Routing for Logical Ring Topology Networks
A comparison of two methods

By Alexandros Giagkos

A dissertation submitted in partial fulfilment of the requirements for the degree of Master of Science in Computer Science in the University of Wales.

Supervisor: Dr Myra S. Wilson

Aberystwyth University

31st August 2008

Declarations

This work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed …….. (**Mr Alexandros Giagkos**)

Date…….....

This dissertation is being submitted in partial fulfilment of the requirements for the degree of Master of Science in Computer Science.

Signed …….. (**Mr Alexandros Giagkos**)

Date…….....

This dissertation is the result of my own independent work/investigation, except where otherwise stated. Other sources are acknowledged by explicit references to the bibliography. A bibliography is appended.

Signed …….. (**Mr Alexandros Giagkos**)

Date…….....

I hereby give consent for my dissertation, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organizations.

Signed …….. (**Mr Alexandros Giagkos**)

Date…….....

Acknowledgements

My sincere gratitude goes to my supervisor, Dr Myra S. Wilson, for accepting and supervising this project. Her invaluable help, support, motivation and excellent cooperation played a significant role for the completion of this work.

I would also like to thank my lecturers, and in particular Mr David Ernest Price, for his continuous help during the whole period of the taught part of the M.Sc., by which I gained solid knowledge in the area of networking and the Internet.

Finally many thanks to my parents for their constant support, understanding and total belief in my abilities.

Abstract

This project looks at the design, implementation and comparison of routing protocols for self-healing, bidirectional, logical ring topology networks. The main goal is to propose two simple protocols by exploring two different techniques of searching and routing optimization. The first one uses a set of straight-forward, static rules and is inspired by a number of already existing protocols for both intradomain and interdomain routing, whereas the second one explores the capabilities of a biological inspired searching method; the use of genetic algorithms.

The secondary goal of this work is to compare the two routing protocols by experimenting in an as realistic as possible network environment. For this reason, a widely known network simulator, ns, was studied and used during the implementation and testing phases of the project. The valuable knowledge taken from the experiments and their results lead to a number of interesting outcomes regarding the efficiency, speed and scalability of the two protocols.

Although this project succeeded in creating two prototypes, collecting experimentation data and comparing their results, the protocols could have been improved and built to provide additional capabilities and sophistication. Possible ways to achieve further enhancements and improvements are included in this work, as the author presents his critical evaluation.

Table of Contents

Chapter One

An introduction to the Internet and Network Routing

1.0 Introduction

The work described in this thesis is concerned with network routing protocols. Readers are assumed to be familiar with the ideas of the Internet architecture and/or at least the ISO model as described in [1], [2] and [3] as well as some basic network routing concepts and techniques widely used in the Internet.

1.1 The Internet and the OSI Reference Model

A network is a group of connected, communicating devices such as computers and printers [1]. It is a combination of hardware and software that sends data from one location to another. The hardware consists of the physical equipment that carries electrical signals from one point of the medium to the other, whereas the software is responsible for organizing and orchestrating the various services we expect from a network. When two or more networks are able to communicate with each other, they create an internet. The most notable internet is called the Internet and is the result of thousands of interconnected networks.

The Internet is very often described as a well structured system. A remarkable significance of this structure is that the Internet combines and brings together a number of hardware and even software technologies and architectures that belong to different organizations around the world. In order to achieve compatibility and interoperability between nodes of two different networks, the Internet is carefully designed to use the concept of layers.

In 1947, a multinational body dedicated to worldwide standards was formed and established; The International Standards Organization (ISO). In 1970s, ISO introduced a standard regarding network communication; The Open Systems Interconnection (OSI) model [2, 3]. The OSI model allows us to understand and design new sets of protocols that permit any two different systems to achieve communication regardless of their underlying architectures. All Internet protocols that are designed for such a communication are eventually becoming parts of the layers of the OSI model. As explained in the beginning of this chapter, the detailed description of the OSI model is not in the scope of this work. For a better understanding of the ongoing chapters, the reader is encouraged to look at references

[1] and [2].

1.2 Network Routing

Two of the most important devices regarding IP network[1] communication are the switches and routers[2]. These devices are used to link two or more segments together, allowing signals originally transmitted from an IP network *A* to reach their destinations, even if their destinations belong to an IP network *B*. Routers in particular, provide a number of advanced functionalities that enhance connectivity, compatibility and packet forwarding. One major functionality is called routing, i.e. the methods by which a router knows where the destination of a packet is and how it can be reached in the most fast and optimal way. Routing is achieved by a number of different methods. Each method is very precise and usually requires a number of techniques that allow data collection, data analysis and construction of routing tables. Each method is effectively a set of instructions that is described by a routing protocol. Obviously, routing protocols and their implementations are the most important tools of keeping the Internet in one piece.

Before going into deeper concepts about IP network routing, it is very important to understand what is meant by the term *network topology* [10, 13] and what implications it has in packet forwarding and packet routing. Previously we said that routers can link two or more different networks together. Each network is a group of devices (or nodes from now on) that can talk to each other. Although the way nodes are connected imagines simple and straight-forward to a novice user's eyes, it is not. How the nodes should be connected is often a result of too many decisions that have to do with the physical topology of the equipment, the network connections it requires as well as the expected quality and speed. In some cases, the network topology depends on the market agreements between different companies. There is a number of different network topologies, such as star, fully mesh, ring, bus, straight line, etc[3] as well as their combinations. Although this work focuses on routing techniques dedicated to ring networks only, very soon the reader will discover that it partially uses ideas and characteristics of a fully mesh topology.

1.3 Dynamic Routing and Self-healing Bidirectional Networks

As stated above, routing is effectively a group of operations that include data

1 Here, IP network is used instead of plain network to indicate the use of the Internet Protocol family.
2 Often called 3rd level switches, since their understanding reaches the 3rd layer of the OSI model.
3 For a well explained list of network topologies please refer to [30].

collection, data analysis and creation as well as maintenance of routing tables. The routing tables can be handled dynamically or statically. Again, the question whether dynamic or static routing is the most appropriate cannot be answered with a simple yes or no. It is a question of how big the network is, how often do topological changes occur and what the actual purpose of the network should be.

In respect to the Internet, routing protocols could not be static. Changes happen very often and the routing tables definitely require a large number of human hours of work, in order to be completed and kept up-to-date. Instead, they need to get updated periodically by fast and efficient automatic behaviours. Protocols such as OSPF [35, 36 and 37], BPG [1] and RIP [32] are used for these purposes. Whenever there is a change in the Internet, for instance a breaking of a link or a router's shutdown, the dynamic routing protocols are responsible for updating all of their tables as fast as possible.

Self-healing is another characteristic of a network, very much related to the dynamic nature of these protocols. We say that a network is self-healing when it can withstand a failure in its transmission paths. In more detail, a self-healing network has the mechanism to spot all causes of instability and, then, it can recover automatically [16, 40].

Additionally, bidirectional networks are obviously the networks that permit traffic flow to both directions.

1.4 Routing protocol categories and famous implementations

We have seen that the way routers distribute packets is related to the size of the networks and their physical or logical topologies. It is equally important for the routing protocols (the software) to be able to recognize and understand the structure of the network, as it is for the hardware to understand and make use of the various communication technologies.

Typically, there are two main categories of routing protocols, depending on the way packets are expected to be transmitted. These two categories are the unicast and multicast routing protocols; however this dissertation focuses on the first category only. This restriction applies because both LRTN and LRTN_GA are designed to be unicast protocols.

Today, unicast routing is achieved by a number of different approaches, depending on the purpose and the desirable final result. Thinking for routing solutions between large autonomous systems (interdomain routing), path vector routing proved to be very useful. The principle of path vector routing is that there is at least one node in each autonomous system that acts on behalf of the entire autonomous system. These nodes, the *speaker nodes* as they

are called in the literature, create a routing table and advertise it to the speaker nodes of the neighbouring autonomous systems. It is worth mentioning that this advertisement refers to the path of the routing and not the cost of the routing. An implementation of the path vector routing is the Border Gateway Protocol, abbreviated as BGP. In BGP, the exchange of network information is done by setting up a communication session between bordering autonomous systems. For reliable delivery of information, a TCP-based communication session is set up between bordering autonomous systems using a TCP listening port[4]. This communication session is required to remain unbreakable, for it is used by both sides to periodically exchange and update their knowledge. When this TCP session breaks for some reason, each side is required to stop using information it has learned from the other side. BPG operates by a number of different messages, OPEN, UPDATE, KEEPALIVE and NOTIFICATION and certain timers.

In respect to the intradomain routing, two widely used approaches are discussed. The distance vector and the link state routing. Two very popular implementations of these approaches are the Routing Information Protocol (RIP) and the Open Shortest Path First (OSPF) respectively. Distance vector routing follows an idea where the route with the minimum distance is the least cost route between any two nodes. In this routing method, each node maintains a table of minimum distances to every other node. By using this table, a distance vector routing protocol is able to forward each incoming packet in the most optimal route.

The most popular protocol used in the intradomain routing (inside an autonomous system) is RIP. It is a simple protocol and a direct implementation of the distance vector routing. In RIP, the destination in the routing table is itself a network and is marked down by its network address. Furthermore a classic RIP routing table contains the address of the destination network, the distance's cost expressed in hops and the next node that can be used in order to reach it. Additionally, RIP is the first routing protocol used in the TCP/IP-based network in an intradomain environment [32, 33]. RIP remains one of the most popular routing protocols for small networks[5]. One full RIP's lifecycle includes the following general operations:

- General packet handling
- Initialization – node discovery and initial message broadcasting
- Normal routing updates – done approximately every 30 sec where updates are

4 Typically the port is TCP/179.
5 In fact most of the DSL/cable modem routers are shipped with RIP installed.

broadcasted.

- Normal response receiving – the routing tables are updated by doing a distributed Bellman-Ford step [34, 2]
- Trigger updates – when the metric for a network changes, an update message is generated automatically, containing only the affected networks
- Route expiration – when a network has not been updated for 3 minutes, its metric is set to infinity and it is a candidate for deletion

On the other side of the spectrum, link state routing has its own popular instance; the OSPF protocol [35, 36, and 37] based on a hop-by-hop communication of routing information and specifically designed for intradomain IP network routing. OSPF provides many features such as flooding and List State Advertisements. However the major feature of OSPF is that is provides the functionality to divide an intradomain network into sub domains, commonly referred to as areas. OSPF packets are encapsulated in IP datagrams and they contain their own acknowledgement mechanism for flow and error control. Neither TCP nor UDP (nor other transport layer protocol) is needed.

1.5 Aims and objectives of the thesis

The first objective of this MSc major project and dissertation is to propose two simple new IP routing protocols, that can create and maintain self-healing, bidirectional, logical ring topology networks of N number of nodes. These new protocols are making use of different concepts of computing, not only to provide stable and optimal routing results, but also to illustrate how various techniques may serve differently in respect to the network scalability.

The first protocol uses a straight-forward set of rules and a number of simple algorithms particularly for searching data and building the routing tables. On the other hand, the second protocol uses a more complicated searching strategy, the genetic algorithms [24].

Both Logical Ring Topology Network protocol, or simply LRTN, and its genetic algorithms version, LRTN_GA, are designed to sit on the network layer of the OSI reference model and provide the routing infrastructure for any transport layer protocol (such as TCP, UDP etc). As the reader will discover, LRTN and LRTN_GA packets are encapsulated into Internet Protocol datagrams in order to achieve their transmission, and they do not mean to carry any useful payload to any application. Instead, they can be used to collect the knowledge that a router needs to possess, for being able to know where to forward each incoming packet.

The second objective of this work is to compare the two protocols. The comparison needs to be driven by the results of a list of experiments made on both software packages. The experiments are explained in detail and presented in chapter later in this document. A comprehensive discussion that aims to explore and understand the capabilities of each protocol is also given. Furthermore a discussion of various applications that might make use of this work is also included as well as a critical evaluation and recommendations for future improvements.

1.6 Overall methodology

In order to make meaningful and also useful recommendations on the present routing protocols as well as to propose, design and implement new approaches, it is very important to fully understand the ideas behind the most popular routing techniques. Coupled with this, it is absolutely necessary to have a complete and detailed picture of how the Internet works and what kind of expectations the users have. This knowledge was acquired from various sources including; books, papers, articles, RFCs, MSc and PhD theses as well as related web sites and work of other individuals and experts of the area. A complete bibliography can be found at the end of the dissertation.

Although this project is looking at some of the core concepts of networking (such as searching for optimal paths), software development covers a large part of the project's overall picture. After all, routing protocols are distributed application systems that need to be fast and efficient. Bad memory management and increased software complexity for instance, are reported to be usual causes of badly written protocols.

Additionally, testing and debugging such a system is not a trivial operation. In order for someone to be absolutely sure that both the design and the implementation of a new protocol are faultless and work properly, a platform that can simulate networking operations has be used. Very soon the need of recruiting such a platform was becoming a reality. Therefore part of the background reading was dedicated in searching and researching about a suitable network simulator. A choice had to be made, based on a number of criteria like the simulator's modularity, complexity and ability to simulate as realistically as possible. Chapter two describes in more detail the network simulator and all the required steps taken in order to extend its code, and attach LRTN and LRTN_GA.

After understanding a) the Internet and the existing protocols, b) the network simulator's architecture and philosophy, and c) the objectives of this work, the road to the beginning of the design and implementation of the proposed routing protocols was opened.

The next step was to separate the work load in parts using milestones and to prepare a work plan of the project. The first milestone included the familiarization process with the network simulator as well as its two programming languages[6]. It also required the designing decisions of LRTN coupled with its specifications and implementation. The second milestone was dedicated to LRTN_GA. When the first two milestones were reached the third one took place, during which appropriate data sets for the experiments were chosen. The third milestone can also be considered as the experimentation phase of the project, i.e. the phase in which the actual experiments took place. The final milestone was reached by the end of the write up of this document.

From a software development point of view, the method that was used during the completion of the project is probably better described as a combination of the spiral model and the use of extreme programming. The steps that were required for the complete picture of the project's work plan involve researching, designing, coding and testing. Similar to the spiral model, these steps were repeated in circles, allowing the overall progress to upgrade in levels. In each new level of progress, the feedback of the previous level was used in order to make small changes in the running system.

Additionally, the use of extreme programming allowed the code to evolve effortlessly, by adding new features and keeping the characteristic of modularity safe. The success of this became clear when LRTN_GA adopted techniques and complete pieces of code from the first protocol, requiring only minimal changes to be done.

1.7 Summary of the project's achievements

As the reader will discover later in this project, the author successfully achieved to design and implement two network protocols that aim to provide routing solutions for ring topology networks. A comparison of the protocols is also achieved by a number of experiments, allowing the author to explore the abilities as well as the limitations of the two routing techniques.

1.8 Structure of the thesis

This dissertation consists of six chapters. In the first one, the reader will find an introduction to the main areas of computing and networking that will be used for the rest of

6 Although the author had an experience with C and Tcl programming languages, in order to extend ns-2 code someone needs good knowledge of C++ and OTcl. Chapter two includes further details.

the work, as well as the project's complete description. Chapter two is the part dedicated to the network simulator, which is used as the development and testing platform of the coding part of the project, its architecture and fundamental concepts. Furthermore chapters three and four present the design and implementation decisions for creating the two proposed protocols.

Both testing and experimenting phases are discussed in chapter five where the author makes a comparison of the two. What is more, chapter six contains the author's critical evaluation of the project and a discussion about future work and suggested ways of improvement. Finally, in the appendices of this document the reader will find useful attachments such as a sample simulation’s script, screen shots of the network simulator, header source code of the two protocols as well as raw data taken from sample experiments.

Chapter Two

The *ns* network simulator

2.0 Introduction

As already described in the previous chapter, designing and implementing new routing protocols is not a trivial operation. Similarly to the development of all distributed systems, a routing protocol requires a reasonable number of different systems to run simultaneously. These systems also need to have a number of different communication links between them.

Choosing which network simulator to use for this project was not an easy job, either. Characteristics that affect such a choice are the modularity of the platform (e.g. how easily can the developer add new features or extend the existing ones?), the accuracy of the platform (e.g. are the results going to be accurate enough in order to provide the ground for further recommendations?), etc. This chapter focuses only on the ns network simulator, and provides an overview of some fundamental concepts that someone needs to know beforehand, in order to be able to use it.

2.1 Reasons for choosing the ns network simulator

There is a number of reasons why ns network simulator is chosen as the simulation platform for this project instead of others. Ns is itself a free (to download) software and an open source project. People are able to download ns source code and run it on all popular operating systems, such as UNIX, GNU/Linux, Mac and Windows.

Being an open source project, ns has its own community. This is a very important aspect for any software, which provides confidence and additional help if it is required. Ns in particular has a number of running mailing lists, for both users and developers, and a list of well-written documentation.

Another reason for choosing ns is because it is being widely used in several Computer Science and Computing Engineering departments as well as technology laboratories and parks around the globe. Examples of departments that teach ns are the University of California, Los Angeles (UCLA)[7], the Canadian Simon Fraser University (SFU)[8], the Sophia-Antipolis[9] park

7 http://www.ucla.edu, last accessed 18/07/08

in France and the Center of Neural Science at New York University[10]. Along with teaching, ns is being used in research. For a list of research achievements please refer to section 2.3.

From a software development point of view, ns and in particular ns-2 has a neat and modular architecture, which again provides confidence and productivity from the very first time of use. As the reader will discover in the following sections, ns is written in C++ and OTcl. Although the author had no previous experience in these programming languages, the fact that they both are object-oriented, was another positive aspect of the particular network simulator.

2.2 Overview of *ns*

Ns is a packet-level simulator [14, 15, 26 and 27]. It is a free object-oriented software package initially appeared as REAL[11] in 1987. Nowadays it is part of the VINT project, collaboration between researchers at UC Berkeley, LBL, USC/ISI and Xerox PARC[12]. It is classified as a centric discrete event scheduler that schedules events against time. A centric scheduler like the core of ns cannot accurately emulate events that happen at the same time. Instead, events are handled one by one. However, for the purpose of simple protocols such as LRTN and LRTN_GA, events are often transitory.

Beyond the event scheduler, ns version 2 (ns-2) implements a number of network components and protocols such as:

- routing techniques for distance vector, link state and multicast
- transport layer protocols TCP, UDP, RTP and SCTP
- application layer protocols HTTP, FTP, Telnet, CBR
- and mobile IP as well as ad hoc routing strategies (AODV, DSDV)

Furthermore, ns project includes a number of other useful tools for logging, tracing and manipulating the results of any simulation. For instance, part of the ns package is the nam program (the Network Animator) which analyses the trace files and visualizes ns outputs. Ns-2 also allows changes for getting into work with UNIX and GNU external commands such as awk or Perl.

---

8 http://www.sfu.ca, last accessed 18/07/08
9 http://www.sophia-antipolis.org, last accessed 20/07/08
10 http://www.cns.nyu.edu., last accessed 20/08/08
11 http://www.cs.cornell.edu/skeshav/real/overview.html, last accessed 12/06/08
12 For further information please visit http://www.isi.edu/nsnam/ns/

It is written in two programming languages, C++ and OTcl – an object oriented version of Tcl. Taking advantage of the power and flexibility of these programming languages, ns developing process is based on the fact that all important parts of a network protocol are written in C++, and all the simulations are done by describing their scenarios with OTcl scripts.

C++ is a very precise system programming language that can manipulate bytes and packet headers. It can also be used to implement complex algorithms for large sets of data. Although software written in C++ requires more time to get recompiled and run, it provides better run-time speed, which is the most important for network protocols. On the other hand, Tcl and eventually OTcl are scripting languages, originally designed to provide agility to other programmable and configurable systems. For instance, in case of a simulation where iteration time[13] is important, OTcl is used to set parameters' values without recompiling the source code of the protocol.

Regarding the issues of compatibility and reusability, ns-2 supports a variety of platforms such as FreeBSD, GNU/Linux, Solaris, Mac and Windows.

Ns-2 has three major changes from ns-1. Firstly, the most complex objects of ns-1 have been decomposed into simpler components. Secondly, the configuration interface has changed to OTcl, and thirdly, the interface code to the OTcl interpreter is separated from the core body of the simulator. Although ns-3 has been released very recently[14], ns-2 is used for the implementation and testing of this project in order to avoid any potential bugs and major changes.

Please keep in mind that for the rest of this dissertation, when the author mentions the abbreviation ns, or the phrase network simulator, he refers to the second version of the ns project.

2.3 Research using *ns*

Ns network simulator has been one of the golden choices in research. People in both academia and industry use ns in order to develop, improve and test various network-based issues from transport layer technologies such as TCP Vegas congestion control [41] and TCP performance over Wireless Networks [42], to Bimodal Multicast [43] and other transport layer protocols for Internet-compatible satellite-networks [44].

Other research areas that make use of ns are wireless sensor networks, wireless-

13 Changing the system and re-run.
14 People of the ns-developers mailing list agreed to release ns-3 on the 30th of June 2008.

networked mobile robots, Internet traffic modelling as well as multimedia.

