume I: Eight True Tales of Flexibility, Speed, and Improved Productivity.* O'Reilly, May 2003.

[7] S. D. et al. *Python Success Stories, Volume II: 12 More True Tales*. O'Reilly, May 2005.

[8] S. Ferg. Python & java: a side-by-side comparison. `http://www.ferg.org/projects/python_java_side-by-side.html`, Feb 2004.

[9] A. Fettig. *Twisted Network Programming Essentials*. O'Reilly, Oct 2005.

[10] A. A. Hanbali, E. Altman, and P. Nain. A survey of tcp over mobile ad hoc networks. *IEEE Communications Surveys and Tutorials*, 7(3):22–36, Aug 2005.

[11] M. Hibler, L. Stoller, J. Lepreau, R. Ricci, and C. Barb. Fast, scalable disk imaging with frisbee. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 283–296, San Antonio, TX, USA, June 2003.

[12] P. Jacquet, P. Mhlethaler, T. Clausen, A. Laouiti, A. Qayyum, and L. Viennot. Optimized link state routing protocol. In *Proceedings of 5th IEEE National Multi-Topic Conference (INMIC'01)*, pages 62–68, Lahore, Pakistan, Dec. 2001.

[13] D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353, pages 153–181. Kluwer Academic Publishers, 1996.

[14] A. Kamerman. Coexistence between bluetooth and ieee 802.11 cck solutions to avoid mutual interference. Lucent Technologies Bell Laboratories, July 2000.

[15] W. Kiess, S. Zalewski, A. Tarp, and M. Mauve. Thoughts on mobile ad-hoc

network testbeds. In *IEEE Workshop on Multi-hop Ad hoc Networks: from theory to reality (REALMAN'05)*, pages 93–100, Santorini, Greece, July 2005.

[16] D. Kotz, C. Newport, R. S. Gray, J. L. andYougu Yuan, and C. Elliott. Experimental evaluation of wireless simulation assumptions. In *Proceedings of the ACM/IEEE International Symposium on Modeling,Analysis and Simulation of Wireless and Mobile Systems (MSWiM)*, pages 78–82, Venice, Italy, Oct. 2004.

[17] S. Kurkowski, T. Camp, and M. Colagrosso. Manet simulation studies: the incredibles. *SIGMOBILE Mobile Computing Communications Review*, 9(4):50–61, October 2005.

[18] H. Lundgren, E. Nordström, and C. Tschudin. Coping with communication gray zones in ieee 802.11b based ad hoc networks. In *Proceedings of the 5th ACM international workshop on Wireless mobile multimedia (WOWMOM '02)*, pages 49–55, Atlanta, Georgia, USA, Sept. 2002.

[19] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[20] C. S. R. Murthy and B. S. Manoj. *Ad Hoc Wireless Networks: Architectures and Protocols*. Prentice Hall PTR, NJ, USA, 2004.

[21] V. D. Park and M. S. Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *IEEE Conference on Computer Communications (INFOCOM'97)*, pages 1405–1413, Kobe, Japan, Apr. 1997.

[22] C. E. Perkins and E. M. Royer. Ad hoc on-demand distance vector routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, New Orleans, LA, USA, Feb. 1999.

[23] K. Ramachandran, E. Belding-Royer, and K. Almerot. Damon: A distributed architecture for monitoring multi-hop mobile networks. In *Proceedings of IEEE International Conference on Sensor and Ad hoc Communications and Networks (SECON2004)*, pages 601–609, Santa Clara, CA, USA, Oct. 2004.

[24] 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 *Proceedings of Wireless Communications and Networking Conference (WCNC2005)*, pages 1664– 1669, New Orleans, LA, USA, Mar. 2005.

[25] R. Srinivasan. RFC 1832: XDR: External data representation standard, Aug. 1995.

[26] C. K. Toh. *Ad Hoc Wireless Networks: Protocols and Systems*. Prentice Hall PTR, NJ, USA, 2001.

[27] C. Tschudin, R. Gold, O. Rensfelt, and O. Wibling. Lunar: Lightweight underlay network ad-hoc routing. In *Proceedings Next Generation Teletraffic and Wired/Wireless Advanced Networking (NEW2AN'04)*, pages 38–43, St. Petersburg, Russia, Feb. 2004.

[28] C. Tschudin, P. Gunningberg, H. Lundgren, and E. Nordström. Lessons from experimental manet research. *Elsevier Ad Hoc Networks Journal*, pages 221–233, Mar. 2005.