Understanding Dynamics of Complex Communication Networks

By

Adam Cornachione

COMMITTEE:

Dr. Brian Hay
Assistant Professor, Department of Computer Science
University of Alaska, Fairbanks

Dr. David Newman
Professor, Department of Physics
University of Alaska Fairbanks

Dr. Orion Lawlor
Assistant Professor, Department of Computer Science
University of Alaska, Fairbanks

Understanding Dynamics of Complex Communication Networks

A

Master's Project

Presented to the Department of Computer Science of the University of Alaska Fairbanks

Adam Cornachione
April, 2012

Understanding Dynamics of Complex Communication Networks

By

Adam Cornachione

RECOMMENDED:

_____

Dr. Brian Hay
Assistant Professor, Department of Computer Science
University of Alaska, Fairbanks

_____

Dr. David Newman
Professor, Department of Physics
University of Alaska Fairbanks

_____

Dr. Orion Lawlor
Assistant Professor, Department of Computer Science
University of Alaska, Fairbanks

_____

Date

TABLE OF CONTENTS

LIST OF FIGURES

Abstract

Society is becoming increasingly dependent on large, complex networks to provide a means for the exchange of information. Because of this increase in dependence, the failure of such systems is increasingly hazardous. In order to better understand the dynamics of such systems, a good model of the system is required. Such a model should not only accurately depict the structure of a system, but also realistic dynamics. The primary objective of this project is to provide a model of communication networks depicting accurate representation of end hosts and communication protocols, along with validated dynamic behavior. These networks, acting autonomously, must also interact with one another. The networks depend on this inter connective web to successfully operate, thus failures in one network may have a hazardous effect on the entire system. By analyzing the behavior of the said model, one can explore the intricacies of complex communication networks and work towards ultimately building more reliable networks.

I. NATURE OF COMPLEX SYSTEMS AND COMPUTER NETWORKS

1.0 Introduction

Today's society has become increasingly dependent on large communication systems, such as the Internet. Because of this dependence, the vulnerability of these systems is a vulnerability of society as a whole. These systems are often forced to run near their operational limit due to the combined impact of hardware limitations on network capacity, society increasing the amount of information that needs to be transferred and processed, and the high level of complexity in the systems resulting from the size and various protocols. By running near their operation limits, the systems are left susceptible to cascading failures, [1] and the dynamics of the systems are difficult to predict. The purpose for this project is to provide a model to explore the intricacies of complex communication networks and provide insight on constructing more reliable systems. The model should give an accurate representation of the dynamical behavior of highly congested networks.

1.1 Complex Systems

Complex Systems have a wide variety of applicable definitions. Essentially a complex system has total system behavior not equal to the sum of the individual pieces. Another way to think of this is a system with output which is not obvious from either the input into the system or the parts from which the system is composed. Although complex systems are each uniquely composed of different parts, their complexity is often defined by their dynamic behavior. Complex systems exhibit examples of non-linear dynamics. Power law tails are one way of exhibiting such nonlinear behavior. A power law is a situation where some quantity can be expressed as a power of another quantity. For example, in the below equation, the quantity P can be expressed as a power t of the quantity s. [2]

P(s) = s^t

Where *t* is the power. This type of relationship will show a straight line when graphed on a double-logarithmic plot.

log P(s) = log s^t

log P(s) = t * log s

The adjusted equation will show a straight line on a double-log plot, with slope *t*.