2.4 Fundamental concepts

This section groups together some of the most important and essential concepts that a developer should be aware of, in order to add or extend features and functionalities of the ns network simulator. Due to the size constraints of this dissertation, the concepts below could not be described in much detail, however further information can be reached from references [26] and [27] of the bibliography.

2.4.1 Code hierarchy

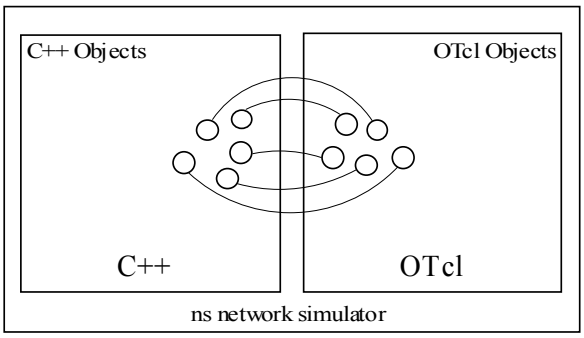


Figure 2.0: Object mapping within ns simulator

There are two kinds of class hierarchies in ns. A C++ hierarchy (sometimes referred to as “compiled hierarchy” in the ns literature) and a similar one within the OTcl interpreter (also called “interpreter hierarchy”). When an instance of a class that belongs to the OTcl interpreter's hierarchy is created, it gets closely mirrored by a corresponding object of the C++ hierarchy. For instance, when users create new simulator objects such as Agents or Nodes[15] via the OTcl interpreter, the appropriate C++ objects are also initiated, providing a one-to-one mapping as shown at figure 2.0. The root of the OTcl hierarchy is the class TclObject and can be found under the tclcl/ directory of ns package. This class defines methods by which user-instantiated OTcl objects are mirrored by objects of C++.

Another key class of the OTcl interpreter hierarchy is the class Tcl. This class includes the methods that C++ invokes in order to access the interpreter itself. In fact, it encapsulates the actual instance of the OTcl interpreter and provides all methods needed to communicate with it. The C++ code:

15 Definitions and details of these classes are given below.

1: *Tcl& tcl = Tcl::instance();*

is used to request a reference to a Tcl instance. Through the Tcl instance the C++ developer can pass values to and from the interpreter. The following section explains how it can be achieved.

2.4.2 Linkage code – OTcl to C++ and vice versa

As shown before, reference to Tcl can be easily obtained from the Tcl class. Once the reference is placed, the developer is able to either pass commands and messages to the OTcl interpreter or get messages from it.

1: *char msg[200];*
2: *strcpy(msg, "Agent/LRTN set pktSize_ 100");*
3: *tcl.eval(msg);*

The example code above illustrates how the command "Agent/LRTN set pktSize_ 100" is passed from the C++ code to the OTcl interpreter by using the method eval (line 3). After executing this code, the OTcl interpreter shall return TCL_OK in case of a success or TCL_ERROR in case of a failure.

Sometimes it is a necessity for C++ code to return result messages to OTcl. When the OTcl interpreter invokes C++ methods, the result of the appropriate procedure (or procedures) is stored in the struct[16] tcl_, and in particular in its private member tcl_->result. Line number 1 of the code below, illustrates the use of resultf(const char*) method, similar to C++ printf.

1: *tcl.resultf("%3.3f secs\n", Scheduler::instance().clock());*
2: *return TCL_OK;*

Yet again, when a C++ procedure invokes an OTcl command, the result message is placed in tcl_->result. In such a case, the return(void) method is used to retrieve the message, as seen in line number 1 of the code below. The message is of type string:

1: *char* rtrn_msg = tcl.result();*

16 Classes in C++ are structs, too.

and has to be converted to the appropriate internal type manually.

Using all these tools that are provided by ns, developers are able to build their protocols, providing at the same time all the hooks between the core source code and the simulation scripts.

2.4.3 Simulation

The heart of a simulation in ns is an OTcl class called Simulator. It provides a set of different interfaces that can be used to configure a simulation and choose the type of event scheduler needed to drive the scenario events. It is common practice to start a simulation script by instantiating and configuring a Simulator object. Once the Simulator is ready, nodes, agents and links between them can be also created, basically constructing a logical topology or, for some experiments, topologies.

The ns simulator is an event-driven simulator. Even packet transmissions are events as defined by the Event class. As we will see later in this chapter, sending a packet to a destination is being achieved by pushing the packet's event into the scheduler's queue. We can immediately understand that the role of the scheduler and the way it handles events is vital. There is a number of different event schedulers available for any simulation each one implementing a different data structure. Examples of schedulers starting from the default one are: the calendar queue, a linked-list, a heap and the real-time. What is more, the unit of all schedulers is the second.

What happens behind the scenes is based in a simple algorithm. The scheduler picks up the event with the next earliest time, executes it and then returns to execute the next one. Furthermore another important think to remember is that given that the simulator is a single-threaded program, only one event is being executed each instance of the time. Hence, the event execution is performed on a first-in first-out manner.

In the appendices the reader will be able to find an example scenario script; however to complete the picture of how simulation works, part of it is explained below.

1: *set ns [new Simulator]*
2: *set node0 [$ns node]*
3: *set node1 [$ns node]*
4: *set agent0 [new Agent/Ping]*
5: *set agent1 [new Agent/Ping]*
6: *$ns attach-agent $node0 $agent0*

7: *$ns attach-agent $node1 $agent1*

8: *$ns duplex-link $node0 $node1 10Mb 10ms DropTail*

9: *$ns at 1.0 "$agent0 start"*

10: *$ns run*

Unsurprisingly, line number 1 instantiates a new Simulator object. Lines 2 and 3 create two different nodes, with their references node0 and node1. Similarly, line numbers 4 and 5 illustrate the creation of two Ping agent objects. Once both nodes and their agents are created, they need to get linked. Lines 6 and 7 attach agents to nodes. Furthermore line number 8 is building the connection link between node0 and node1 which is set up to be a line of 10Mb bandwidth, and a delay factor of 10ms. DropTail[17] is the type of the queue management used in the link. Line number 9 sets the first scheduled event of the simulation. Agent0 is expected to start at 1.0 second of the simulation life-cycle. Finally the *run* command of the final line fires up the simulation.

2.4.4 Nodes and Agents

In the previous section we saw how nodes and agents are created in a simulation. But, how are they represented programmatically and what is the purpose of an agent? These questions will hopefully get answered by this part of the document.

A node is the most important component of a network topology. It is an element, a representation of any network device such as router, computer, printer, etc. Following the object-oriented paradigm, a node is defined in the OTcl class Node. However most of the components of a node are themselves objects, defined in other OTcl classes that extend TclObject. The typical structure of a node consists of an address classifier (classifier_) and a port classifier (dmux_), as shown in figure 2.1. The principle of these two classifiers is to distribute any incoming packet to the correct agent that is attached on their nodes or to any outgoing link.

The concept is not that complicated if someone compares it with the idea of addresses and port numbers in IP networking. Generally, we use port numbers to distinguish between different services that are bound on the same IP address. The same idea applies here, too. Since more than one agent (similar to services in this case) can be attached on a particular node, the dmux_ classifier is used to help distinguishing between them.

17 DropTail implements FIFO scheduling and drop-on-overflow buffer management, which is typical to the most routers of the Internet these days.

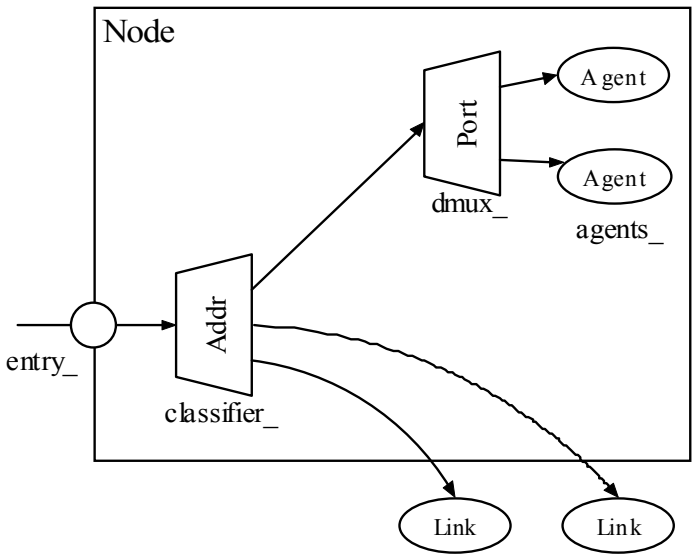


Figure 2.1: The node architecture - Unicast[18]

Other components of a node are the entry_, which represents the node's address, and the agent_, which is a list of all references to the attached agents.

Please note that all the above apply in the case of unicast networking. Multicasting is a bit more complicated and requires different node architecture. Nevertheless, multicast networking was not used for this project and it is out of the scope of this document.

Conversely, an Agent is the software that represents the end-point of a transmission. It is the place where network layer packets are created and/or consumed. Agents are used in the implementation of different protocols and even services of the network, transport and application layers. For instance, the "Agent/Ping" mentioned in section 2.4.3 is the implementation of the application layer protocol ICMP. As expected, the Agent class can be found in both C++ and OTcl hierarchies[19].

Only an object of a class that extends the Agent class can be considered as a simulator's agent. As with all objects, agents have a state and a set of methods. Some of the most important attributes of an agent's state are the addr_ (the source address, usually the node's address), the dst_ (the destination address) and the size_ of the supported packets. Regarding the agent's methods, one of the most important is the allocpkt() which returns a pointer to a fresh, new packet. Obeying the object-oriented paradigm again, there is a small number of methods that are indented to be overridden by those classes that extend Agent.

14 The original of this figure can be found in the NS manual [9].

19 C++ part of Agent class can be found in ns/agent.cc and ns/agent.h whereas OTcl support is in ns/tcl/lib/ns-agent.tcl.

Two of these methods are the command(int, const char*const*) and the recv(Packet*, Handler*). The first one is used as part of the linkage code. Using again the example of section 2.4.3, line number 9 passes the word command *start* to agent0. This word is being injected to the C++ code as a constant array of characters, element of the arguments' array of the command method. The second method, recv, is as straight-forward as it sounds; it gets called whenever the agent receives a new packet and controls the agent's reactions.

2.4.5 Sending and receiving packets – encapsulation and packet headers

Before attempting to describe what the function is when an agent transmits or receives a packet, it is important to say a few words about TCP/IP protocol suite and encapsulation. TCP/IP does not match exactly with the OSI model. It is made of five layers: physical, data link, network, transport and application. Unlike the OSI model, TCP/IP is hierarchical and modular in such a manner that it can contain a number of relatively independent protocols in each of its layers. Depending on the requirements of a system, these protocols can cooperate with each other. Moreover each upper level protocol of the stack is supported by one or more lower level protocols [1].

The last two sentences introduce the concept of encapsulation. By encapsulation, packets of a specific protocol can provide the space to hold data originally created by other protocols. Encapsulation plays a significant role in the ns world, which certainly provides an easy way to use it. Generally speaking, there are two parts that form a data packet; the packet header and the packet payload. The packet header has itself its own structure. Each part of the structure represents either an attribute of the packet or an indication of how it is meant to be handled. The rest of the packet is called payload; the piece of data that is attached to the packet and uses it as the medium of transportation. By encapsulation, packet information of an upper layer protocol becomes the payload of a lower layer one.

The ns network simulator allows the creation of new packet types and header. A header or "the bag of bits" as it is called in the simulator's literature, is programmatically represented by a C++ struct. Good practice and typical ns coding style expect that the names of all header structs have the prefix hdr_. Furthermore it is mandatory for a header struct to include an offset integer that represents the beginning of the header, in addition to an inline static function access(const Packet*). The access functions are called by low level procedures to retrieve the beginning of each header in case of encapsulation. The default headers for each

packet are the common header (hdr_common) and the IP header (hdr_ip)[20].

Both LRTN and LRTN_GA use their own headers to accommodate information about packets and packet transmissions, such as source address, destination address, round-trip time, etc. Although they are network layer protocols, they do not pass their packets directly to the link layer as we would expect. Instead, all packets are first encapsulated inside IP datagrams before going to the lower layers. A complete presentation of LRTN and LRTN_GA headers is given in the appropriate chapters, where the design of each protocol is discussed.

2.4.6 Queues

The concept of queues is also important. Queues in the network simulator represent the places where packets are held before and sometimes after their processing. They are implementations of different data structures and rules that clarify which is the next packet to handle as well as how packets are dropped after their work is done. At the moment, ns already supports a number of queues. The default one that is also being used for this project is the drop-tail[21], an implementation of a first-in-first-out packet management. However round-robin, fair queuing or priority queuing are also available.

Another aspect of queues and packet management is that they allow the use of a delay time for each packet transmission. This delay time is configurable in the simulator. If we go back to the line number 8 of section 2.4.3, we see that the link between node0 and node1 is using a DropTail queue management with 10ms delay. In a few words, this is achieved by blocking the downstream queue of the source node while a packet is in transit and there is at least one other packet to send.

Yet again, developers are able to implement their own queues or modify the existing ones. The core class of queues in ns is the C++ Queue class.

2.4.7 Timers

As described earlier, ns is an event-driven simulator. Depending on the needs of the system, some procedures have to happen in order. In some other cases they also need to be repeated. All these procedures need to be represented by events that are sent to the event scheduler for execution. Timers and TimerHandlers are used for these purposes; to sort out

20 By default, ns-2 adds all of each known headers to a packet. Typically it is not a good think to do, because it dramatically increases the size of a packet. This can be solved by calling the OTcl procedure “remove-all-packet-headers” at the beginning of the simulation's script.

21 The drop-tail packet management is the default option in any typical router nowadays.

and organize the order by which the events are sent to the scheduler.

Defining a new Timer in ns is a clear and rather simple technique. First, someone needs to create a class that extends TimerHandler and provide the virtual method expire(Event*). Examples of Timers and how they are used in network routing can be found in both LRTN and LRTN_GA protocols.

2.4.8 Changes required for the installation

In this final section of the second chapter, the author lists the steps that need to be taken, in order to successfully install LRTN and LRTN_GA in the ns-2.33 full package.

Both protocols use the same packet type called PT_LRTN. Later versions of ns kept records of declared types in an array data structure, defined in common/packet.h. Unlikely, ns-2.33 expects all packet types to be defined as static constant packet_t variables. Hence, the following line needs to be added in common/packet.h:

```
static const packet_t       PT_LRTN = 61;
```

The number 61 happens to be the last available number of the author's installation and it may vary in other installations. However PT_NTYPE should always be the last one. This is a rule which all ns-2 developers need to respect. In fact, PT_NTYPE is not a packet type itself; instead it is used to keep the total number of known types by the simulator.

Once this is done, the name of the packet type needs to be given. This is done inside the p_info class, defined in the same file. An array of strings (char**) of PT_NTYPE size is used to accommodate the names of the packet types. The next step is to add the following line somewhere at the end of the initName() method's body:

```
name_[PT_LRTN]="LRTN";
```

Clearly, accessing name_[61] gives access to the name of the project's packet type. In order to satisfy the OTcl linkage code, the OTcl hierarchy needs to know about the new protocol. This is achieved by adding "*LRTN*" at the end of the foreach body of the tcl/lib/ns-packet.tcl file. By this, LRTN becomes known protocol to the OTcl interpreter and it can be used exactly as all the others.

The two final steps of the installation are to include the source code of the two protocols in the C++ hierarchy. They can be added under their own dedicated directories, lrtn/

and lrtn_ga/ respectively. In order to have both protocols compiled within the simulator, Makefile has to be also updated by adding the lines below to the OBJ_CC variable:

*lrtn/lrtn_rt.o lrtn/lrtn_core.o *
*lrtn_ga/lrtn_rt.o lrtn_ga/lrtn_core.o *

before:

$(OBJ_STL)

As mentioned at the beginning of this section, these steps assume that a full package of ns-2.33 has already been successfully installed in the system. For further general questions regarding ns-2 installation, configuration and troubleshooting the reader is encouraged to refer to [26].

2.5 Conclusion

Ns network simulator has proved to be a very useful tool during the implementation of this work. Although the author's knowledge of C++ and OTcl was rather poor at the beginning, ns structure and architecture were not that difficult to follow.

Chapter Three

The Logical Ring Topology Network (LRTN) Protocol

3.0 Introduction

Up to this point the reader should have a clear view of some of the most important ideas and techniques that are used in network routing. He (or she) should also be able to discuss about the architecture and the fundamental concepts of the ns network simulator.

The next chapters of the dissertation, starting with the current one, constitute the main body of the project's work. In this chapter the first protocol is being proposed and discussed.

3.1 Aims and objectives of the LRTN protocol

LRTN is a network layer protocol designed to provide routing solutions for logical ring topology networks. A logical ring topology network is a network whose nodes are connected in such a way that a message originally generated by node Ni, can propagate through all other nodes until it returns back to Ni. Then it is said that it has completed a full circle.

Moreover LRTN is designed to provide more complex features such as self-healing and bidirectional packet flow. Although packet forwarding is not one of the protocol's responsibilities, LRTN can be used to help routers discover which neighbour is the next hop for an incoming packet. A neighbour, in the context of LRTN, is any node that is interesting in becoming or is already part of the ring topology. In order to get connected, a prospective member of the ring needs to know at least one of the ring's existing members.

In the unfortunate event of instability, the protocol is able to detect the dead link and reconstruct the ring. Instability is defined as the situation when one or more nodes join or exit the topology. Once it happens, the nodes automatically recover ending up with the most optimized ring solution. In the real world, the cost that defines an optimal ring is calculated by a variety of different factors [12, 20]. We usually expect that the traffic flow must be less than the capacity of each intermediate link, and the delay must be less than the specified maximum delay. However for the purpose of this work, the way cost is given is simply by calculating the round-trip time for a full ring circle.

3.2 List of specifications

A complete specification list of LRTN is given below. A node must be able:

1. to accommodate information of its neighbours,
2. to communicate with all neighbours in a fully mesh topology network,
3. to update all information data about others in case of instability,
4. to share its knowledge of other nodes with its neighbours,
5. to learn about other nodes by the information it retrieves from its neighbours,
6. to collect enough information in order to propose an optimal routing path to others,
7. to compare routing paths until they all decide which one is faster (based on round trip times),
8. to adopt a final routing path,
9. to periodically check for routing path's stability as well as neighbour-to-neighbour communication.

3.3 Neighbours and diagnostics

The first challenge that is generally faced while designing a routing protocol is to decide how the topology participants should discover new members and detect instability. As we said, in order to maintain a routing table, a node needs to collect and analyse routing data. In other words, it has to share its knowledge and adopt the knowledge of others. LRTN mimics the way existing protocols achieve all these, by sending and receiving diagnostic packets periodically. For LRTN, sending and receiving assume that the nodes of the network are connected by such a synthesis, that they create a fully mesh topology network (figure 3.0).

Additionally, as the reader will discover later in this chapter, each node of the topology needs to know what the cost is between itself and its neighbours. The cost is expressed as the round trip time of each pair and is measured by sending and receiving diagnostic packets.

There are two types of diagnostic packets. The idea of which is driven by the ICMP timestamp request and reply. Periodically, each node is scheduled to send LRTN_DIAGN packets to all of its neighbours. Unlike ICMP, the formula that is used to calculate the round trip time between a source and a destination machine assumes that their clocks are

synchronized. This is not a major problem for this prototype version of the protocol, because it is designed to run in a simulator where the same clock is used. Part of the header of an LRTN_DIAGN packet is a sent_time field, in which the timestamp of the beginning of the transmission is saved. The destination node that receives a packet of this type is expected to answer with an LRTN_DIAGN_ACK. This acknowledgement packet is sent back to the source node, forwarding the value of the sent_time field of the LRGN_DIAGN previous packet.

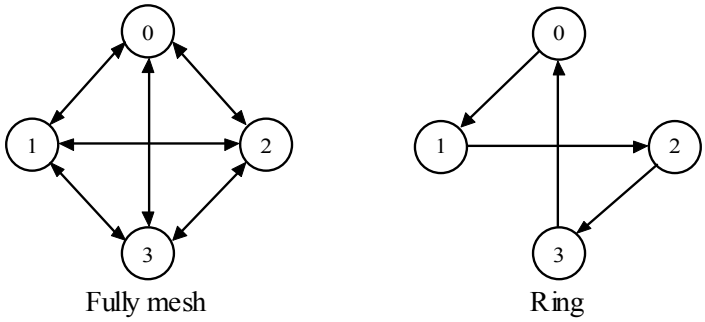


Figure 3.0: Fully mesh and ring topology networks of 4 nodes

Once the LRTN_DIAGN_ACK packet reaches its destination, the first node is able to calculate the round trip time of a packet between a neighbouring node and itself. The same happens for all known neighbouring nodes. Each topology participant is eventually becoming able to construct a routing table containing – at the beginning – the round trip times that it has with all of its neighbours. This routing table is then used by the rest of the LRTN business logic.