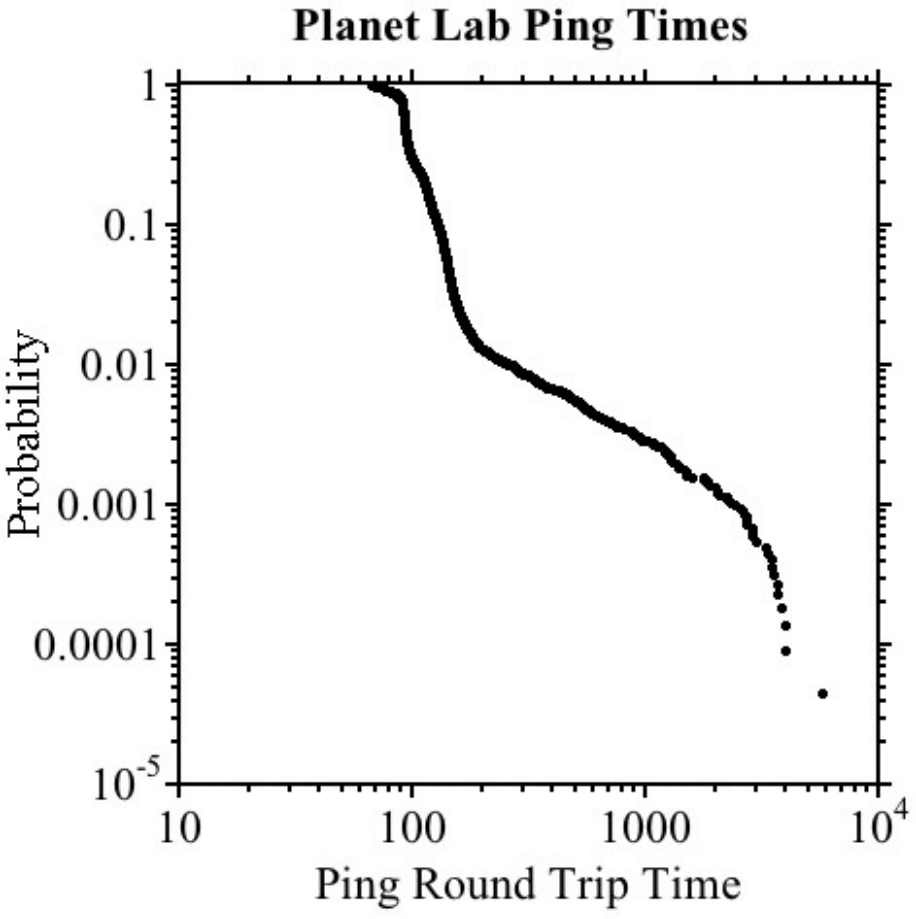


Fig. 1 Power Law Tail in Ping-Times

Ping times are a measurement of the time it takes one "packet" of information to be sent from one host to another and back again. Every ping, traveling the same distance between two hosts, may take a different amount of time to return. These pings are a good representation of the speed of communication between two end hosts in a network and will be the primary source for validation of dynamic behavior. Some pings come back quite fast, and others take much longer. The probability distribution function (PDF) of the return times for each ping is a plot of the frequency, or occurrence of a ping return time, against the return time. Fig 1. above shows a PDF of ping times of two hosts across the Internet. The strong linear correlation in the probability of the ping times starting just after 100 seconds and continuing to around 2000 seconds on the double logarithmic plot shows dynamic behavior characteristic of complex systems. [1] [2] This correlation between the ping return times and their frequency is important in the validation of the model to be constructed. It is necessary for this model to capture dynamic behavior which reflects behavior seen in real networks.

Purpose of study

There is a pressing need to understand the behavior of complex systems. Society is dependent on many such systems including the US Economy, power transmission grids, and the internet. Because of our dependence on such systems, any vulnerability in these systems is a vulnerability of society as a whole; therefore there exists an increasing importance in improving their reliability. Social and economic pressures often push these systems to run near their operational limits, which leaves such systems vulnerable to cascading failures. Large-scale failures of our communication networks can have huge implications on national security, human health and safety, not to mention personal communication.

The behavior of these systems, however, can be difficult to predict, and each system operates in a different manner. For example the Internet follows a completely different structure design than the Power Transmission Grid, and will likely operate differently when pushed near its operational limit. Because of the variations of such systems, each will need a model which captures the specific behavior of each system. [1] It is important for this model to capture the structure and protocols unique to communication networks along with complex dynamic behavior. In order to be useful towards the study of such networks, simulations will need not only be an accurate representation of the said system, but will need to scale well enough to run for long time periods to simulate the evolution of networks, while capturing the dynamic behavior.