However, calculating the round trip times between nodes is not the only purpose of an LRTN_DIAGN packet. It is also designed to carry another valuable set of data aimed to provide self-healing. Along with the sent_time field, the LRTN_DIAGN packet contains space that accommodates the list of all neighbours of the source node. Programmatically, what is sent is an array of nsaddr_t[22] type elements that represents the neighbours' addresses. Before the destination node sends back an acknowledgement of the diagnostic packet, it can update its own list of neighbours against the incoming packet. This technique is inspired by both distance vector routing (such as RIP) and path vector routing (such as BGP). A node updates what it known about the topology by comparing its own knowledge with the one that is given for free by the others.

22 This type is used by ns-2 in place of int32 to represent network addresses.

This is an elegant way of sharing network information and making sure that in the end, all nodes will end up with the same list of neighbours. But what happens if a node goes off-line and how do all nodes get notified of the event? The answer to these rhetorical questions is given by two other techniques which, in LRTN terminology, are called instability discovering and dynamic pruning. A network is by nature an unstable system. Machines might go off-line, communication links might be broken and marketing decisions might affect traffic flow. Similarly to others, in a logical ring topology network instability occurs when a node becomes unreachable. Since every LRTN_DIAGN packet has to be acknowledged, it is reasonably easy to detect whether or not a neighbouring node can be reached. If a node does not reply to a diagnostic packet, it gets pruned from the source node's list of neighbours. As in a chain reaction, once a neighbour is pruned it does not get advertised by the next LRTN_DIAGN packets and in a short time period it will be totally isolated.

Although this technique seems to work in theory, in practice it leads to an unwanted result. As network routing changes and routers go off-line unexpectedly, there is a great possibility that either LRTN_DIAGN or LRGN_DIAGN_ACK can be lost. Then, very unfairly, the corresponding destination node is considered unreachable and gets pruned. Obviously, this is where the drawback appears. To avoid it, LRTN is designed to wait for three diagnostic cycles until it removes a node from its list. Programmatically, this is done by attaching an integer variable to each neighbouring node. This variable represents the number of diagnostic packets that remain unacknowledged. Once this number reaches an internal constant threshold value, a node is free to remove the expired node records. The default threshold is 3 and a diagnostic cycle starts every 150 milliseconds. These values can be changed by editing the lrtn_core.h (version 1) and specially:

```
#define NEIGHBORS_DIAGN_PERIOD 0.15
#define DIAGN_REP_MAX 3
```

The threshold value 3 and the periodic timer value 150 are chosen empirically. Experiments during testing and debugging shown that by using these numbers the diagnostic system is able to detect instability soon enough, without causing useful time to be spent.

3.4 Design decisions and business logic

This section presents the core of the LRTN protocol. It provides detailed information regarding the design and the implementation techniques which are used within the protocol's

business logic.

3.4.1 The LRTNAgent class

The heart of the protocol is the LRTNAgent class and can be found in the file lrtn_core.cc (version 1). This class implements all the rules and the algorithms needed in order to achieve the aims and the objectives. It is also responsible for constructing, transmitting and receiving LRTN packets. Objects of this class are instantiated and attached on each node of the topology.

The OTcl linkage code permits the instantiation of the Agent from the simulation script. The user is able to create an LRTNAgent object by either declaring the agent's network address or not. If the user decides not to give a specific network address to an agent object, it gets a default one (given by ns-2). Additionally, before giving the start command to an LRTNAgent, the user has to give at least one known neighbour's network address. This is done by the OTcl code:

```
1: $agent0 add-neighbor $node1
```

where agent0 becomes familiar with node1. What happens behind the scenes is included inside the body of the mandatory *command* method as shown below:

```
1: [snip]
2: } else if (strcmp(argv[1], "add_neighbor") == 0) {
3:         Node* param_a_ = (Node*) TclObject::lookup(argv[2]);
4:         if (param_a_ == NULL)
5:                 return TCL_ERROR;
6:         neighbors_add(param_a_->address());
7:         return TCL_OK;
8: }
9: [snip]
```

Line 2 introduces the OTcl command add_neighbor that is expected to appear in the simulation script. The following argument of the add_neighbor command is passed as the third element of the array argv. Since the simulator's components are TclObjects in ns-2 and are stored in a hash table data structure, the pointer to the neighbouring node is returned by a

single lookup. Then, the method neighbors_add(nsaddr_t*) gets called in order to add the address of that node to the internal list of neighbours. This list is represented as a map data structure. The reason of choosing a map data structure instead of others, is because a map provides fast lookups and an interface of adding, iterating, and removing elements. The key of the map is defined as the network address of a node (a unique value) whereas the value is itself a C++ struct that contains 1) the network address, 2) the round trip time and 3) the number of unacknowledged LRTN_DIAGN packets.

Once an LRTNAgent knows at least one neighbour, it can start transmitting the beacon messages (LRTN_DIAGN packets). As described in the previous section, this transmission occurs every 150 milliseconds. A timer, called LRTNNeighborDiagnTimer is scheduled to repeat the diagnostic call. At this stage, it is worth to document the concept and use of the JITTER variable. Having in mind that Timers and TimerHandlers are used to organize and sort out the events, it is important to introduce a small random factor that will help Timers avoid collisions in pushing events in the queues. JITTER is in fact a MACRO defined by the line (file lrtn_core.h, version 1):

*#define JITTER (Random::uniform()*0.05)*

So, by adding JITTER to the time that is required by the next Timer to get triggered, the chance of having a collision is minimized. An example of rescheduling a Timer by using JITTER is shown below:

Scheduler::instance().schedule(this, &intr, NEIGHBORS_DIAGN_PERIOD + JITTER);

Via the diagnostic packets, the LRTNAgent is able to collect the data it needs in order to construct and maintain its internal routing table. How the routing table is constructed and represented is discussed in the following section.

3.4.2 The routing table – LRTNRoutingTable class

The routing table is itself one of the most important parts of the protocol and therefore it is represented by its own class. An LRTNRoutingTable object holds the current state of the routing table, which is up-to-date information about the topology that can indicate the next hop of a packet that needs to be forwarded.

In a successfully established ring topology network, each node is connected and can forward packets to only two neighbours. Depending on the direction of the traffic, they can be considered as the node's sender and receiver. Sender is the neighbour that sends packets to the node, and the receiver is the neighbour that receives packets from the node. Two members of the LRTNRoutingTable class, sender_ and receiver_ are used to store the network addresses of the two neighbours respectively. In order to change the direction of the traffic flow, the only thing that needs to be done is to swap the values of these members. Another responsibility of LRTNRoutingTable class is to help LRTNAgent build, store, parse and compare potential routing paths.

A full set of UML diagrams for both LRTN classes can be found in the Appendices at the end of this document.

3.4.3 Potential routing paths and the LRTN_RT_MSG packet type

A potential routing path is a full circle of nodes, a ring topology, which is proposed by a node as the most optimal. Once each node has a potential routing path to propose, the next step is to decide which one is really the most optimal and start using it.

LRTN builds potential routing paths by using a straight-forward technique and algorithm. The idea is that all agents will construct a special message (encapsulated in an LRTN_RT_MSG type of packet) and send it to the neighbour with the smallest round trip time, based on the knowledge they have got from the diagnostics data. The receiver will then sign the message and forward it to its own next hop, again the one with the smallest round trip time. The restriction that applies here is that if a node has already signed the message, it cannot be used as the next hop of another one. This rule makes sure that no loops will happen and the final message will contain signatures of all the participants of the topology. Furthermore once the last (unused) participant of the topology receives the message, it needs to send it back to its original owner. An idea similar to the Internet Protocol's (IP) TTL value is adopted to enhance this rule. Each LRTN_RT_MSG contains a field that indicates the total number of neighbours in the topology, or in other words the total size of the list of neighbours of the originator. The value of this field gets de-incremented each time it reaches a new node.

Programmatically, the message that contains the signatures of all nodes is the payload of a LRTN_RT_MSG type packet. They (the signatures) are represented as a linked-list of a C++ struct type. This struct contains the network address as well as the round trip time between the next hop and the node that is currently signing the message. The reason for which all the round trip times are included, is because in the end they will be summarized to produce

the overall round trip time of the complete routing path.

Once an original owner of an LRTN_RT_MSG packet receives a fully signed message back, it has a potential routing path available. However this is an optimal routing path in respect to the current node. There cannot be any guarantee that it is the most optimal routing path that can be created or all nodes in the topology will end up with the same proposition. This is the reason why this routing path is also called *potential* routing path.

At this point, all LRTNAgents of the topology need to decide which one of them has managed to get the most optimal path. This decision is designed to be taken by comparing the potential routing paths of all the participants. Since a ring topology network has a decentralized synthesis, every time that an LRTNAgent has a potential routing path available, it needs to advertise the path to all known nodes, in a similar way that ICMP router advertisement is achieved. The adopted technique that is used in LRTN for advertising paths and deciding the final one is explained in the next section.

3.4.4 Routing path advertisements and the LRTN_RT_ADV packet type

A new type of LRTN packets, defined as LRTN_RT_ADV, has been introduced for routing path advertisement. The purpose of these packets is to carry full potential routing paths to all nodes of the topology, allowing them to share and compare their proposals and essentially decide which will be globally adopted as the most optimal.

As shown earlier, a potential routing path is a fully signed message, programmatically represented as a linked-list data structure, stored in LRTNRoutingTable objects. The part of the business logic that has to do with the advertisement, consists of a set of straight-forward rules. After receiving a complete LRTN_RT_MSG, the agent constructs an LRTN_RT_ADV packet that carries both the routing path and its total round trip time, and sends it to all of its known neighbours. A neighbour that receives the packet follows the rules outlined below:

- If it has already received its own fully signed LRTN_RT_MSG message back, it compares the two routing paths by their round trip times. The winner (most optimal of the two) is adopted and the other one is being dropped. After this point, the winner is also being advertised, making sure that all other nodes are notified of the change. In case of having both incoming and local routing paths give the same total round trip times, the one originally created by the node with the smallest network address is adopted. The technique is inspired by the way designated parent routers are determined in Reverse Path Forwarding for packet broadcasting [28, 29].

- If its local potential routing path is NULL (the LRTN_RT_MSG is still on its way), it immediately adopts the new one.

After a number of cycles, proportional to the total number of the ring participants, all nodes are expected to end up with the same, most optimal, routing path. Another thing to mention is that every time a new adoption occurs, the LRTNRoutingTable immediately updates its sender_ and receiver_ members.

3.5 LRTN packet header

The LRTN packet header is defined in the C++ header file lrtn_core.h. Depending on the type of the packet and the purpose of the corresponding transmission, some fields of the header may be used or may be left blank. This section presents the four versions of LRTN header, as they are used for each different LRTN packet type.

3.5.1 LRTN_DIAGN and LRTN_DIAGN_ACK

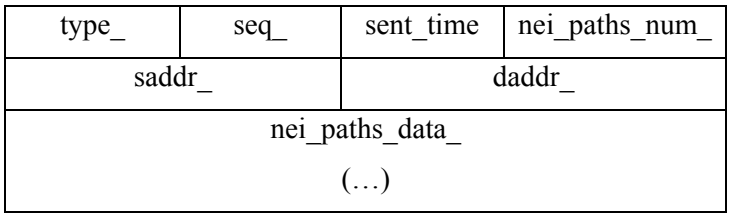


Figure 3.1: LRTN_DIAGN and LRTN_DIAGN_ACK header

- ***type_***: the type of the LRTN packet, values may be LRTN_DIAGN or LRTN_DIAGN_ACK.
- ***seq_***: the sequence number of the packet. This number gets incremented by one for an acknowledgement packet.
- ***saddr_***: the network address of the source.
- ***daddr_***: the network address of the destination.
- ***sent_time***: the timestamp of the packet creation. This is used to calculate the transmission times of a diagnostic packet and its acknowledgement.
- ***nei_paths_num_***: this number represents the number of neighbouring participants in of the next field.
- ***nei_paths_data_***: this is where the list of known neighbours is kept within the packet. The length of this field may vary, depending on the number of the participants.

3.5.2 LRTN_RT_MSG

type_	seq_	rt_msg_rcnt_	rt_rtt_
saddr_		daddr_	
rt_msg_participants_num_			
rt_msg_originator_			
rt_entries_data_ (…)			

Figure 3.2: LRTN_RT_MSG header

r

- ***type_***: the type of the LRTN packet, default value is LRTN_RT_MSG.
- ***seq_***: the sequence number of the packet.
- ***saddr_***: the network address of the source.
- ***daddr_***: the network address of the destination.
- ***rt_msg_originator_***: the network address of the original owner of the routing message.
- ***rt_msg_participants_num_***: the total number of the participants that have already signed the message.
- ***rt_msg_rcnt_***: this is the counter that gets de-incremented in each new hop. Similar to TTL.
- ***rt_rtt_***: this represents the current summation of the round trip time values given by the participants that have already signed the message. When the message is fully signed, this field holds the total round trip time of the potential routing path.
- ***rt_entries_data_***: this is where the list of signatures is kept within the packet. The length of this field may vary, depending on the number of participants.

3.5.3 LRTN_RT_ADV

type_	seq_	rt_msg_participants_num_	rt_prouting_rtt_
saddr_		daddr_	
rt_entries_data_ (…)			

Figure 3.3: LRTN_RT_ADV header

- ***type_***: the type of the LRTN packet, default value is LRTN_RT_ADV.
- ***seq_***: the sequence number of the packet.
- ***saddr_***: the network address of the source.
- ***daddr_***: the network address of the destination:
- ***rt_msg_participants_num_***: the total number of the participants that have already signed the message. This field now holds the total number of participants.
- ***rt_prouting_rtt_***: this field holds the total round trip time of the potential routing path.
- ***rt_entries_data_***: this is where the list of signatures is kept within the packet. The length of this field may vary, depending on the number of participants.

3.6 LRTN Timers

LRTN uses two timers to support its operations. Both of them are periodic timers that either control the sending of packets or start the routing procedures.

3.6.1 LRTNNeighborDiagnTimer

The neighbours' diagnostic timer controls the sending of the diagnostics packets. Although the protocol specifies that this timer must be set to 150 milliseconds, the working model uses a random time between 150 and 200 milliseconds. This is to prevent any possible synchronization and therefore collision in the simulator's schedulers. The range is given by adding a random jiffer value (as described in section 3.4.1). Besides the periodic time as well as the jiffer are both configurable values and can be changed by editing the file lrtn_core.h file.

3.6.2 LRTNRoutingTimer

The routing timer controls the starting of the routing process, in a periodic schedule. The default value is set to 3 seconds. Similarly to the previous timer, this value can be configured from the file lrtn_core.h.

3.7 Conclusion

The LRTN protocol has been tested against a large number of scenarios and has proved to work, without any unfortunate outcomes, providing both speed and efficiency in network routing. As the results of the experiments reveal, nodes that operate using LRTN can create logical ring topologies that can detect instabilities and recover automatically. Moreover implementing this protocol has been a great opportunity for the author to see in practice, how network routing is actually achieved and apply methods and techniques borrowed by existing protocols of the Internet.

Chapter Four

The Genetic Algorithms in Logical Ring Topology Network Protocol

4.0 Introduction

In this chapter, the author proposes and presents the second network routing protocol of this MSc project. The major difference between the two protocols is the way they construct and propose potential routing paths. The new protocol is basically the GA (Genetic Algorithms) version of LRTN (described in chapter three). At the end of this chapter, the reader will hopefully be in a good position to discuss GA concepts as well as how they have been embedded in network routing and in particular in the LRTN_GA protocol.

4.1 Aims and objectives of the LRTN_GA protocol

LRTN_GA is a network layer protocol which can be used to provide network routing for logical ring topology networks. Nodes that implement LRTN_GA are able to communicate with each other in a bidirectional rounded manner. Furthermore they are able to detect network instability and automatically recover, as expected in a dynamically constructed and self-healing network.

Nodes that operate using LRTN_GA are able to join or leave any existing ring topology that is maintained by the protocol. The nodes' communication, instability discovery and routing path adoption is done by sending and receiving specially crafted packets. Likewise LRTN, the nodes are assumed to be able to see each other, just like being connected in a fully mesh combination.

Yet again, packet forwarding is not in the list of the protocol's responsibilities. Instead, LRTN_GA is designed to help routers by indicating the next hop of an incoming packet.

4.2 List of specifications

The specifications of LRTN_GA are listed below. Each node must be able:

- to accommodate information of its neighbours,
- to communicate with all its neighbours (in a fully mesh topology network),

- to update all information data about its neighbours in a case of instability,
- to share its knowledge with others,
- to learn about other nodes by parsing the information it received from its neighbours,
- to collect enough information in order to know the round trip times between pairs of each node of the topology,
- to propose an optimal routing path, using genetic algorithms and genetic operators,
- to share the routing path with all of its neighbours,
- to compare routing paths until they all decide which one is the most optimal (based on the RTT),
- to adopt the final routing path,
- to periodically check for routing path stability as well as neighbour-to-neighbour communication.

4.3 Genetic Algorithms overview

Genetic algorithms are one of the best ways to solve a problem for which little is known. They are search algorithms based on the mechanics of natural selection and natural genetics. Initially developed by John Holland and his team of colleagues and students at the University of Michigan, genetic algorithms have been successfully applied to a variety of search problems. Furthermore GAs have shown that they are efficient at searching large problem spaces and have been successfully used in a number of engineering problem areas, including telecommunications network design [4, 11, 17, 18, 19, 21 and 22].

The sizes of the networks during the last years are growing fast and greedily, and people have already experienced the need of new network solutions, that are able to provide fast and as optimal as possible packet routing between large sets of nodes. This sets the stage for a complex problem with a solution that targets at the optimal topological connections, and routing. As this chapter shows and references [21, 22] indicate, GAs can be used as the searching techniques in order to provide optimal routing results.

As defined at the beginning of this section, Genetic algorithms are biologically inspired, being influenced by theories of evolution. Thus the terminology used in this chapter is also borrowed from Biology science. Before continuing with the basic ideas of GAs, a list of the most important terms is presented:

- ***individual***: a possible solution,
- ***population***: a group of all individuals,

- ***search space***: all possible solutions to the problem,
- ***chromosome***: a blueprint of an individual. Usually a string representation of bits,
- ***trait***: possible aspect of an individual,
- ***allele***: possible settings for a trait,
- ***locus:*** the position of a gene on the chromosome,
- ***genome***: a collection of all chromosomes for an individual

The basic genetic algorithm attempts to evolve traits that are optimal for a given problem. The most common type of GAs works like this: A population is created with a group of individuals that have been constructed randomly. The individuals in the population are then evaluated. The evaluation function is provided by the programmer and gives to the individuals a score based on how well they perform at the given task. The individuals are then selected based on their fitness values. The higher fitness value the higher the change of being selected.

Once the parents (the most efficient individuals) are selected, they reproduce, creating one or more offspring. Reproduction is done by a genetic operator called crossover operator. By crossover, the chromosomes of both parents are combined in order to construct new ones. Another genetic operator may also be used. This operator is called mutation operator and is done by randomly swapping the locus of two genes in a chromosome.

After crossover and/or mutation have been applied, the offspring's fitness value indicates whether it will survive or it will be left behind. Generally, good enough solutions from one population are taken and used to form a new population. This is motivated by the hope, that the new population will be better than the old one. This continues until a suitable solution is found or a certain number of generations have passed, depending on the needs of the programmer.

Programmatically, the outline of the basic GA can be represented as:

1. generate random population of N chromosomes
2. evaluate the fitness f(x) of each chromosome
3. new population steps:
    i. select 2 parents according to their fitness
    ii. crossover with a crossover probability
    iii. mutate with a mutation probability
    iv. place new offspring in a new population
4. replace new population with the old one

5. test, if the stopping condition is satisfied
6. loop to 2

The rest of this chapter is used to present and discuss how learning algorithms, GAs, have been successfully embedded in LRTN_GA protocol. Design decisions as well as genetic operations and their parameters are presented and discussed.

4.4 Neighbours and diagnostics

The technique for discovering neighbours as well as network instability that is used by LRTN_GA, is an exact copy of the technique used in the initially proposed protocol: LRTN. No change has been made in any part of the code, neither for the diagnostics system nor for any of the two packet types. For a complete description and discussion, the reader is kindly asked to refer to the previous chapter, and in particular to section 3.3.

4.5 Design decisions and business logic

This section summarizes and explains the most important decisions made in order to design and implement LRTN_GA, including ns-2 classes and timers as well as a complete description of the genetic operators and their parameters.

4.5.1 The LRTNAgent class

Similarly to LRTN, LRTN_GA has its own LRTNAgent which is responsible for creating, transmitting and parsing LRTN_GA packets. It is also responsible for constructing and maintaining a routing table and its state.

The main difference between this LRTNAgent and the one that is used in the first routing protocol is the way they form potential routing paths. Instead of a) releasing a special message that needs to get signed by all the participants, and b) advertising it as the most optimal, the new agent follows a different strategy. It is designed to collect information about a fully mesh topology, and based on that knowledge to propose what might be the most optimal routing path for a ring network.

In order to collect the information that is needed to form a complete picture of the fully mesh connection, LRTNAgents need to share and combine what they all know about the network. The idea is simple. If an LRTNAgent knows the round trip times between all

possible pairs of nodes in a fully mesh topology, it can logically form the topology and use it for making routing decisions.

An LRTNAgent implements the idea above by using a new LRTNRoutingTable class described below.

4.5.2 The routing table – LRTNRoutingTable class

The LRTNRoutingTable, in the context of LRTN_GA, is the place where the state of the internal routing table is kept. The table is represented by a C++ map data structure and is designed to host all possible pairs of nodes accompanied by their round trip times. Each row of the map has the form:

<std::string, double>

where the key is of type string of the standard C++ library and the value is the double variable for the corresponding time. Upon its construction, all data of this map are copied from the neighbours' list (introduced and described in 3.4 and 3.4.1).

Furthermore the routing table is responsible for keeping the state of the currently used routing path. For this simplified version of LRTN_GA, the state is expressed by two parameters; the routing path itself and its total round trip time. The sender_ and receiver_ members are also present in order to support bidirectional traffic flow (as described in 3.4.2).

Finally, the LRTNRoutingTable is used to help both LRTNAgent and GAs build, and maintain routing paths.

4.5.3 The RTT Table and the LRTN_RTT_TABLE packet type

A complete routing table consists of a full set of possible pairs and their round trip times. To make sure that this table contains the full set of data, each LRTNAgent must construct a special piece of information, called the RTT table or local RTT table, and send a copy of it to all of its neighbours. On the other hand, each LRTNAgent that receives a message like this is expected to parse it and use it in order to fill its local routing table. The routing process (the beginning of the genetic algorithms and their operations) starts when an LRTNAgent has received one RTT table for each member of its local neighbours' list. At that stage, we are assured that the routing table has one record for each possible pair of nodes.

The transportation medium for the local RTT tables is a new LRTN_GA packet type,

called LRTN_RTT_TABLE. Programmatically, this packet is designed to carry two kinds of data. Firstly, a linked-list of type rt_table_entries, a struct that includes the network address of the destination node and the corresponding round trip time. Note that the source network address is not included in the struct, because it can be retrieved from the daddr_ field of the header[23]. Secondly, the packet carries the total size of the linked-list. This number will be used in order to iterate through the data structure.

4.6 Genetic operators in LRTN_GA

This section is used to describe all decisions made for applying GAs in the LRTN_GA protocol. Starting with the encoding scheme of the system, this section attempts to provide all the details that the reader will need to know, in order to understand in depth how LRTN_GA proposes routing solutions.

4.6.1 Encoding

Encoding in GAs is the way genes are represented and used to formulate chromosomes in a population. It is how the gab between symbolic representations and complex high-level behaviours is bridged. Typically, each gene represents a property of the environment and is written down as a number or symbols that response to this property. The simplest representation for genetic algorithms is the one that was used by John Holland: a string of bits [23], for example “10010100”. In such an encoding style, the search space of a considered problem is mapped into a space of binary strings through a coder mapping. Then, after reproducing an offspring, the decoder mapping is applied to bring it back to its real form.

Although Holland's representation is widely used, it was not adopted as the encoding scheme of LRTN_GA. In a problem where the best solution is effectively a routing path that is written as a chromosome, each node can be considered as a unique gene. Furthermore, it is vital for a routing path of a ring structure to include each node only once. Additionally, if we think about the fitness value of a chromosome, we immediately realize that in LRTN_GA case, it depends on the locus, i.e. position of the gene in the chromosome. These three facts lead to the conclusion that the best encoding for LRTN_GA is the permutation encoding. As with permutation numbers, permutation encoding wants all chromosomes to be unique combinations of genes. Another important think to remember is that by this particular

23 The complete header structure is given in section 4.8.

encoding scheme, the search space of the GAs is equal to the permutations of *n* number of nodes. Given by the mathematical formula *n! = (n – 1)! n.*

The following table shows how the permutation space increases for a total number of nodes between 0 to 10:

n	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***
n!	1	1	2	6	24	120	720	5040	40320	362880	3628800

Figure 4.0: Table that illustrates how the permutation space gets bigger, as the network increases

Clearly, as new nodes enter the ring topology the permutation space is rapidly increased to a big number, which, in an optimistic point of view, is quite a challenge for a learning algorithm such as GA. To illustrate the decision's result, a potential chromosome for a network of five nodes is represented in LRTN_GA as “03124”. Each number in the chromosome is the network ID (similar to network address) of the node it represents. A one-to-one mapping, if we remember that ns-2 is automatically assigning and increasing the nodes' IDs.

4.6.2 Generation of the initial population

In the world of GAs, the initial population is the first set of samples of possible chromosomes that will be examined and used for the future generation cycles. Each chromosome in the set represents one possible solution to the problem. The initial population is expected to be a set of randomly selected chromosomes.

However random generation was a bit awkward. After searching in the standard C++ library, the attention was dropped at a method called next_permutation(). Although next_permutation() always returns unique sequences, these sequences arrive in a particular order. What this method does basically, is to rearrange all given elements into a lexicographically next greater permutation of elements. For LRTN_GA this is not a desired option. An example will clearly illustrate the reason:

The first 10 next_permutation() returned values for a given range or numbers, 0 to 19, are:

0 1 3 2 *4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19*
0 2 1 3 *4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19*
0 2 3 1 *4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19*

0 3 1 2 *4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19*

0 3 2 1 *4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19*

1 0 2 3 *4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19*

1 0 3 2 *4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19*

1 2 0 3 *4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19*

1 2 3 0 *4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19*

1 3 0 2 *4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19*

Although these numbers are all unique permutations, they cannot be considered as part of a reasonable population. The crossover and even mutation operator will not produce improved offspring. It is generally accepted that both operators work more effectively if they are applied in chromosomes where genes are randomly mixed up.

A different approach was taken for solving this problem. In order to create random chromosomes to form the initial population, LRTN_GA uses the random_shuffle() C++ method and a combination of data structures. In the protocol's implementation a chromosome is represented by a vector data structure. The reason, for which a vector was selected instead of a simple array, is that vectors in C++ are fast and provide a number of features such as iterators and methods to push_back and remove elements without keeping track of the structure's index[24]. What happens when the create_initial_population() methods is called, is that a vector gets filled with all known neighbours' addresses[25] that are found in the neighbours' list (described in section 3.4.1). Once this is done, the procedure continues by calling random_shuffle() for the vector. This function is not producing any permutations at all; instead, it shuffles the elements of a given array-like data structure. A check to see whether a shuffled result has been produced before ensures that each sequence is used only once. The argument here is that the time it needs to check whether a shuffled sequence has already been created, is dramatically less than the time next_permutation() needs to calculate all possible permutations and then randomly pick a random set.

A map data structure is also used in order to store the chromosome's population. The key of this map is the string representation of the vector, constructed as shown below:

For a vector with elements: 0 2 1 3 4,
Chromosome's string representation: 0|2|1|3|4

On the other hand, the value of each map element is a copy of the vector that contains the

24 However, by using an index, the developer can easily access particular elements of the vector.
25 In form of node IDs.

chromosome (vector of genes). The logic described here is repeated until the number of cycles equals a constant, a configurable variable that is used to indicate the size of the initial population. The default size is 150. Notice that for up to five nodes in a topology, the search space is 120. To avoid infinite loops, when the size of the permutation space is reached, the procedure that creates random chromosomes is also forced to exit.

In GAs, the size of the population should be determined in advance. Although GAs are used for large search spaces, very big population sizes usually do not improve the performance. It is generally accepted that a good population size is about 20-30, and sometimes 50-100 depending on the problem. Furthermore the population size remains constant from one generation to the next. In some situations however, it can be useful to have a population that changes sizes. As the reader will discover later, LRTN_GA is one of these situations.

4.6.3 The fitness function

A fitness value is the metric by which we can tell whether a chromosome is good or bad. Typically fitness is calculated by the fitness function (or evaluation function in some parts of the literature). Clearly, the fitness function plays a very important role in GAs. Depending on the returned value of the function, the probability of a chromosome being selected for the next generation is determined.

The business logic of a fitness function varies, based on the needs of each problem. It can be simple or more sophisticated. For instance, when using genetic algorithms to sort numbers into numeric order, a suitable fitness measure might be to count how many numbers are placed in the correct position. A more sophisticated measure of fitness could be to measure how far from its correct place each incorrectly placed number is.

In LRTN_GA routing protocol, the fitness value is measured by determining the total cost of a routing path. Since the only factor that is used to calculate the cost is the round trip time, the fitness value of a chromosome (possible routing path) is its total round trip time.

Generally, the fitness function should return higher values for better solutions. In our case though, the most optimal solutions is the one with the lowest cost. In order to make sure that the selection process is using the fitness values correctly, the fitness function of the protocol returns a number inversely proportional to the determined cost.

By concrete mathematics [8] we know that two quantities y and x are said to be inversely proportional when:

$$y = \frac{c}{x}, \text{ or also written as } y \propto x^{-1}$$

where c is a constant value. Using the above, the fitness function returns higher values for better chromosomes. A selection method is then used, to determine the probability of choosing each one.

4.6.4 Selection

There is a number of different methods of selection that can be used with genetic algorithms. Depending on the nature of the problem and its complexity, selecting parents can be achieved by rank selection, fitness proportionate selection, tournament selection, steady state selection, and others. For the purposes of this protocol, the fitness proportionate selection, or also known as the Roulette Wheel selection is used.

Before explaining the main reason of using it instead of the others, it is important to discuss a very interesting aspect of network routing and routing paths. As we already said, the number of the possible solutions that reflect to a ring topology network is equal to the permutation space of the total number of nodes that participate in the topology. We also said that the cost of each solution is the sum of the round trip times between all pairs of nodes that are found in the path. Furthermore, the natural characteristic of a ring structure is that there is no leading or last node. Combining these three together, if the round trip time of the route: [1, 3, 4, 0, 2, 1] is 500ms, we expect that the round trip time of the route: [3, 4, 0, 2, 1, 3] is also 500ms. Clearly it is true, because both of these paths are eventually describing the same ring. Likewise, the two chromosomes 1|3|4|0|2 and 3|4|0|2|1 are describing the same result.

The roulette wheel selection is a proportionate selection; the possibility of a chromosome being selected is proportional to its presence on the roulette wheel. As shown in the previous paragraph, LRGN_GA design allows a number of chromosomes to effectively represent the same solution of the problem. For the roulette wheel point of view, each individual still has its own probability of being selected. However, the probability of selecting a possible solution is eventually equal to the sum of the probabilities of selecting each of the chromosomes that represent the solution. The sample pie chart in figure 4.0 (ii) illustrates how it is mapped on the roulette wheel. Chromosomes 1 and 5 represent the same ring solution; however they are different in terms of their genes' structures. Although they both have the same selection probability (18.35%), their solution probability is 36.70%.

Another important characteristic of the roulette wheel is the random factor, the

throwing of the ball. Again, thinking of the roulette pie and how it is being sliced, even the worst chromosome of the population has its own little probability of being selected. This allows mating of parents to be more interesting and to give a flavour of realistic hope to the next generations.

Mathematically speaking, the probability of choosing a chromosome in respect to the roulette wheel selection is given by the formula:

$$P(h_i) = \frac{fitness(h_i)}{\sum_{j=1}^{p} fitness(h_j)}$$

where $P(h_i)$ is the probability of a hypothesis (a particular solution) h_j and p is the size of the population. Hence, the probability of a particular hypothesis to be selected is calculated by its fitness value over the summation of the fitness values of all the chromosomes in the population. The following table illustrates how the selection probabilities are being calculated using the above formula, in an initial population of only three chromosomes.

Chromosome	***Cost***	***Fitness value***	***Probability score***
1	245ms	0.00408	0.217 or 21.7%
2	203ms	0.00492	0.262 or 26.2%
3	102ms	0.00980	0.521 or 52.1%

Table 4.0: Selection probabilities using Roulette Wheel selection

The fastest routing path is given by the third chromosome, with a cost of 102ms. The fitness value for this cost is 0.00980. Using the roulette wheel's formula, the probability of being selected is 52.1%. Furthermore these values are mapped as shown at figure 4.0 (i).

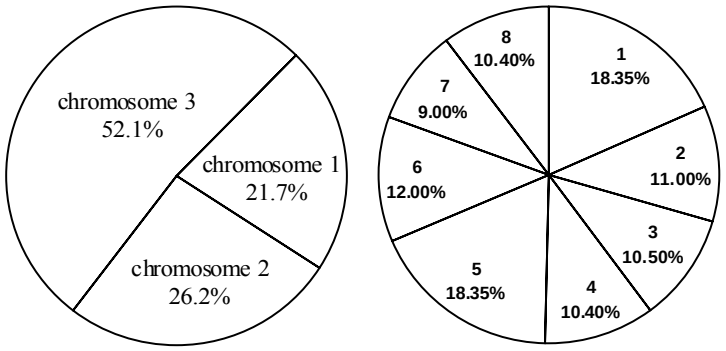


Figure 4.0: Roulette wheel examples

In the code, the roulette wheel is represented by a map data structure. A map in this case allows fast searching and iteration. The key of the map is defined as a double value, which is designed to keep the probability of a chromosome. The probability is now expressed as a percentage with a range of 0.00 to 100.00. The value is, as expected, the vector containing the corresponding chromosome.

In order to emulate the throwing of the ball, each new chromosome's probability is added to the sum of the current probabilities of the chromosomes already in the map. This trick simplified the way selecting index is positioned in the data structure, in order to locate the selected chromosomes. The following piece of code is taken from the LRTNAgent class and illustrates how the first parent is being selected:

```
1: selection_pos1 = (((((double) rand() / (double) (RAND_MAX + 1)) * 100)) * -1) + 1;
2: for (roulette_wheel_iter = roulette_wheel_.begin() ;
3:    roulette_wheel_iter != roulette_wheel_.end(); roulette_wheel_iter++) {
4:          if (selection_pos1 <= roulette_wheel_iter->first) {
5:          parent1 = roulette_wheel_iter->second;
6:          parent1_pos = roulette_wheel_iter->first;
7:          break;
8:          }
9: }
```

Line 1 defines a random double number (range 0 to 100). This number, as shown in line 4, is used to find the result position on the wheel, indicating the corresponding solution.

At this point, the first step creating the next generation is completed. The next step is the step of mating parents, and is discussed in the next section.

4.6.5 Crossover

Typically, the fitness chromosomes are selected in each generation to mate with each other, and each pair of chromosomes is allowed to produce two offspring. Mating is achieved by a genetic operator called crossover.

Depending on the encoding style, the most usual crossover operators are the single point, two points, uniform, and the permutation crossover[26]. By this genetic operator, two chromosomes are combined together in order to create totally new ones (offspring). Although the new chromosomes are expected to be themselves unique, their mixture is expected to be close enough to both of those of their parents.

Back to LRTN_GA, since the encoding evolves real numbers and their permutations, the crossover is being done by a one-point permutation method. Permutation crossover is a bit tricky, in relation to others. Initially, the first part of the new chromosome is taken from the first parent. Then, the second part is filled with all genes of the second parent, which are not present in the new chromosome's first part. An example is given below:

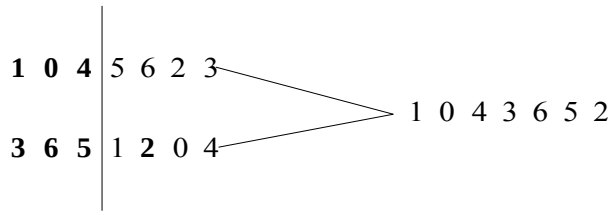


Figure 4.1: The permutation crossover operator

Programmatically, the two parents are selected from the population and are mated with the following algorithm:

1. set crossover point = chromosome.size / 2
2. first part of the parent 1 is copied to the offspring's chromosome
3. the rest of the offspring's slots are filled with the genes from parent, that are not already used

26 Notice that all these crossover techniques require chromosomes of the same length.

The implementation is as follows:

```
1: int size = parent1.size();
2: int pos = size / 2;
3: int flag = FALSE;
4: vector<nsaddr_t> offspring = parent1;
5: int cnt = pos;
6: for (int i = 0; i < size; i++) {
7:         for (int j = 0; j < pos; j++)
8:                 if (parent2[i] == offspring[j]) flag = TRUE;
9:         if (!flag) offspring[cnt++] = parent2[i];
10:        flag = FALSE;
11: }
12: return offspring;
```

The offspring that is returned by line 12 is then evaluated by the fitness function. The fitness value of the new chromosome plays an important role in its future. At this point, LRTNAgent has to decide whether or not the offspring is able to survive to the next generation.

Ideally, in genetic algorithms the population that becomes subject to the next generation process is expected to include the best individuals of the previously used population. To satisfy this statement, a number of optimization rules have been applied to the protocol. These rules are listed below:

1. IF the fitness value of the offspring is less than the fitness values of both of its parents, THEN throw the offspring away and keep both of its parents to the next population.
2. IF the fitness value of the offspring is better than at least one of the fitness values of its parents, THEN keep the offspring and the parent with the best fitness value. Throw away the bad parent.

These two rules are making sure that the offspring is permitted to proceed to the next generation if and only if it looks promising. Crossover takes place in every GA cycle with a crossover probability. Although the default probability value (90%) has been proven to work just fine, the reader and user of the LRTN_GA protocol is more than welcome to make

changes and tune the genetic algorithms as he/she desires.

Another issue, that someone needs to keep in mind when dealing with permutation crossover, is to make sure that the permutation rule is not violated. Chromosomes in the population must be unique. In each GAs cycle, after the crossover operator is finished, the offspring may end up to be an exact copy of an existing member of the new population. Therefore, a checking takes place to make sure that only unique sequences are kept. This has a very interesting result; the working population gets shrunk in each GAs cycle[27]. There are two ways to look at this phenomenon. The first one is to argue that it is a technique of optimizing the algorithm and maximizing its performance. This is partly the truth, especially when the search spaces are small, because only the best chromosomes will be kept in each GAs cycle and eventually survive. On the other hand, if the population size is not kept the same, the number of samples that are examined during the generation processes is quite limited. In large permutation spaces, this may result to a solution that is far away from the best one. But, do GAs give the *best* solution anyway?

4.6.6 Mutation

Mutation is the second genetic operator that is used in genetic algorithms. Similarly to crossover, mutation depends on the encoding style of the chromosomes. In most of the cases, it is a unary operator (i.e., it is applied to just one argument – a single gene) that is usually applied with a low probability, usually 1-2%.

However, in a problem such as LRTN_GA, where permutations are used, the mutation operator is designed a bit differently. The basic idea here is that two randomly selected genes of the chromosome swap their positions. The following algorithm explains how mutation is achieved in LRTN_GA:

1. set mutation point1 = a random number between 1 and chromosome's size
2. set mutation point2 = a random number between 1 and chromosome's size – different from point1
3. define new gene = chromosome[point1]
4. chromosome[point1] = chromosome[point2]
5. chromosome[point2] = generation

27 Such behaviour can be compared with the Hill Climbing searching technique, something that is discussed later in this chapter.

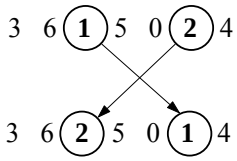


Figure 4.2: The mutation operator for permutations

The default mutation probability is empirically set to 2%; however users can change the value depending on the needs of the system. As for all probability values of the genetic operators, the mutation probability is subject to any optimization and tuning.

4.6.7 The stopping conditions

A very important issue when programming with GAs is to decide when they should stop. There are typically two ways in which a genetic algorithm is terminated. Usually, a limit is put on the number of generations, after which a good enough solution is considered to have been found. In some other problems, the genetic algorithms stop when a particular solution has been reached, or when the solution with the best fitness value in the current population has reached a particular fitness level.

In respect to the LRTN_GA routing protocol, the working (and will become next) population is getting trimmed in every GAs cycle. Therefore a stopping condition is reached when the population size reaches a minimum level. The actual minimum size is set to 2 (for a 1 to 1 node connection).

Another stopping condition is the GA_CYCLES factor. This is the number of iterations the genetic algorithms will have for every routing process. The default value for this factor is set to 20, and is of course configurable to fit different needs. At this point it is worth saying that when LRTN_GA ends up with more than one solutions in the last generation, it proposes the chromosome with the best fitness value. This chromosome is called the winner.

4.7 Routing path advertisements and LRTN_RT_ADV packet type

Once genetic algorithms stop and the LRTNAgent has an acceptable solution of the problem, it is time to share the result with all other neighbours. Obviously, for a large number of nodes in the topology, LRTNAgents are not expected to produce the same routing solutions. Therefore they need to agree on which path is considered as the most optimal and then start the adopting process.