1.2 Introduction to Complex Communication Networks

Defining a large communication network

A large communication network, as defined for this project, refers to an interconnected computer system. This could be as simple as a single client to server connection, expanding to a larger local area network or as large and unruly as the entire Internet. This tool will need to be able to accurately model a wide variety of networks differing not only in size but also in structural characteristics.

The Internet is a prime example of a large communication network used extensively in our society. There are other large scale networks, but the Internet provides an excellent example of why the failures of such systems can be so problematic. If the Internet as a whole were to go down, many of our fast, effective ways of communicating would be inaccessible, as alternative methods of communication are much slower, or severely limit the amount of information which can be exchanged. Reliability in our large communication networks is a matter of national

security, thus a need for more resilient and reliable networks is important. Although the failure of the entire Internet may seem unlikely, failures of smaller portions happen frequently, having the mentioned effect on a smaller area.

There are many characteristics common to the Internet in contrast to similarly structured infrastructure systems. The speed and manner in which the Internet grows is different from the power transmission grid, although both are structurally similar. Humans interact with the internet in many different ways. Network traffic consists of business transactions, media streaming, direct communication and many other exchanges. Network dynamics therefore affect numerous other infrastructure systems, such as the economy with business information trade or the utilization of electricity from the power transmission grid. This coupling to so many other systems creates an even more pressing need to keep a certain level of reliability in the communication network.

1.3 Related Work

There is much research being conducted in complex systems for a variety of reasons. It's a relatively young field, and there is a considerable amount not yet understood concerning many of the characteristics and dynamic behavior of such systems. Fueling this research is the increasing dependence on these unpredictable systems and the growing need to improve their reliability.

Since today's society is built around many large infrastructure systems, a lot of research is being conducted to study the dynamic behavior of such systems. [4] Since computers can aid in creating models to study such systems, there are many projects in creating robust simulators. [12] One of the most directly related studies is that of cascading failures. Many of these systems may be pushed to sit near their threshold [1], which leaves them susceptible to failures resulting

from small localized incidents, which can snowball and collapse into failures of all scales, up to entire system collapse. [2] Research in cascading failures studies how best to construct a system, focusing on aspects such as structural design and materials used in construction to limit both the magnitude of cascading events, and the frequency. [1]

Open source projects used to design computer simulators are also very relevant work. Because of the amount of research done with predictive models, people are trying to create the best models they can to capture real-world dynamics.[3][6][7][12] This project involves extending the Objective Modular Network Testbed in C++ (OMNET), [3] an open sourced network modeling simulator. The OMNET project provides the public with a tool for creating arbitrary network simulators. The reasons for choosing OMNET for this project tool are outlined in Section 2.0.

Because communication networks specifically are so crucial to society, much research in creating more resilient and more efficient networks is called upon. Good predictive models can be used in learning more about the dynamics of systems, and are often a good place to learn about the effects of new structure, design and communication protocols on dynamic behavior. They are more cost effective and far less time consuming than creating physical models to test. Projects such as the Global Environment for Network Innovations (GENI) Project are an example to how important the research of large communication networks is to society. GENI is a very large project promoting realistic and valuable network research. [4]

1.5 Project Requirements/Scope

This project is concerned with constructing an environment suitable for studying the dynamic behavior of various communication networks. These networks may vary in size from small, local networks, to large-scale models of the internet. The model must capture the non-

linear dynamic behavior characteristic of complex systems, and will be validated against real data taken from real networks. It must imitate real protocols used in communication today. Simulations, in order to capture the evolution of a system, will need to scale well enough to run much faster than real time. The last issue is to make the tool easy to use. There will need to be an easy way to create networks and perform various simulations on them.