Similarly to the first proposed routing protocol, LRTN, a new packet type is

introduced for carrying the GA winners[28]. The rules of adoption are the same as in LRTN, and are described in chapter 3 section 3.4.4.

4.8 LRTN_GA packet header

LRTN_GA packet header is defined in lrtn_core.h (version 2). The packet types LRTN_DIAGN, LRTN_DIAGN_ACK are omitted since they have been already presented in chapter 3, section 3.5.1.

4.8.1 LRTN_RTT_TABLE

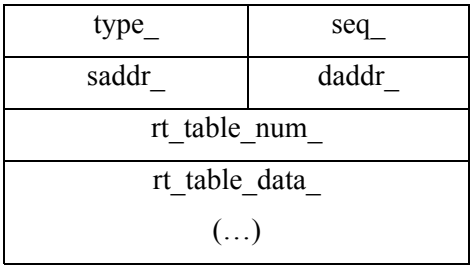


Figure 4.3: LRTN_RTT_TABLE header

- ***type_***: the type of the LRTN_GA packet, default value is LRTN_RTT_TABLE.
- ***seq_***: the sequence number of the packet.
- ***saddr_***: the network address of the source.
- ***daddr_***: the network address of the destination.
- ***rt_table_num_***: this represents the total number of participants in the RTT table.
- ***rt_table_data_***: this is where the list of the pairs is kept for the current RTT table.

4.8.2 LRTN_RT_ADV

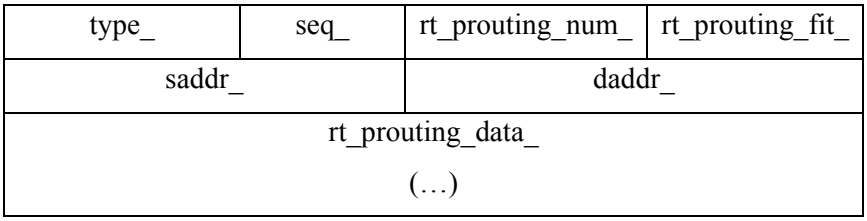


Figure 4.4: LRTN_RT_ADV header

28 GA winners are described in section 4.6.7.

- ***type_***: the type of the LRTN_GA packet, default value is LRTN_RTT_TABLE.
- ***seq_***: the sequence number of the packet.
- ***saddr_***: the network address of the source.
- ***daddr_***: the network address of the destination.
- ***rt_prouting_num_***: the number of participants of the routing path.
- ***rt_prouting_fit_***: the total fitness value of the routing path.
- ***rt_prouting_data_***: this is where the routing path is kept.

4.9 LRTN_GA Timers

The timers that are used in LRTN_GA are similar to the ones used in the LRTN routing protocol. The reader is kindly requested to refer to chapter four, sections 3.6, 3.6.1 and 3.6.2.

4.10 LRTN_GA and the Hill Climbing method

The area of searching by using learning algorithms lists a number of different methods that can be applied in order to find optimal routing paths for networks with large searching spaces [5, 6]. The aim of this section is to discuss and compare the hill climbing method with the genetic algorithms and the way they have been implemented in the LRTN_GA routing protocol. The need of this discussion is driven by the fact that hill climbing method is very close to the searching strategy and optimization rules of this work.

The hill climbing [24] is an example of using information about the search space to search in a reasonable efficient manner. It usually becomes the first choice for a lot of systems, because it is simple to implement. Hill climbing can be used to solve problems that have many solutions, some of which are better than the others. It starts with a random solution and iteratively makes small changes to it, in order to improve it. Furthermore a hill climbing method terminates when it cannot see any new improvement. At that point it stops, it is said that the current solution is close to the most optimal one. One similarity between hill climbing and the way GAs are optimized in LRTN_GA, is that they both come up with a solution that, although it is not guaranteed to be the best one, it tends to be close to perfection.

The generation and testing approach of the hill climbing method is described by a widely known algorithm that is used to climb a mountain in fog with an altimeter but no map. The algorithm is defined as following:

1. for each direction, north, south, east and west, check the height one foot away from your current location
2. as soon as you find a position where the height is higher that your current position, move to that location and restart the algorithm
3. the stopping condition is set to when all the directions lead lower than your current position

The three rules above remind us of the optimization rules of the crossover operator used in this protocol, where an offspring is allowed to survive only if its fitness value is better than at least one of its parents' fitness. The philosophies of these optimization techniques are similar and aim to provide fast and as efficient as possible results.

4.11 Conclusion

Researching, designing and implementing the LRTN_GA protocol has been a very interesting process. Not only it gave to the author the opportunity to go deeper in the area of learning algorithms, and in particular GAs, it also offered the experience of using them within a network application. After a number of successful experiments, LRTN_GA has been proved to be a routing protocol that can propose routing solutions for both small and large topologies. The results of the LRTN_GA are presented in the next chapter, in conjunction with a discussion of the statistical comparison between the two proposed protocols of this dissertation.

Chapter Five

Experiments and their results – A comparison of the two protocols

5.0 Introduction

By reaching this point of the dissertation, the reader should be able to know how both LRTN and LRTN_GA protocols are designed and implemented. The next important stage of documenting this work is to actually describe the experimentation period of the project, and present the results.

Starting by explaining how testing and debugging were achieved in terms of software engineering, this chapter includes a full description of the simulation scenarios used in ns-2, as well as a discussion of the results that have been produced. At the end of this chapter, the reader will find a comparison of the two protocols, reasons and assumptions of whether or not one of them is better than the other.

5.1 Testing and debugging

Currying out tests for the protocols' code against memory leaks or faults of other types was a challenge. As described in chapter two, ns-2 is a simulator engine built in C++ that uses an OTcl (object-oriented Tcl) interpreter for configuring and passing commands to the appropriate components of the simulation.

During coding, the debugging at C++ level was done by the standard debugger, gdb. However, it is a painful reality that when looking at the OTcl code for debugging OTcl procedures, the developer needs to get back at C++ classes, and vice versa. The strategy that was followed to ease the pain, was to take tiny steps (making small changes[29]) each time, and add plenty of indicating printf() functions before re-running the simulation. Debug methods were also used for the same reason, an example of which is the neighbors_print() that can be found in the source code.

Another issue, particularly with LRTN_GA implementation was the memory management and memory debugging. Although the computer that was used during the completion of the coding part of the project had enough memory to cover its needs, a number of simulations ran out of memory because of the default operating system restrictions. This

[29] Something highly related to Extreme Programming.

was solved by using ulimit (in a bash shell) to dedicate enough memory.

The data structures that have been used for both LRTN and LRTN_GA at C++ level provide efficient and flawless memory management. In order to save memory at the OTcl level, all simulation scripts use arrays for storing their nodes and agents. By using an array to keep a sequence of nodes, for instance, only one variable is created; the one that points to the beginning of the array.

Another important issue with memory management in ns-2 is that, by default, the simulator puts all known packet headers in each new packet transmission. To avoid this, all simulations start by disabling this feature. The only acceptable headers are the common, IP and LRTN.

Furthermore it is worth saying that OTcl has no garbage collector and the de-allocation of memory for a whole object is not done automatically. This causes memory leaks to happen unexpectedly and the developer ought to be very careful with object instantiation. For the experimentation phase, only nodes and agents that are used get created, making sure that the memory usage is controlled.

Once a routing protocol was completed, the code was tested against a number of sample scenarios. These scenarios cover a number of wanted and unwanted events, such as adding new nodes in the topology and stopping an agent.

For instance, the table below was used to test LRTN with a topology of 4 nodes, with 10Mbit connections and a random delay factor between 0 and 10 on each link:

#	*Description*	*Timestamp*	*Expected*	*Result*	*Comments*
1	Are N nodes created?		Yes	Yes	
2	Are N agents created?		Yes	Yes	
3	Are N links created?		Yes	Yes	constructing a fully mesh topology
4	Did agent N add agent N in its neighbours' list?		Yes	Yes	neighbors_print() was called
5	Are all agents started?	0.010000s	Yes	Yes	
6	Are N agents started?	0.010000s	Yes	Yes	all 4 agents started
7	Did agent 1 add node 2 in its neighbours' list?	0.010033s	Yes	Yes	instability detected, routing process started
8	Did agent 2 add node 3 in its neighbour's list?	0.038810s	Yes	Yes	instability detected, routing process started
9	Did agent 3 add node 1 in its neighbour's list?	0.015726s	Yes	Yes	instability detected, routing process started

#	*Description*	*Timestamp*	*Expected*	*Result*	*Comments*
10	Did agents 0, 1, 2, 3 send LRTN_DIAGN packets?		Yes	Yes	periodically
11	Did agents 0, 1, 2 and 3 reply with appropriate LRTN_DIAGN_ACK ?		Yes	Yes	periodically
12	Did agent 0, 1, 2 and 3 start the periodic routing process?	3.010000s	Yes	Yes	
13	Did agent 0 receive a full rt_message back?	3.111288s	Yes	Yes	agent 0 received the message and adopted it, potential path 0 2 1 3
14	Did agent 1 receive a full rt_message back?	3.089881s	Yes	Yes	agent 1 received the message and adopted it, potential path 1 3 0 2
15	Did agent 2 receive a full rt_message back?	3.106556s	Yes	Yes	agent 2 received the message and adopted it, potential path 2 1 3 0
16	Did agent 3 receive a full rt_message back?	3.110090s	Yes	Yes	agent 3 received the message and adopted it, potential path 1 3 0 2
17	Did agents with bad solutions adopt the good ones?		Yes	Yes	best routing path is 1 3 0 2 agent 0 adopted at 3.123619s agent 2 adopted at 3.128557s

Table 5.0: Sample test sheet for LRTN testing process

Similarly to this sheet, others have been used to test both protocols, providing useful feedback during the development process.

5.2 Simulation scenarios – The experiments

The rest of this chapter describes the experimentation phase of the project and presents the results. The experiments that took place during this phase are grouped into three categories, which aim to test and evaluate the two protocols against their specifications. The first category includes experiments that target at the diagnostics' system. The second and the third categories aim to provide useful information about the routing ability of the protocols, i.e. they measure their speed and efficiency, in terms of optimal routing solutions.

Number of nodes	*Constant delay*	*Number of repeats*
4	10Mb (0,10)	50
6	10Mb (0,10)	50
8	10Mb (0,10)	50
14	10Mb (0,10)	50
16	10Mb (0,10)	50
18	10Mb (0,10)	50
4	10Mb (100,180)	50
6	10Mb (100,180)	50
8	10Mb (100,180)	50
14	10Mb (100,180)	50
16	10Mb (100,180)	50
18	10Mb (100,180)	50

Table 5.1: Simulation scenarios

The table above summarizes the twelve different scenarios that have been designed and executed during the experimentation phase. In order to test how the two protocols operate within networks of different sizes, the scenarios were designed to run the simulator with 4, 6, and 8 nodes. Networks with 14, 16 and 18 nodes were also designed, for exploring the protocols' capabilities within bigger networks.

The time delay between each link indicates how busy the network is. Thus, the two protocols have been tested in both “empty” and “busy” networks, with random delay factors between 0-10 for empty, and 100-180 for busy. In order to get as accurate as possible results for our discussion, each scenario was repeated for 50 times. That was also important because the connectivity between the nodes is designed to have a random delay factor[30].

For each new simulation, the created nodes were connected in such a way, that a fully mesh topology was generated.

5.3 Self-healing results

The diagnostic system is exactly the same for the two protocols. Therefore the results regarding the self-healing characteristic apply to both of them. In order to test the ability of node discovery, the following skeleton scenario was designed:

30 This reflects the real network, in which the delay of a link between two routers depends on a number of different factors.

- create N number of nodes
- node N1 knows all other neighbouring nodes
- all other nodes (N-1) know all neighbouring nodes except one
- thus, N-1 nodes of the topology need to add one neighbouring node in their list
- start all nodes at 0.010000s

The above skeleton scenario was applied to all experiments, and repeated 50 times for each different network size and network delay factors (0-10, 100-180).

The sample table below summarizes the average of the times that were needed in order to complete each node discovery (marked with bold). These numbers are taken from the trace files of the corresponding simulations. A full set of these files can be found in the CD. Additionally, the table also includes two samples of each scenario.

Nodes	***Constant Delay***	***Discoveries***	***Discovery starts***	***Discovery finishes***	***Discovery lifecycle***	***Average (of 50)***
4	0-10	3	0.012726	0.041810	0.029084	
4	0-10	3	0.012726	0.037810	0.025089	
						0.0289924 s
6	0-10	5	0.014422	0.044810	0.030388	
6	0-10	5	0.014726	0.042810	0.028084	
						0.0294324 s
8	0-10	7	0.010033	0.050974	0.040941	
8	0-10	7	0.014033	0.055974	0.041941	
						0.0417954 s
14	0-10	13	0.010413	0.053413	0.048142	
14	0-10	13	0.013414	0.051942	0.041253	
						0.0454262 s
16	0-10	15	0.0179744	0.054156	0.043818	
16	0-10	15	0.010832	0.062726	0.048106	
						0.0470333 s

Nodes	*Constant Delay*	*Discoveries*	*Discovery starts*	*Discovery finishes*	*Discovery lifecycle*	*Average (of 50)*
18	0-10	17	0.011002	0.051726	0.049276	
18	0-10	17	0.019840	0.050726	0.049114	
						0.0450689 s
4	100-180	3	0.013810	0.045033	0.028777	
4	100-180	3	0.014810	0.098726	0.079084	
						0.5605642 s
6	100-180	5	0.118160	1.249033	1.162567	
6	100-180	5	0.763425	1.091033	0.872392	
						0.8517960 s
8	100-180	7	0.062383	1.060810	0.901573	
8	100-180	7	0.397020	0.049664	0.347356	
						1.0512392 s
14	100-180	13	0.030313	1.355133	1.231548	
14	100-180	13	0.026441	1.833517	1.542489	
						1.5285204 s
16	100-180	15	0.025077	1.413156	1.411921	
16	100-180	15	0.0265686	2.215726	2.251960	
						1.6289525 s
18	100-180	17	0.033258	2.442664	2.390594	
18	100-180	17	0.043868	2.620156	2.623712	
						1.6750659 s

Table 5.2: Sample data from the diagnostic system

As expected, the average (out of 50) of the times that are needed to complete each neighbour's discovery is proportionally increased to the size of the network (number of nodes). It is also worth noticing that although the delay factor increases, it does not

dramatically affect the time LRTN and LRTN_GA need in order to complete a full discovery.

5.4 LRTN & LRTN_GA – Routing results

Experiments also gave to the author the opportunity to create and collect sets of data, which aim to discover the routing capabilities of each protocol. In the same way that the diagnostic system was tested, skeleton simulation scenarios had been created and ran repeatedly. The following bullet-point lines list the skeleton scenario's important schedules:

- apply all discovery rules (as listed in section 5.3)
- routing process in all agents should start at 3.01000 and get repeated every 3 sec (+JIFFER).
- disconnect half of the nodes in the topology at 5.00000s

These rules were applied for all topology sizes, 4, 6, 8, 14, 16, and 18 as well as for the two constant traffic networks, “empty” (0-10) and “busy” (100-180). The results of the two routing protocols LRTN and LRTN_GA are summarized and presented by the following tables.

5.4.1 LRTN – Tables of results

Number of nodes	***Constant Delay***	***Routing lifecycle[31] average (in sec)***	***Routing path cost average (in ms)***
4	0-10	0.194864	155.357
6	0-10	0.256189	170.746
8	0-10	0.280727	183.988
14	0-10	0.431934	349.153
16	0-10	0.567775	386.340
18	0-10	0.637542	451.170
4	100-180	0.615815	587.852
6	100-180	0.752164	651.163

31 This is the required time for a complete routing cycle in the topology.

Number of nodes	*Constant Delay*	*Routing lifecycle[31] average (in sec)*	*Routing path cost average (in ms)*
8	100-180	0.867485	748.636
14	100-180	0.974294	1035.224
16	100-180	1.156702	1157.873
18	100-180	1.274348	1331.990

Table 5.3: LRTN routing results

As shown in the table above, LRTN is increasing the required routing time by less than 0.100s every time two new nodes join or leave the topology for empty networks, and 0.150s for busy ones.

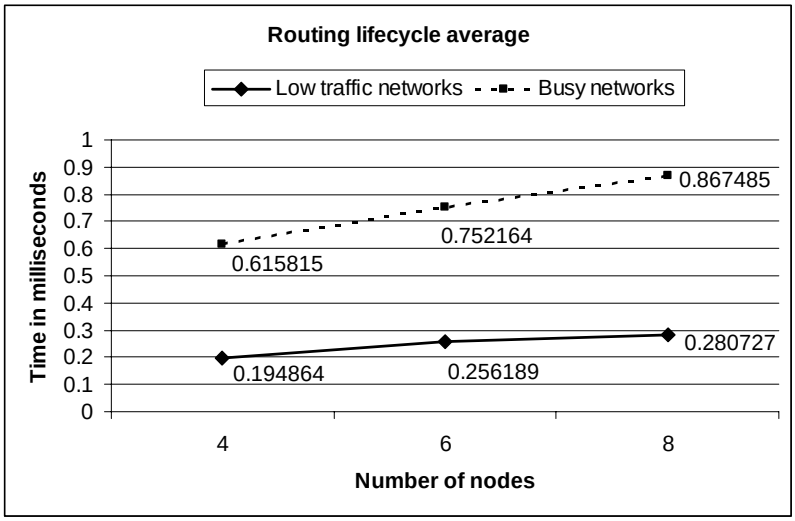


Figure 5.0: (LRTN) Line chart for routing average times for small scale networks

The previous figure as well as the one below illustrates the progress of the average required time to complete a full routing process in the network. As we see, the times are increasing smoothly for either low traffic or busy networks, something that indicates the stability and efficiency of the routing protocol.

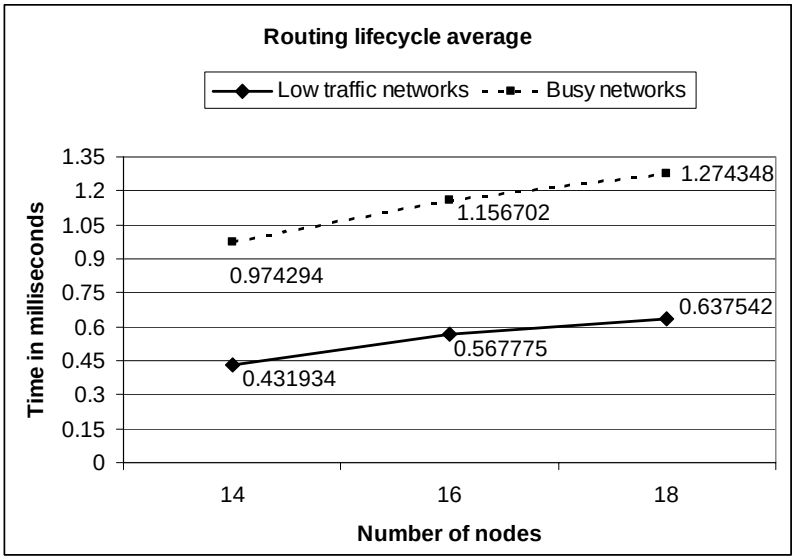


Figure 5.1: (LRTN) Line chart for routing average times for large scale networks

5.4.2 LRTN_GA – Tables of results

Number of nodes	***Constant Delay***	***Routing lifecycle average (in sec)***	***Routing path cost Average (in ms)***
4	0-10	0.103446	147.034
6	0-10	0.129452	263.983
8	0-10	0.139356	253.716
14	0-10	0.140341	401.510
16	0-10	0.155681	596.341
18	0-10	0.161026	754.123
4	100-180	0.378985	801.344
6	100-180	0.411892	1043.880
8	100-180	0.420590	1228.150
14	100-180	0.570402	1536.951
16	100-180	0.624865	1957.006
18	100-180	0.689906	2475.321

Table 5.4: LRTN_GA Routing results

In the case of LRTN_GA things look faster. The required routing time is increased by less than 0.020s which is kept constant regardless any topological change.