Functional Requirements

- Working network modeling tool validated against real network data
    - Must display non-linear dynamics (power law in ping times)
    - Should imitate real network structure and communication protocols
- Model must include dynamic routing capabilities
    - Necessary for simulating network growth and response to node failure
- Must scale well to simulate high levels of congestion
- Need a way to quickly create networks and set up simulations

Non-Functional Requirements

- Model should be easy to use
- Limited amount of training to create and run various simulations
- Will require a simple method for creating networks and running simulations

II. PROJECT APPROACH AND METHODOLOGY

2.0 Approach Summary

This project addresses the need to better understand communication systems and computer networks using simulation techniques. The primary objective of the project is to develop a working model which simulates the dynamic behavior of various communication systems. Specific project tasks include:

1. Select existing modeling tool(s) to create initial network model
2. Develop a series of experiments which will be used to develop and test the model
3. Evaluate the performance of the model against identified criteria (requirements)
4. Provide recommendations on
    - Model effectiveness
    - Areas for improvement
    - Future work based on project results

2.1 Introduction to OMNET++

What is OMNET/INET

Objective Modulare Network Testbed in C++ (OMNET) [3] is a popular open source network modeling tool, making it a good choice to build the initial network model. OMNET is a simulation library written in C++, designed specifically for building network simulators. It works by designing modules, which all communicate by passing messages between one another. It is a discrete event simulator, using a heap as its main data structure for controlling the timing of message passing between objects. These features provide easy ways of designing modules used to represent the components of a large-scale communication network. OMNET was the

first tool chosen for implementation of the network simulator, and ultimately through tests and modifications remained the main tool.

The INET framework [6] is a library of OMNET modules which are designed for networking protocols. INET has modules designed for every layer of the Open Systems Interconnection model (OSI). OMNET modules representing hosts, routers, transmission control protocol (TCP) and all the basic requirements for a communication network are defined by INET. This doesn't mean that the work is already done; current OMNET models and simulations fall short when constructing the discussed model: (1) OMNET could not process enough traffic to fully congest large networks faster than real time, (2) the INET framework alone does not provide dynamic routing capabilities, (3) There is no current method to automate the construction of initialization files. These issues are handled in Section 3 as the development process is broken down into experiments used for testing and improving the features of OMNET and validating the results.

Reasons for choice of OMNET

1. Discrete event simulators are typically easy to work with and can be powerful tools for running simulations.
2. OMNET is open source and freely available for academic use, and is well documented and maintained.
3. OMNET can run on many environments, spanning Linux and Windows machines.
4. It is easily modified and users can tailor the program, adding their own features and modules, according to their needs.

Along with all of these favorable features of OMNET, the INET framework [6] already has many of the objects and protocols for communication networks designed and implemented.

INET provides enough features to span most of the scope of this project, and satisfies the functional requirement for using real communication protocols. INET is not complete, however, and lacks a few features necessary to satisfy all of the project requirements. INET only provides ways to construct completely static networks, which doesn't allow any growth of the network, or much capabilities of dynamic response to congestion. Issues like these will need to be handled through a testing phase in which modifications to the framework will be necessary.

2.2 Experimental Methodology

After the main development tool, OMNET, was chosen, there were six main steps in creating this model. At each step one or more tests were performed, the results of which were analyzed to provide information on how to proceed in order to satisfy all of the functional requirements.

Exp. 1: This experiment involves installing OMNET and INET and running simple network simulations. There is expected to be a learning curve when implementing OMNET and INET. OMNET is a large program with many features, possibly not all of which will need to be utilized. Open source projects often come with little documentation, so the first experiment is to get both OMNET and the INET framework installed and run very basic network simulations. The simulations, because of their simplicity, are expected to run smoothly with no obvious bugs or issues at this point.

Exp. 2: One of the issues most likely to be faced is that of scalability. This experiment is designed to test the scalability of OMNET in terms of both size of network and amount of traffic. Simulators have limits to the amount of information which can quickly be processed. It is likely that there will be some scaling issues with OMNET and INET because many of the simulations for testing the complex dynamics of networks will force the networks to their threshold. Thus

the high amount of traffic expected to be handled by OMNET might take a hit on its performance to the point that either modifications will be necessary, or a different tool may have to be used.