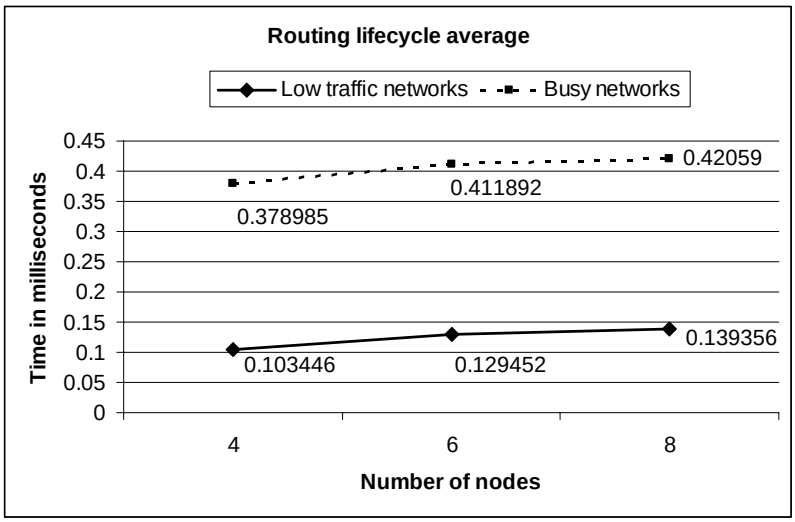


Figure 5.2: (LRTN_GA) Line chart for routing average times for small scale networks

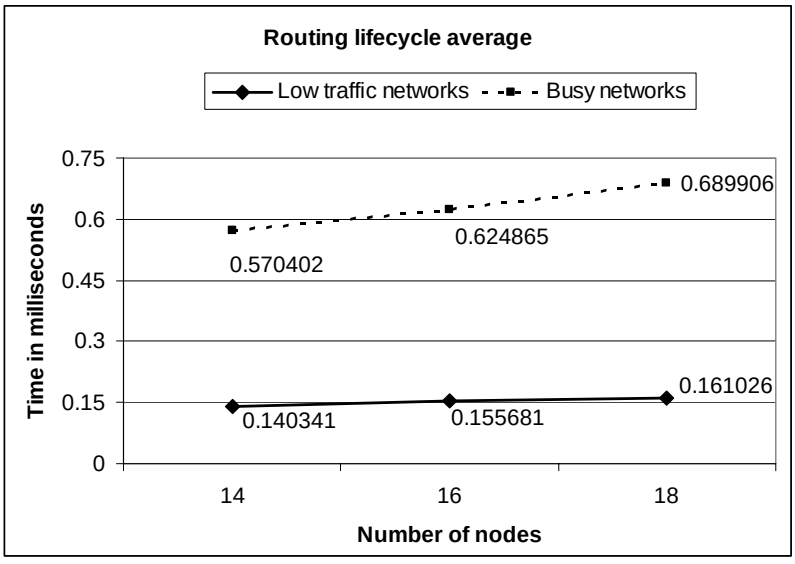


Figure 5.3: (LRTN_GA) Line chart for routing average times for large scale networks

Same as in LRTN figures, the results of the second protocol show that the changes between the average times are increasing smoothly. Another point that can be raised here is that the LRTN_GA routing protocol operates similarly (in terms of speed) for both low traffic and busy networks. This is an expected behaviour, because LRTN_GA does not depend on the packet exchange. Instead it depends on each agent's CPU power and available memory.

5.5 Comparison and discussion

There is a number of comments that can be made regarding the comparison of the two protocols and their different behaviours in terms of speed and efficiency in proposing the most optimal routing path.

Firstly, the experiments exposed that LRTN_GA provide routing solutions faster, in comparison with the static rules of LRTN. This is easily explained, because the searching strategy of LRTN_GA depends on the genetic algorithms and their optimization [39, 48], whereas LRTN's ability to provide solutions depends on the network traffic and the packet transmission (remember the messages-signatures technique).

However, one obvious disadvantage for LRTN_GA is that it does not provide the most optimal solutions particularly in busy and large networks. This has a logical explanation as well. If we recall the GA optimization parameters, the maximum initial population size is set to 150. For networks with small number of nodes, e.g. 4, the search space is equal to the permutation space, thus 24. In cases where the whole of the search space is used during the GA cycles, the result tends to be very accurate. This is not the case for networks of 6 nodes (720 permutations) or more. This is a very important indication that the optimization parameters of LRTN_GA need improvement. On the other hand, although LRTN depends on the network and gets slower as the network traffic increases, it produces the most optimal solutions.

Another important issue in network routing is studied by traffic engineering. Traffic engineering focuses on optimization techniques of an operational network, so that performance requirements are met. It is an essential component of IP networks, especially if the network is large or heavily used by many users and services. In order to compare the two protocols, it is important to measure and compare the traffic that is produced entirely by them.

5.6.1 Traffic engineering – Table of results

Number of nodes	***Constant Delay***	***LRTN Sent pkts***	***LRTN_GA Sent pkts***	***LRTN Recv pkts***	***LRTN_GA Recv pkts***
4	0-10, 100-180	648	434	664	440
6	0-10, 100-180	679	464	687	472
8	0-10, 100-180	943	708	968	758

Number of nodes	*Constant Delay*	*LRTN Sent pkts*	*LRTN_GA Sent pkts*	*LRTN Recv pkts*	*LRTN_GA Recv pkts*
14	0-10, 100-180	1757	1298	1836	1331
16	0-10, 100-180	2106	1531	2143	1552
18	0-10, 100-180	2357	1749	2374	1757

Table 5.5: Traffic results for both protocols

As shown above, LRTN sends and receives more packets than LRTN_GA does in order to complete a routing path finding. This is again expected, because as we previously said LRTN depends on the network traffic and link capacities. Moreover LRTN_GA requires almost the 2/3 of the average LRTN traffic in both sending and receiving packets (as shown in figure 5.4 below).

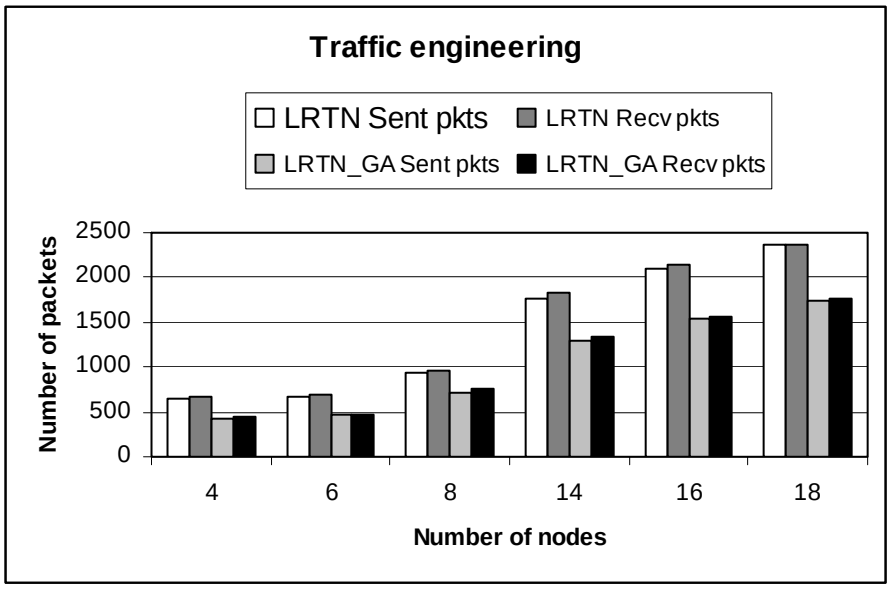


Figure 5.4: Traffic results for both protocols (sent/recv)

The lists below summarize the advantages and disadvantages of LRTN and LRTN_GA:

LRTN advantages:

- fast enough for small networks
- produces optimal solutions
- the average required routing time increases smoothly

LRTN disadvantages:

- depends on the network
- slow for large topologies and/or busy networks
- sends and receives a lot of packets

LRTN_GA advantages:

- equally fast for all types of networks
- the average required routing time increases smoothly
- requires less packet traffic

LRTN_GA disadvantages:

- results are not the most optimal solutions, but they are close to the best ones

5.6 Conclusion

Both LRTN and LRTN_GA have their advantages and disadvantages. Although the static rules of LRTN are slower than the genetic algorithms, they are able to provide better solutions in either small or large network topologies. LRTN_GA on the other hand, tends to be equally fast for both “empty” and “busy” networks. All experiments were made by using a single machine (AMD 3.2 GHz dual core, 2 GB physical memory/512 swap, running Debian GNU/Linux). A complete hierarchy of the simulation results can be found in the CD.

Chapter Six

Critical evaluation and future work

6.0 Introduction

In this chapter, the author provides extra details and alternative techniques that can be applied in respect to the design and the implementation of the two proposed protocols. A critical evaluation is also presented, summarizing pros and cons of the current work.

As an overall conclusion of this dissertation, the author discusses issues for future work and applications that may make use of the two protocols.

6.1 LRTN

Although LRTN's current design and implementation is actually producing valuable routing solutions and useful results, it is a prototype version which contains a number of bad practices. Inside the boundaries of a network simulation the protocol is not expected to crash miserably, however in real a network situation things are going to be rather different.

A good starting point is the technique by which the round trip times are measured. LRTN_DIAGN packets and their acknowledgements play an important role in the lifecycle of the protocol. A misuse or drawback immediately affects the protocol's ability to predict correct and, in some cases, meaningful results. By the current technique the round trip time between two nodes is calculated as the difference between the current time and the time a packet was sent. However, if someone looks deeply in the way ns-2 and LRTN are built, she will discover that this round trip time includes:

- the time the packet needs in order to get created, and leave its sender's queue
- the time it needs to travel through the medium
- the time it needs to get pushed from the receiver's queue
- the time it needs to get handled in the receiver's agent
- the time the receiver's agent needs to construct a new packet and eventually send it back to the initial sender

Additionally, all these happen if and only if the clocks of both sender and receiver are synchronized. A major drawback, if you think of a real network. The solution can be found in

the (more) sophisticated formula of ICMP protocol.

Another important change that needs to be done in the diagnostics system is the use of the sequence numbers of each packet. Although an agent is programmed to prune its list of neighbours after 3 lost LRTN_DIAGN packets, in a real situation where security is required the sequence numbers can be used to match each LRTN_DIAGN with its own LRTN_DIAGN_ACK reply. In fact, as shown in the appropriate chapter, the LRTN header contains a field dedicated to this value. Thus it can be used for routing packets as well, improving not only efficiency but also robustness.

Another characteristic of LRTN that can be classified as a problem, is that it requires a fully mesh topology network in order to discover the participants. This is a simple starting point in the world of the network simulation; however in the real Internet firewalls do not always permit such connectivity. The protocol should have embedded a more sophisticated technique, which would allow each node to know via which close neighbours it can reach others.

Finally, in order to enhance routing within the ring topology, the routing tables could also be improved in such a way that they will be able to keep track of the number of hops or costs (in RTT) between their agents and all other neighbours. By this improvement, each agent will be able to know the distance between itself and a particular destination, thus the direction of each packet will be decided on the fly in a packet per packet basis.

6.2 LRTN_GA

As in LRTN, the synchronization problem still occurs in this second routing protocol. In fact, simulations could be easily crashed in both protocols by starting them with a difference in times. As they are all programmed to repeat some processes periodically, the lack of synchronization will affect their operations. For instance, thinking of the routing process of LRTN_GA, data might not be up-to-date while an agent is working on the GA cycles. Even worst, since they would not operate in parallel, an LRTN_RT_ADV packet might reach an agent that will immediately drop it, because routing at the end of that agent has not been started yet. What is considered as a catastrophe for LRTN_GA, is a usual scenario in the Internet.

Another problem with LRTN_GA is the way it calculates random numbers. At the moment, the function which is used to produce random numbers is the rand() function of the C++ standard library. Although the way random factors are generated seems to work fine, they are closely related to the current clock time (rand is actually seeded by the current time).

Ns-2 provides a better random number generation. The implementation is included in the RNG classes and provides a combined multiple recursive generator MRG32k3a proposed by L'Ecuyer [25].

Many problems of both practical and theoretical importance concert themselves with the choice of a "best" configuration or set of parameters to achieve some goal [31]. Sooner or later, the optimization problem in any GA implementation becomes quite visible. The author's critique covers the frequency by which crossover and mutation operators are applied in each GA cycle. Theory desires each operator to occur based on a predefined probability. However, what is not that efficient in LRTN_GA is the way this probability is being defined and represented. In terms of LRTN_GA, the crossover probability affects the number of crossovers that will take place in the current population. In other words, 90% crossover probability means that 90% of the current population size will be subject to crossover operations. A conversion from double to integer helps in order to get rid of the unwanted decimals. The same applies in case of mutation. The following table illustrates the number of crossover and mutation cycles that will take place in each GA cycle:

Current population	***Crossover cycles (90%)***	***Mutation cycles (2%)***
120	108	2
90	81	1
50	45	1
30	27	0
10	9	0
5	4	0

Table 6.0: Parameters for the two genetic operators

In a more sophisticated system than this prototype, a function similar to the way roulette wheel selection emulates the throwing of the ball, could be used in order to measuring the operator's bias.

The fitness function might also need improvement. Since the only factor that affects the cost of each connection is the round trip time, a more complex fitness function could compare each individual's fitness with the fitness of a predefined minimum acceptable solution. This could also be considered as another stopping condition.

6.3 Future work

As shown in the previous section, there are many things that could be modified and added, in order to allow both protocols work in real situation scenarios.

In order to improve both protocols, future work is very welcome. In addition to what is included in the sections 6.1 and 6.2, future work might also cover features such as support of different topologies and security enhancements by encryption and authentication.

6.4 Application that may use LRTN and LRTN_GA

Closely related to future work, this section aims to present ideas regarding the type of applications that might make use of LRTN and LRTN_GA.

Being network routing protocols, their primary objective is to provide fast and reliable communicational solutions to other distributed systems. Both LRTN and LRTN_GA can be used when message propagation in a rounded manner is required. This might involve software automatic updates or vulnerability checking.

Moreover applications of the system administration area could take advantage of such a protocol, in order to control a number of systems for operations such as passing simple commands like “start a daemon” or “send the Apache log file”.

6.5 Conclusion

“Technologies related to networks and inter-networking may be the fastest growing in our culture today. One of the ramifications of that growth is a dramatic increase in the number of professions where an understanding of these technologies is essential for success”, Forouzan B.A.

Network and in particular networking routing is one of the most important areas in computing. Not only because it allows the existence of distributed systems, but also because via these distributed systems, people are able to share their knowledge, thoughts, interests and joy. Although LRTN and LRTN_GA are simple prototype versions, they offered to the author a priceless background in network design and programming.

Bibliography

[1] B. A. Forouzan, "*TCP/IP Protocol Suite*", McGraw Hill, 3rd Ed., 2006

[2] D. Medhi, K. Ramasamy. "*Network Routing, Algorithms, Protocols and Architectures*", Elsevier, 2007

[3] B. A. Forouzan, *"Data Communications and Networking"*, McGraw Hill, 4th Ed., 2007

[4] Goldberg D. E., *"Genetic Algorithms in Search, Optimization & Machine Learning"*, Reading, MA: Addison Wesley, 1989

[5] Murphy R. R., "*Introduction to AI Robotics*", MIT Press, 2000

[6] Cawsey A., "*The essence of Artificial Intelligence*", Prentice Hall, 1998

[7] Wilf H. S., "*Algorithms and Complexity*", A K Peters, 2nd Ed., 2002

[8] Graham R. L, Knuth D. E., Patashnik O., "*Concrete mathematics: A foundation for computer science*", Addison-Wesley, 2nd Ed., 1998

[9] Fall K., Varadhan K., "*The ns Manual (formerly ns Notes and Documentation)*", 2008

[10] Airoldi E. M., Carley K. M., *"Sampling Algorithms for Pure Network Topologies"*, SIGKDD Explorations, Vol 7, Issue 2, 2005, pp. 13-22

[11] Routray S., Sherry A.M., Reddy B.V.R, "*ATM Network Planning: A Genetic Algorithm Approach*", JATIT, 2005-2007, pp. 74-79

[12] Yaged, B., "*Minimum cost routing for static network models"*, Networks, vol. 1, pp. 139-172. 1971

[13] Chin K., *"The Behaviour of MANET Routing Protocol in Realistic Environments"*, Asia-Pacific Conference on Communications, Perth, Western Australia, 3 - 5 October 2005, http://ro.uow.edu.au/infopapers/61 accessed on 27/06/2008

[14] Altman E., Jimenez T., *"NS Simulator for beginners"*, Lecture Notes 2003-2004, Univ. de Los Andes, Merida, Venezuela and ESSI, Sophia-Antipolis, France, Dec 2003

[15] Perbellini G., *"Network advanced modelling in NS-2"*, UniversitÃ degli Studi di Verona, FacoltÃ di Scienze MM. FF. NN. Dipartimento di Informatica, EDALab: Embedded System Design Center, 2005

[16] Mohapatra P., Gui C., *"SHORT: Self-Healing and Optimizing Routing Techniques for Mobile Ad Hoc Networks"*, ACM, 2003, pp. 279-290

[17] Davis, L., Cox, A., and Qiu, Y., *"A Genetic Algorithm for Survivable Network Design in Proceedings of the Fifth International Conference on Genetic Algorithms"*, Morgan Kauffman, 1993, pp 408-415

[18] Davis, L., and Coombs, S., *"Genetic Algorithms and Communication Link Speed Design: Theoretical Considerations in Proceedings of the Second International Conference on Genetic Algorithms"*, Lawrence Erlbaum, 1987, pp 252-256

[19] Coombs, S., and Davis, L., *"Genetic Algorithms and Communication Link Speed Design: Constraints and Operators in Proceedings of the Second International Conference on Genetic Algorithms*", Lawrence Erlbaum, 1987, pp 257-260

[20] Steiglitz, K., Weiner, P., and Kleitman, D. J., "*The design of minimum cost networks*", IEEE Transactions on Circuit Theory, pp. 455-460, Nov. 1969

[21] R. Elbaum and M. Sidi, *"Topological Design of Local Area Networks Using Genetic Algorithms"*, Proc. IEEE Infocom 95, IEEE Computer Society Press, Los Alamitos, Calif., 1995, pp. 64-71

[22] King-Tim Ko, Kit-Sang Tang, Cheung-Yau Chan, Kim-Fung Man, *"Using Genetic Algorithms to Design Mesh Networks"*, IEEE, Aug. 1997, pp. 56-61

[23] Coppin B., *"Artificial Intelligence Illuminated"*, Jones and Bartlett Publishers, 2004

[24] Russell S. J., Norvig P., *"Artificial Intelligence: A Modern Approach"*, Prentice Hall,

2nd Ed. 2003

[25] Pierre L'Ecuyer., *"Good parameters and implementations for combined multiple recursive random number generators"*. Operations Research , 47(1):159–164, 1999

[26] "The Network Simulator ns-2: Frequently Asked Questions", http://www.isi.edu/nsnam/ns/ns-faq.html last accessed on 14/06/2008

[27] "*Using ns and nam in Education*", http://www.isi.edu/nsnam/ns/edu/index.html, last accessed on 14/06/2008

[28] Yogen K. Dalal , Robert M. Metcalfe, "*Reverse path forwarding of broadcast packets*", Communications of the ACM, v.21 n.12, p.1040-1048, Dec 1978

[29] RFC3684, Feb 2004, "*Topology Dissemination Based on Reverse-Path Forwarding (TBRPF)*", ftp://ftp.rfc-editor.org/in-notes/rfc3684.txt

[30] "*A Guide to Network Topology*", http://learn-networking.com/network-design/a-guide-to-network-topology, last accessed at 18/07/2008

[31] Papadimitriou C., Steiglitz K., "*Combinatorial Optimization: Algorithms and Complexity"*, Dover, 1998

[32] Hedrick C. L., "*Routing Information Protocol"*, IETF RFC 1058, June 1988. http://www.rfc-editor.org/rfc/rfc1058.txt

[33] Leiner B., "*Policy issues interconnected networks"*, IETF RFC1124, September 1989, (available only as postscript or PDF file), http://www.faqs.org/rfc/rfc1124.pdf

[34] Dreyfus, S. E., "*An Appraisal of Some Shortest-Path Algorithms"*, OR, 17 (1969), 395-412

[35] Moy J., "*OSPF version 2"*, IETF RFC 2328, April 1998, http://www.rfc-editor.org/rfc/rfc2328.txt

[36] Moy J., *"OSPF: Anatomy of An Internet Routing Protocol"*, Addison-Wesley, 1998

[37] Moy, J., *"The OSPF specification"*, IETF RFC 1131, (available only as postscript or PDF file), http://www.faqs.org/rfc/rfc1131.pdf

[38] Werner J.C., Fogarty T.C., *"Map algorithm in routing problems using genetic algorithm."*, IFORS Triennial Conference – Edinburgh/Scotland July 8-12, 2002

[39] Davies C., Lingras P., *"Discrete Optimization Genetic algorithms for rerouting shortest paths in dynamic and stochastic networks"*, ELSEVIER, European Journal of Operational Research 144 (2003), pp. 27–38

[40] George S., Evans D., Marchette S., *"A Biological Programming Model for Self-Healing"*, ACM, Oct 2003

[41] Low S.H., Peterson L., and Wang L. *"Understanding TCP Vegas: A Duality Model"*, http://www.cs.princeton.edu/nsg/papers/vegas.html, accessed on 11th June 2008

[42] Bikram Bakshi, Krishna, P., Pradhan, D.K., and Vaidya, N.H., *"Performance of TCP over Wireless Networks "*, http://www.cs.tamu.edu/faculty/vaidya/Vaidya-mobile.html, accessed on 11 June 2008, 17th Intl. Conf. on Distributed Computing Systems (ICDCS), Baltimore, May, 1997

[43] Birman K.P., Hayden M., Ozkasap O., Xiao Z., Budiu M., Minsky Y., *"Bimodal Multicast"*, ACM, 1999, 17(2), pp.41-88

[44] Henderson T.R., Katz R.H., *"Transport protocols for Internet-compatible satellite networks"*, IEEE, Vol. 17, No. 2, 1999, pp. 345-359

[45] Wall K., *"Tcl and Tk Programming for the Absolute Beginner"*, Delmar, 1st Ed., 2007

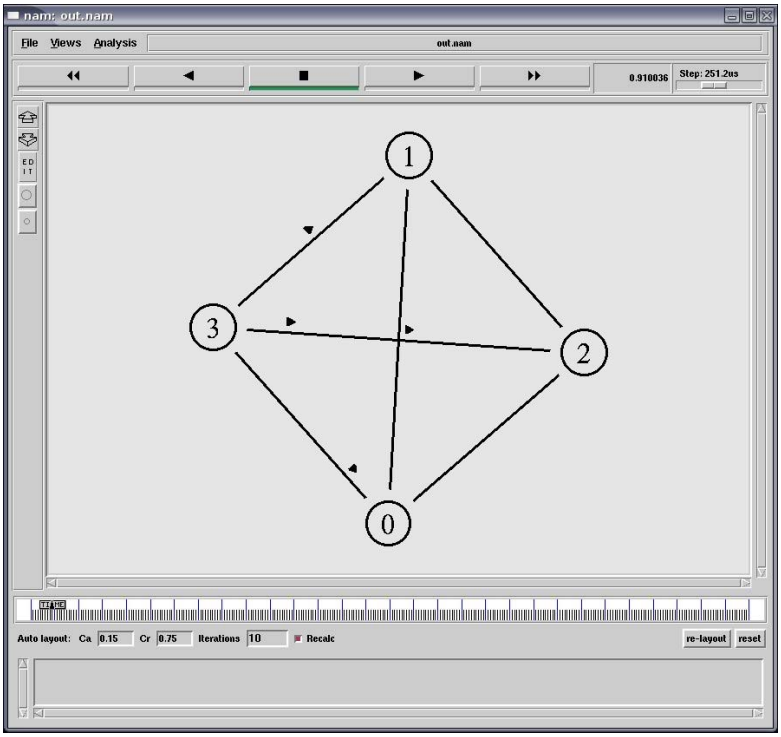
[46] Stroustrup B., *"The C++ Programming Language"*, Addison Wesley, 3rd Ed., 2000

[47] Awerbuch B., *"A new distributed depth-first-search algorithm, Information Processing Letters"*, Vol. 20,3, 1985, pp. 147

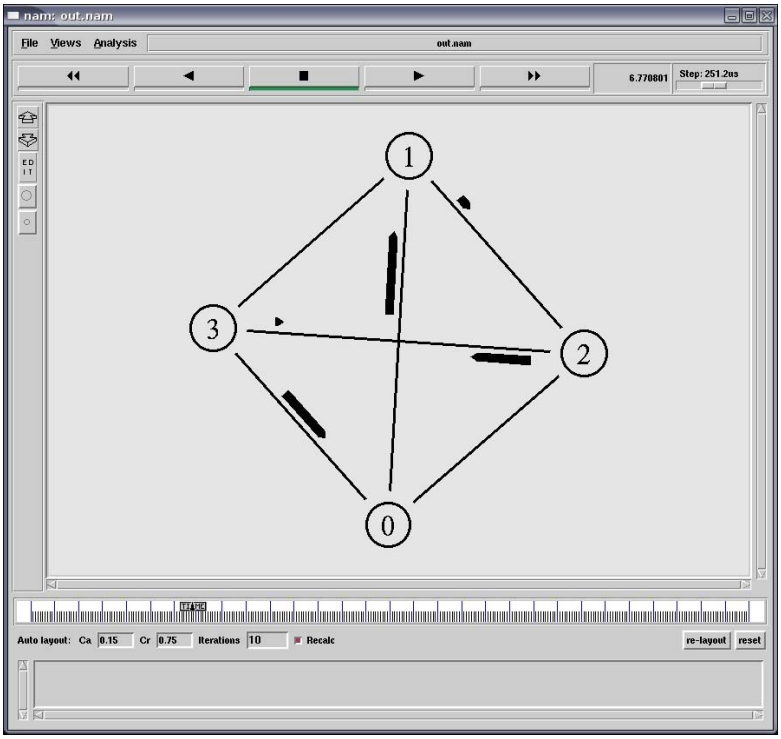
[48] Stan van Hoesel, *"Optimization in telecommunication networks"*, Statistica Neerlandica, Vol. 59, nr. 2, 2005, pp. 180

[49] Deepinder S., Tayang F., Shukri A., Raj N., *"Open Shortest Path First (OSPF) Routing Protocol Simulation"*, ACM, 1993

Appendix I



Screenshot 1: LRTN_DIAGN and LRTN_DIAGN_ACK packet flow (LRTN_GA)



Screenshot 2: Traffic generation & diagnostic packet flow (LRTN)

Appendix II

```
##
# lrtn_core.tcl - for either LRTN or LRTN_GA
#
# ns-2 simulation script
#
# MSc major Project:
# "Routing for Logical Ring Topology Networks
# A comparison of two methods"
#
# Author: Alexandros I. Giagkos
# Supervised by: Dr Myra S Wilson
#
# Date: 10/07/08
##

remove-all-packet-headers
add-packet-header CMN IP LRTN

# random number generator: min + (random * (max - min))
set ns [new Simulator]

# set nodes to N - 1, where N is the number of nodes
set nodes 39
set nf [open out.nam w]
set tf [open out.tr w]
$ns namtrace-all $nf
$ns trace-all $tf

proc finish {} {
        global ns nf tf
        $ns flush-trace
        close $nf
        close $tf
#        exec nam out.nam &
        exit 0
}
for { set k 0 } { $k <= $nodes } { incr k } {
       set node($k) [$ns node]
       set agent($k) [new Agent/LRTN $k]
       $ns attach-agent $node($k) $agent($k)
}
for { set i 0 } { $i <= $nodes } { incr i } {
       for { set j 1 } { $j <= $nodes } { incr j } {
              set r [expr int( rand() * 180)]
              if { $i != $j } {
              $ns duplex-link $node($i) $node($j) 10Mb $r\ms DropTail
              #$ns duplex-link $node($i) $node($j) 10Mb 10ms DropTail
              $ns connect $agent($i) $agent($j)
              $agent($i) add_neighbor $node($j)
              }
       }
}
for { set i 0 } { $i <= 20 } { incr i } {
 $ns at 0.01 "$agent($i) start"
}
for { set i 21 } { $i <= $nodes } { incr i } {
 $ns at 5.01 "$agent($i) start"
}

$ns at 10.0 "finish"
$ns run
```

Appendix III

```
/*
 * lrtn_core.h - LRTN
 * Header file for the LRTN Agent class
 *
 * MSc major Project:
 * "Routing for Logical Ring Topology Networks
 * A comparison of two methods"
 *
 * Author: Alexandros I. Giagkos
 * Supervised by: Dr Myra S Wilson
 *
 * Date: 14/06/08
 */

#include "classifier-port.h"
#include "random.h"
#include "trace.h"
#include "scheduler.h"
#include "address.h"
#include "ip.h"
#include "agent.h"
#include "packet.h"
#include "object.h"
#include "lrtn_rt.h"
#include <map>

#ifndef _lrtn_core_h_
#define _lrtn_core_h_

#define TRUE 1
#define FALSE 0
#define NULl 0
#define NEIGHBORS_DIAGN_PERIOD 0.15
#define ROUTING_PERIOD 3.00
#define TRAFFIC_PERIOD 0.01
#define JITTER (Random::uniform()*0.05)
#define DIAGN_REP_MAX 2
#define CURRENT_TIME Scheduler::instance().clock()

enum LRTNTYPE {
       LRTN_DIAGN, LRTN_DIAGN_ACK, LRTN_RT_MSG, LRTN_RT_ADV, LRTN_TRAFFIC
};

struct fullmesh_neighbor {
       nsaddr_t addr_;
       int rep_times_;
       double rtt_;
};

typedef map<nsaddr_t, fullmesh_neighbor*, less<int> > NEI_MAP;

/* predefine this class for timers */
class LRTNAgent;

/* predefine this struct */
struct rt_message;
struct rt_entry;

/* LRTN pkt header */
struct hdr_lrtn {

       /* LRTN common */
       int type_;
```

```
        int seq_;
        nsaddr_t saddr_;
        nsaddr_t daddr_;

        /* LRTN_DIAGN & ACK */
        double sent_time_;
        int nei_paths_num_;
        nsaddr_t *nei_paths_data_;

        /* LRTN_RT_MSG */
        nsaddr_t rt_message_originator_;
        int rt_message_participants_num_;
        rt_message *rt_entries_data_;
        int rt_message_rcnt_;
        int rt_rtt_;

        /* LRTN_RT_ADV */
        double rt_prouting_rtt_;

        static int offset_;
        inline static int& offset() { return offset_; }
        inline static hdr_lrtn* access(const Packet* p) {
                return (hdr_lrtn*) p->access(offset_);
        }
};

class LRTNNeighborDiagnTimer : public Handler {
 public:
        LRTNNeighborDiagnTimer(LRTNAgent* a) { a_ = a; }
        void handle(Event*);
        void start(double t);
        Event intr;
        LRTNAgent *a_;
};

class LRTNRoutingTimer : public Handler {
 public:
        LRTNRoutingTimer(LRTNAgent* a) { a_ = a; }
        void handle(Event*);
        void start(double t);
        Event intr;
        LRTNAgent *a_;
};

class LRTNTrafficTimer : public Handler {
 public:
        LRTNTrafficTimer(LRTNAgent* a) { a_ = a; }
        void handle(Event*);
        void start(double t);
        int isactive_;
        Event intr;
        LRTNAgent *a_;
};

/* predefine this class for LRTNAgent */
class LRTNRoutingTable;

struct rt_message;

class LRTNAgent : public Agent {
 friend class LRTNNeighborDiagnTimer;
 friend class LRTNRoutingTimer;
 friend class LRTNTrafficTimer;
 public:
        LRTNAgent();
```

```
        LRTNAgent(nsaddr_t addr);
        int command (int argc, const char*const* argv);
        void recv(Packet *p, Handler *);
        void start();
        void stop();
        int initialized() { return 1 && target_; }

        int neighbors_add(nsaddr_t neighboraddr);
        void neighbors_diagncall();
        void neighbors_print();
        void neighbors_prune();
        void create_nei_paths_data_space(Packet*, int);
        void fill_nei_paths_data_space(Packet*);

        void send_rt_message(int, rt_message*, rt_entry*);
        void parse_rt_message(Packet*);
        void parse_rt_adv(Packet*);
        void process_routing();

        void transmit_pkt(Packet*, nsaddr_t);
        void send_traffic_pkt();

 protected:
        nsaddr_t agent_addr_;
        int isactive_;
        int glob_seq_;
        int t_recv_pkts_;
        int t_sent_pkts_;

        NEI_MAP neighbors_;
        NEI_MAP::iterator nei_iter;
        int neighbors_diagncycles_cnt_;
        double neighborsdiagn_period_;
        double neighborsdiagn_jitter_;
        int nei_stability_;
        int routing_stability_;

        LRTNRoutingTable* rtable_;

        PortClassifier* dmux_;
        Trace* logtarget_;

        LRTNNeighborDiagnTimer neighborsdiagn_timer_;
        LRTNRoutingTimer routing_timer_;
        LRTNTrafficTimer traffic_timer_;

        int allow_traffic_;
};

#endif
```

```
/*
 * lrtn_rt.h - LRTN
 * Header for the LRTNRoutingTable class
 *
 * MSc major Project:
 * "Routing for Logical Ring Topology Networks
 * A comparison of two methods"
 *
 * Author: Alexandros I. Giagkos
 * Supervised by: Dr Myra S Wilson
 *
 * Date: 14/06/08
 */

#ifndef _lrtn_rt_h_
#define _lrtn_rt_h_

#include "lrtn_core.h"

struct rt_entry {
       nsaddr_t addr_;
       double rtt_;
};

struct rt_message {
       nsaddr_t addr_;
       double rtt_to_next_;
       rt_message *next_;
};

class LRTNRoutingTable {
 public:
       LRTNRoutingTable(nsaddr_t addr);
       nsaddr_t agent_addr_;
       NEI_MAP nodes_;
       int num_entries_;

       double rt_time_;

       rt_entry **rt_message_entries_;
       nsaddr_t rt_message_next_hop_;

       rt_message *rt_potential_rtable_;
       double rt_potential_rtable_rtt_;
       int rt_potential_rtable_entries_;

       nsaddr_t sender_; //  the node that sends me pkts
       nsaddr_t receiver_; // next hop

       int update_rtable(NEI_MAP);
       rt_message* build_new_rt_message();
       rt_entry* find_rt_message_next_hop();
       rt_entry* find_rt_message_next_hop(rt_message*);
         int compare_proutings(int, rt_message*, double);
         void fix_sender_receiver();
};

#endif
```

```
/*
 * lrtn_core.h - Header file for the LRTN_GA Agent class
 *
 * MSc major Project:
 * "Routing for Logical Ring Topology Networks
 * A comparison of two methods"
 *
 * Author: Alexandros I. Giagkos
 * Supervised by: Dr Myra S Wilson
 *
 * Date: 20/06/08
 */

#include "classifier-port.h"
#include "random.h"
#include "trace.h"
#include "scheduler.h"
#include "address.h"
#include "ip.h"
#include "agent.h"
#include "packet.h"
#include "object.h"
#include "lrtn_rt.h"
#include <map>
#include <vector>
#include <string>

#ifndef _lrtn_core_h_
#define _lrtn_core_h_

#define TRUE 1
#define FALSE 0
#define NEIGHBORS_DIAGN_PERIOD 0.15
#define ROUTING_PERIOD 3.00
#define TRAFFIC_PERIOD 0.01
#define JITTER (Random::uniform()*0.05)
#define DIAGN_REP_MAX 2
#define MAX_POPULATION_SIZE 150
#define MAX_GA_CYCLES 20
#define CROSSOVER_PERCENTAGE 0.90
#define MUTATION_PERCENTAGE 0.02
#define CURRENT_TIME Scheduler::instance().clock()

enum LRTNTYPE {
     LRTN_DIAGN, LRTN_DIAGN_ACK, LRTN_RTT_TABLE, LRTN_RT_ADV, LRTN_TRAFFIC
};

struct fullmesh_neighbor {
     nsaddr_t addr_;
     int rep_times_;
     double rtt_;
};

typedef map<nsaddr_t, fullmesh_neighbor*, less<int> > NEI_MAP;
typedef map<std::string, vector<nsaddr_t> > ADDRV_MAP;
typedef map<double, vector<nsaddr_t> > ROULETTE_WHEEL_MAP;

/* predefine this class for timers */
class LRTNAgent;

/* predefine this struct */
struct rt_table_entry;
struct rt_gene;

/* LRTN pkt header */
```

```
struct hdr_lrtn {

       /* LRTN common */
       int type_;
       int seq_;
       nsaddr_t saddr_;
       nsaddr_t daddr_;

      /* LRTN_DIAGN & ACK */
      double sent_time_;
      int nei_paths_num_;
      nsaddr_t *nei_paths_data_;

      /* LRTN_RTT_TABLE */
      int rtt_table_num_;
      rt_table_entry** rtt_table_data_;

      /* LRTN_RT_ADV */
      int rt_prouting_num_;
      double rt_prouting_fit_;
      rt_gene* rt_prouting_data_;

      static int offset_;
      inline static int& offset() { return offset_; }
      inline static hdr_lrtn* access(const Packet* p) {
            return (hdr_lrtn*) p->access(offset_);
      }
};

class LRTNNeighborDiagnTimer : public Handler {
 public:
      LRTNNeighborDiagnTimer(LRTNAgent* a) { a_ = a; }
      void handle(Event*);
      void start(double t);
      Event intr;
      LRTNAgent *a_;
};

class LRTNRoutingTimer : public Handler {
 public:
     LRTNRoutingTimer(LRTNAgent* a) { a_ = a; }
     void handle(Event*);
     void start(double t);
     Event intr;
     LRTNAgent *a_;
};

class LRTNTrafficTimer : public Handler {
 public:
      LRTNTrafficTimer(LRTNAgent* a) { a_ = a; }
      void handle(Event*);
      void start(double t);
      int isactive_;
      Event intr;
      LRTNAgent *a_;
};

/* predefine this class for LRTNAgent */
class LRTNRoutingTable;

class LRTNAgent : public Agent {
 friend class LRTNNeighborDiagnTimer;
 friend class LRTNRoutingTimer;
 friend class LRTNTrafficTimer;
 friend class LRTNRoutingTable;
```

```
 public:
       LRTNAgent();
       LRTNAgent(nsaddr_t addr);
       int command (int argc, const char*const* argv);
       void recv(Packet *p, Handler *);
       void start();
       void stop();
       int initialized() { return 1 && target_; }

       int neighbors_add(nsaddr_t neighboraddr);
       void neighbors_diagncall();
       void neighbors_print();
       void neighbors_prune();
       void create_nei_paths_data_space(Packet*, int);
       void fill_nei_paths_data_space(Packet*);

       void send_local_rtt_table();
       void create_initial_population();
       unsigned long perm_space(int);
       void add_rtt_table(Packet*);
       void fill_up_roulette_wheel(ADDRV_MAP);
       double fitness_function(vector<nsaddr_t>);
       double probability_function(std::string, ADDRV_MAP);
       double population_average_fitness(ADDRV_MAP);
       vector<nsaddr_t> crossover_1point_function(vector<nsaddr_t>,
vector<nsaddr_t>);
       vector<nsaddr_t> mutation_function(vector<nsaddr_t>);
       void send_local_rt_path(vector<nsaddr_t>);
       void parse_rt_adv(Packet*);
       void ga_start();

       void transmit_pkt(Packet*, nsaddr_t);
       void send_traffic_pkt();

 protected:
        nsaddr_t agent_addr_;
       int isactive_;
       int glob_seq_;
       int t_recv_pkts_;
       int t_sent_pkts_;

       NEI_MAP neighbors_;
       NEI_MAP::iterator nei_iter;
       int neighbors_diagncycles_cnt_;
       double neighborsdiagn_period_;
       double neighborsdiagn_jitter_;
       int nei_stability_;

       LRTNRoutingTable* rtable_;
       ADDRV_MAP addrV_population_;
       ROULETTE_WHEEL_MAP roulette_wheel_;

       PortClassifier* dmux_;
       Trace*logtarget_;

       LRTNNeighborDiagnTimer neighborsdiagn_timer_;
       LRTNRoutingTimer routing_timer_;
       LRTNTrafficTimer traffic_timer_;

       int allow_traffic_;
       int rtt_tables_received_;
};

#endif
```

```
/*
 * lrtn_rt.h – LRTN_GA
 * Header file for the LRTNRoutingTable class
 *
 * MSc major Project:
 * "Routing for Logical Ring Topology Networks
 * A comparison of two methods"
 *
 * Author: Alexandros I. Giagkos
 * Supervised by: Dr Myra S Wilson
 *
 * Date: 20/06/08
 */

#ifndef _lrtn_rt_h_
#define _lrtn_rt_h_

#include "lrtn_core.h"
#include <string>
#include <map>

typedef map<std::string, double > PAIRS_RTT_MAP;

struct rt_table_entry {
       nsaddr_t daddr_;
       double rtt_;
};

struct rt_table {
       nsaddr_t saddr_;
       int rt_table_num_entries_;
       rt_table_entry** data_;
};

struct rt_gene {
       nsaddr_t addr_;
       rt_gene *next_;
};

class LRTNRoutingTable {
 public:
       LRTNRoutingTable(nsaddr_t addr);
       nsaddr_t agent_addr_;

       rt_table* local_rt_table_;
         PAIRS_RTT_MAP pairs_rtt_;
         PAIRS_RTT_MAP::iterator pairs_rtt_iter;

       int update_local_rt_table(NEI_MAP);

       rt_gene* winning_route_;
       double winning_route_fit_;
};

#endif
```

Appendix IV

Simulation time: 0s – 4s
Number of nodes: 4
Protocol: LRTN
Constant delay: 0s – 10s (low traffic)

[0.010000] Agent 0 started.
[0.010000] Agent 1 started.
[0.010000] Agent 2 started.
[0.010000] Agent 3 started.
[0.010033] Agent 1: new nei added..
[0.010033] Agent 1: instability detected, routing process started.
[0.015726] Agent 3: new nei added..
[0.015726] Agent 3: instability detected, routing process started.
[0.038810] Agent 2: new nei added..
[0.038810] Agent 2: instability detected, routing process started.
[3.010000] Agent 0: periodic routing process started.
[3.010000] Agent 1: periodic routing process started.
[3.010000] Agent 2: periodic routing process started.
[3.010000] Agent 3: periodic routing process started.
[3.089881] Agent 1: full rt_message arrived: 1 (23.872497 to reach next) 3 (22.680789 to reach next) 0 (36.087851 to reach next) 2 (58.161924 to reach next) RTT: 140.803061 ms
[3.089881] Agent 1: I found a better rtable.
I'm agent 1 and this is my potential rtable (4) : 1 3 0 2 - f-1 l-2
[3.106556] Agent 2: full rt_message arrived: 2 (55.265099 to reach next) 1 (23.872497 to reach next) 3 (22.680789 to reach next) 0 (47.991673 to reach next) RTT: 149.810058 ms
[3.106556] Agent 2: I found a better rtable.
I'm agent 2 and this is my potential rtable (4) : 2 1 3 0 - f-2 l-0
[3.110090] Agent 3: I found a better rtable.
I'm agent 3 and this is my potential rtable (4) : 1 3 0 2 - f-1 l-2
[3.111288] Agent 0: full rt_message arrived: 0 (36.087851 to reach next) 2 (55.265099 to reach next) 1 (23.872497 to reach next) 3 (45.884175 to reach next) RTT: 161.109623 ms
[3.111288] Agent 0: I found a better rtable.
I'm agent 0 and this is my potential rtable (4) : 0 2 1 3 - f-0 l-3
[3.112937] Agent 2: I have a faster rtable.
[3.121233] Agent 3: I have a faster rtable.
[3.123619] Agent 0: I found a better rtable.
I'm agent 0 and this is my potential rtable (4) : 1 3 0 2 - f-1 l-2
[3.125989] Agent 3: I have a faster rtable.
[3.128557] Agent 2: I found a better rtable.
I'm agent 2 and this is my potential rtable (4) : 1 3 0 2 - f-1 l-2
[3.134549] Agent 1: I have a faster rtable.
[3.138058] Agent 3: full rt_message arrived: 3 (22.680789 to reach next) 0 (36.087851 to reach next) 2 (55.265099 to reach next) 1 (30.496921 to reach next) RTT: 144.530660 ms
[3.138058] Agent 3: I have a faster rtable.
[3.144224] Agent 0: I have a faster rtable.
[3.157650] Agent 1: I have a faster rtable.

Simulation time: 0s – 4s
Number of nodes: 4
Protocol: LRTN
Constant delay: 100s – 180s (busy network)

[0.010000] Agent 0 started.
[0.010000] Agent 1 started.
[0.010000] Agent 2 started.
[0.010000] Agent 3 started.
[0.031033] Agent 1: new nei added..
[0.031033] Agent 1: instability detected, routing process started.
[0.117726] Agent 3: new nei added..
[0.117726] Agent 3: instability detected, routing process started.
[0.155810] Agent 2: new nei added..
[0.155810] Agent 2: instability detected, routing process started.
[3.010000] Agent 0: periodic routing process started.
[3.010000] Agent 1: periodic routing process started.
[3.010000] Agent 2: periodic routing process started.
[3.010000] Agent 3: periodic routing process started.
[3.385422] Agent 1: full rt_message arrived: 1 (93.348518 to reach next) 0 (90.255318 to reach next) 2 (230.227483 to reach next) 3 (275.738398 to reach next) RTT: 689.569718 ms
[3.385422] Agent 1: I found a better rtable.
I'm agent 1 and this is my potential rtable (4) : 1 0 2 3 - f-1 l-3
[3.388285] Agent 0: full rt_message arrived: 0 (90.255318 to reach next) 1 (93.348518 to reach next) 2 (203.692261 to reach next) 3 (275.738398 to reach next) RTT: 663.034496 ms
[3.388285] Agent 0: I found a better rtable.
I'm agent 0 and this is my potential rtable (4) : 0 1 2 3 - f-0 l-3
[3.411246] Agent 2: full rt_message arrived: 2 (217.326563 to reach next) 1 (93.348518 to reach next) 0 (91.299466 to reach next) 3 (286.309700 to reach next) RTT: 688.284248 ms
[3.411246] Agent 2: I found a better rtable.
I'm agent 2 and this is my potential rtable (4) : 2 1 0 3 - f-2 l-3
[3.412056] Agent 3: full rt_message arrived: 3 (247.057910 to reach next) 0 (91.299466 to reach next) 1 (134.777110 to reach next) 2 (227.313352 to reach next) RTT: 700.447837 ms
[3.412056] Agent 3: I found a better rtable.
I'm agent 3 and this is my potential rtable (4) : 3 0 1 2 - f-3 l-2
[3.418471] Agent 0: I have a faster rtable.
[3.420165] Agent 1: I found a better rtable.
I'm agent 1 and this is my potential rtable (4) : 0 1 2 3 - f-0 l-3
[3.500917] Agent 2: I have a faster rtable.
[3.515378] Agent 1: I have a faster rtable.
[3.517694] Agent 3: I found a better rtable.
I'm agent 3 and this is my potential rtable (4) : 1 0 2 3 - f-1 l-3
[3.518828] Agent 3: I found a better rtable.
I'm agent 3 and this is my potential rtable (4) : 0 1 2 3 - f-0 l-3
[3.524180] Agent 2: I have a faster rtable.
[3.527137] Agent 2: I found a better rtable.
I'm agent 2 and this is my potential rtable (4) : 0 1 2 3 - f-0 l-3
[3.538505] Agent 1: I have a faster rtable.
[3.542067] Agent 3: I have a faster rtable.
[3.554572] Agent 0: I have a faster rtable.
[3.569487] Agent 0: I have a faster rtable.

Simulation time: 0s – 7s
Number of nodes: 4
Protocol: LRTN_GA
Constant delay: 0s – 10s (low traffic)

[0.010000] Agent 0 started.
[0.010000] Agent 1 started.
[0.010000] Agent 2 started.
[0.010000] Agent 3 started.
[0.012726] Agent 3: new neighbor added..
[0.019033] Agent 1: new neighbor added..
[0.041810] Agent 2: new neighbor added..
[3.010000] Agent 0: periodic routing process started.
[3.010000] Agent 0: Total sent 107, total received 103 pkts
[3.010000] Agent 1: periodic routing process started.
[3.010000] Agent 1: Total sent 109, total received 106 pkts
[3.010000] Agent 2: periodic routing process started.
[3.010000] Agent 2: Total sent 107, total received 104 pkts
[3.010000] Agent 3: periodic routing process started.
[3.010000] Agent 3: Total sent 109, total received 104 pkts
permutation space is 24
2 0 3 1
[3.056569] Agent 2: winner's fitness: 172.023
permutation space is 24
2 0 3 1
[3.058941] Agent 3: winner's fitness: 172.023
[3.062994] Agent 0: a new rtable came and mine is NULL.
permutation space is 24
2 0 3 1
[3.064835] Agent 1: winner's fitness: 172.023
permutation space is 24
2 0 3 1
[3.068067] Agent 0: winner's fitness: 172.023
[3.068340] Agent 2: I have a faster rtable.
[3.074194] Agent 1: I have a faster rtable.
[3.089196] Agent 0: I have a faster rtable.
[3.090276] Agent 2: I have a faster rtable.
[3.097165] Agent 1: I have a faster rtable.
[3.101485] Agent 0: I have a faster rtable.
[3.103804] Agent 3: I have a faster rtable.
[3.104095] Agent 2: I have a faster rtable.
[3.105303] Agent 3: I have a faster rtable.
[3.105912] Agent 1: I have a faster rtable.
[3.114916] Agent 3: I have a faster rtable.
[6.024210] Agent 3: periodic routing process started.
[6.024210] Agent 3: Total sent 217, total received 212 pkts
[6.031235] Agent 0: periodic routing process started.
[6.031235] Agent 0: Total sent 217, total received 214 pkts
[6.046669] Agent 1: periodic routing process started.
[6.046669] Agent 1: Total sent 221, total received 217 pkts
[6.059547] Agent 2: periodic routing process started.
[6.059547] Agent 2: Total sent 219, total received 214 pkts
permutation space is 24
0 3 1 2
[6.072466] Agent 1: winner's fitness: 154.846
permutation space is 24

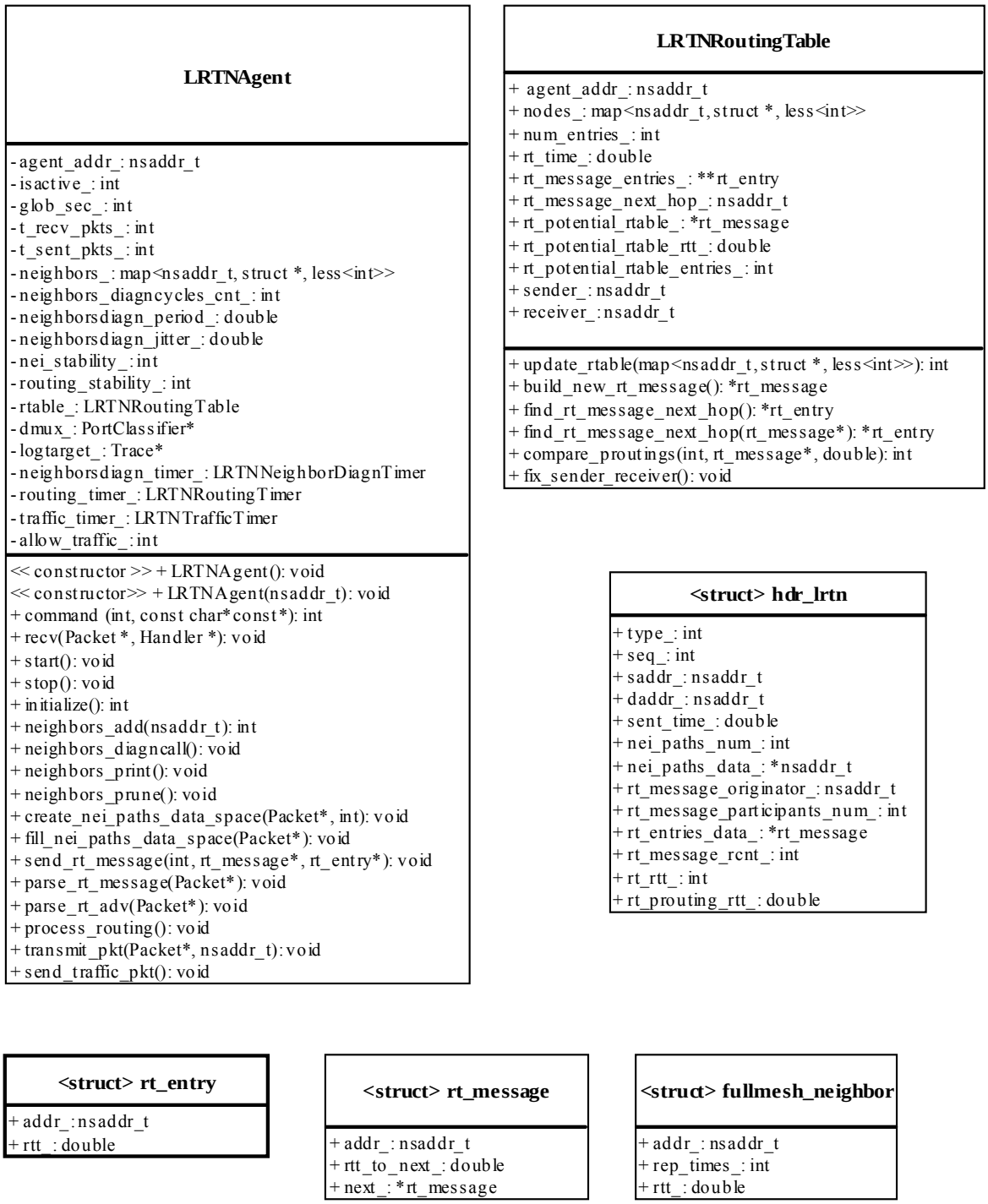
2 0 1 3
[6.087081] Agent 2: winner's fitness: 187.242
permutation space is 24
2 0 1 3
[6.091340] Agent 0: winner's fitness: 187.242
[6.091997] Agent 0: I found a better rtable.
[6.101906] Agent 0: I have a faster rtable.
[6.111686] Agent 3: I have a faster rtable.
permutation space is 24
2 0 1 3
[6.114061] Agent 3: winner's fitness: 187.242
[6.114773] Agent 1: I have a faster rtable.
[6.115862] Agent 3: I have a faster rtable.
[6.117884] Agent 2: I found a better rtable.
[6.117971] Agent 2: I have a faster rtable.
[6.123818] Agent 2: I have a faster rtable.
[6.125706] Agent 0: I have a faster rtable.
[6.131236] Agent 3: I found a better rtable.
[6.133791] Agent 3: I have a faster rtable.
[6.134296] Agent 0: I have a faster rtable.
[6.135990] Agent 2: I have a faster rtable.
[6.139508] Agent 1: I have a faster rtable.
[6.140663] Agent 1: I have a faster rtable.
[6.144045] Agent 3: I have a faster rtable.
[6.144446] Agent 0: I have a faster rtable.
[6.145537] Agent 1: I have a faster rtable.
[6.146053] Agent 1: I have a faster rtable.
[6.156987] Agent 1: I have a faster rtable.
[6.176245] Agent 2: I have a faster rtable.

Simulation time: 0s – 7s
Number of nodes: 4
Protocol: LRTN_GA
Constant delay: 100s – 180s (busy network)

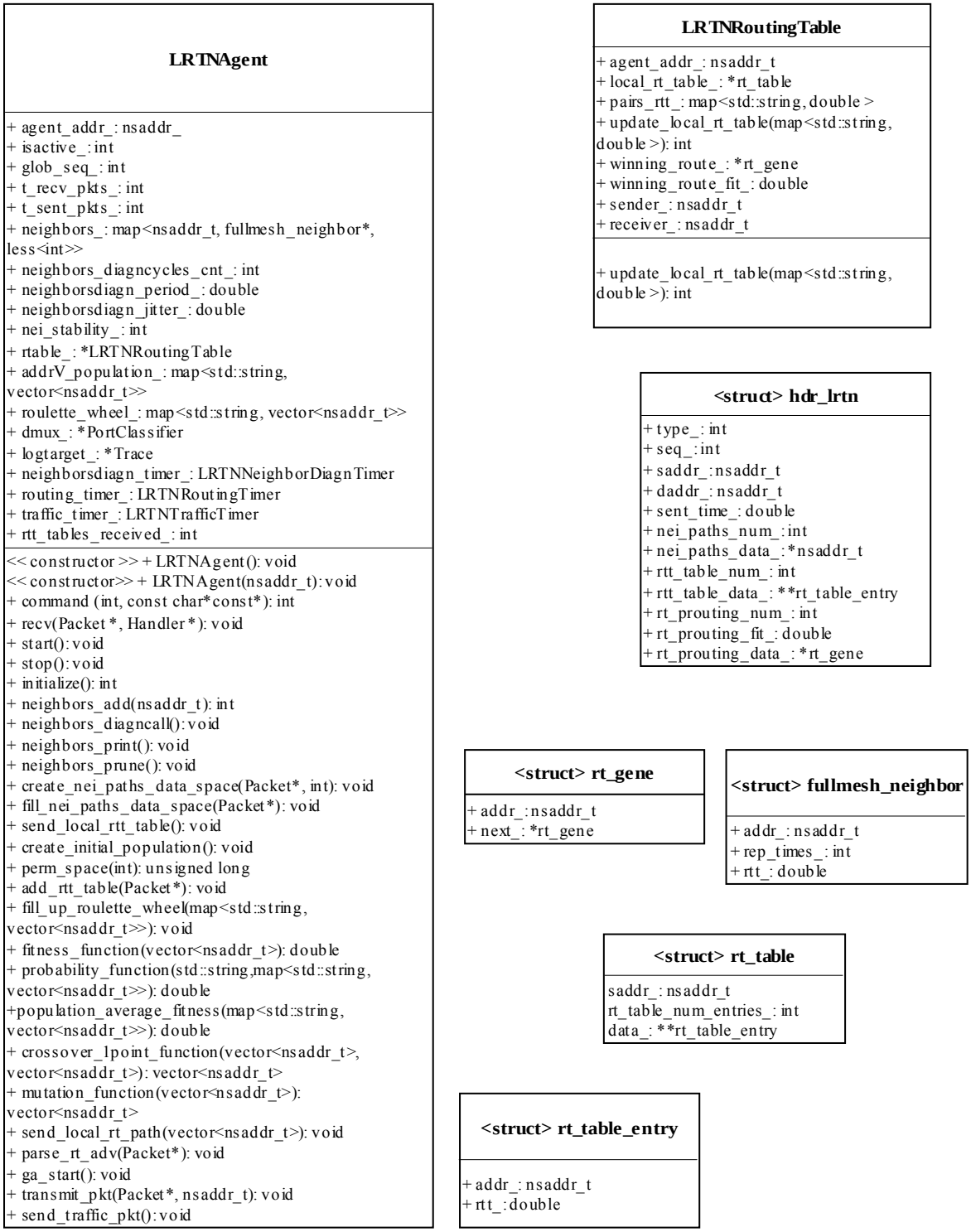
[0.010000] Agent 0 started.
[0.010000] Agent 1 started.
[0.010000] Agent 2 started.
[0.010000] Agent 3 started.
[0.015033] Agent 1: new neighbor added..
[0.038726] Agent 3: new neighbor added..
[0.043810] Agent 2: new neighbor added..
[3.010000] Agent 0: periodic routing process started.
[3.010000] Agent 0: Total sent 106, total received 102 pkts
[3.010000] Agent 1: periodic routing process started.
[3.010000] Agent 1: Total sent 105, total received 100 pkts
[3.010000] Agent 2: periodic routing process started.
[3.010000] Agent 2: Total sent 108, total received 102 pkts
[3.010000] Agent 3: periodic routing process started.
[3.010000] Agent 3: Total sent 108, total received 101 pkts
permutation space is 24
3 2 0 1
[3.075085] Agent 0: winner's fitness: 498.009
[3.084564] Agent 1: a new rtable came and mine is NULL.
[3.086473] Agent 2: a new rtable came and mine is NULL.
permutation space is 24
3 2 0 1
[3.122009] Agent 3: winner's fitness: 498.009
permutation space is 24
3 2 0 1
[3.135835] Agent 2: winner's fitness: 498.009
[3.137606] Agent 3: I have a faster rtable.
permutation space is 24
3 2 0 1
[3.139067] Agent 1: winner's fitness: 498.009
[3.150562] Agent 0: I have a faster rtable.
[3.157615] Agent 0: I have a faster rtable.
[3.170106] Agent 0: I have a faster rtable.
[3.217217] Agent 1: I have a faster rtable.
[3.225905] Agent 3: I have a faster rtable.
[3.234746] Agent 2: I have a faster rtable.
[3.239061] Agent 3: I have a faster rtable.
[3.260487] Agent 2: I have a faster rtable.
[3.260531] Agent 1: I have a faster rtable.
[6.027800] Agent 2: periodic routing process started.
[6.027800] Agent 2: Total sent 218, total received 211 pkts
[6.037038] Agent 1: periodic routing process started.
[6.037038] Agent 1: Total sent 216, total received 208 pkts
[6.049525] Agent 3: periodic routing process started.
[6.049525] Agent 3: Total sent 216, total received 210 pkts
[6.053390] Agent 0: periodic routing process started.
[6.053390] Agent 0: Total sent 215, total received 210 pkts
permutation space is 24
2 0 1 3
[6.078705] Agent 0: winner's fitness: 350.638
[6.095099] Agent 1: I have a faster rtable.

```
permutation space is 24
1 0 3 2
[6.130994] Agent 3: winner's fitness: 554.239
[6.132557] Agent 2: I have a faster rtable.
[6.134097] Agent 3: I found a better rtable.
[6.161511] Agent 0: I have a faster rtable.
permutation space is 24
1 0 3 2
[6.164571] Agent 1: winner's fitness: 554.239
permutation space is 24
1 0 3 2
[6.173020] Agent 2: winner's fitness: 554.239
[6.180450] Agent 0: I have a faster rtable.
[6.197317] Agent 0: I have a faster rtable.
[6.214111] Agent 0: I have a faster rtable.
[6.218525] Agent 1: I have a faster rtable.
[6.221923] Agent 1: I found a better rtable.
[6.247412] Agent 2: I have a faster rtable.
[6.258680] Agent 2: I found a better rtable.
[6.259662] Agent 0: I have a faster rtable.
[6.266544] Agent 2: I have a faster rtable.
[6.278810] Agent 3: I have a faster rtable.
[6.287955] Agent 3: I have a faster rtable.
[6.296334] Agent 1: I have a faster rtable.
[6.301633] Agent 0: I have a faster rtable.
[6.322193] Agent 2: I have a faster rtable.
[6.328555] Agent 3: I have a faster rtable.
[6.377656] Agent 1: I have a faster rtable.
[6.379702] Agent 3: I have a faster rtable.
```

Appendix V



UML Class diagrams for LRTN protocol



UML Class diagrams for LRTN_GA protocol