Exp. 3: This experiment aims to identify nonlinear dynamics in OMNET network simulations. The simulations must capture the important dynamic behavior for studying their complexity. Ping tests will be used to look for power law distributions in the network. If INET correctly implemented the networking protocols, then as long as enough traffic can be produced, the network model should reflect real network dynamics. Experiment 3 will also provide a way to strip down the network and examine a few areas where feedback in the system can lead to nonlinear dynamics during congested periods.

Exp. 4: This experiment was set up to test the dynamic routing capabilities of INET-Quagga [7]. INET does not alone provide dynamic routing in network simulations. INET-Quagga is a framework used to provide dynamic routing capabilities to INET simulations. The dynamic routing is expected to provide response to congestion. That is, if a router becomes overly congested and drops most of the incoming traffic, or just fails outright, then new routes will be established. The expectation was that the implementation of INET-Quagga would effectively provide the necessary dynamic routing capabilities for the network simulator.

Exp. 5: This experiment is concerned with the construction a network model representing a real-world network, specifically the UAA Network. Implementing and testing a real world network provides another way to validate the network model. It can help determine how well the simulations can reflect reality and give a feel for the limitations of the tool. Although implementation of the network will be by hand, it is expected that it is straightforward to design and simulate an OMNET network emulating one from the real world.

Exp. 6: This experiment will test how network responds to dynamical changes in simulation. The final deliverable will be a network simulator capable of capturing realistic network dynamics and responding to network congestion. The expectation is that the dynamic routing protocols will discover new routes when routers either undergo high congestion or node failure. With this stage completed, a network modeling simulator will be ready for more advanced testing and ultimately use in the study of the dynamic behavior of complex communication networks.

III OMNET EXPERIMENTS AND RESULTS

3.0 Exp. 1: Working with Simple OMNET Programs

Constructing Modules and Networks

There are four main steps in creating network models, running simulations and processing results in OMNET.

1. OMNET++ models are built from less complex modules, or components, which communicate by exchanging messages. The modules can be nested, or grouped to form compound modules. Networks are created from connecting various modules and compound modules together. Fig 2 below shows how various modules can be nested to create a Router. The Router module is part of the INET Framework.

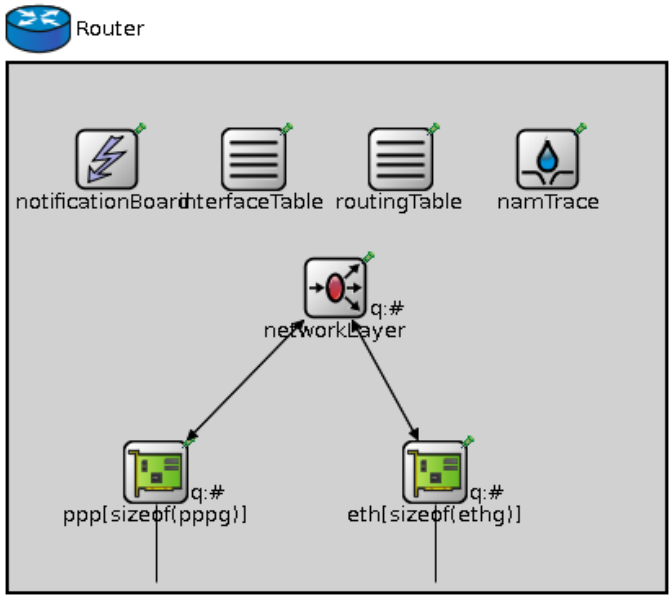


Fig 2 Instance of a Router Module from the INET framework.

2. The structure of networks are defined in the NED language. OMNET provides a GUI which can be used to graphically create networks by clicking and dragging

modules onto the network and connecting them to one another. The Fig 3 below shows a simple network created in OMNET using the INET modules for Routers and StandardHosts.

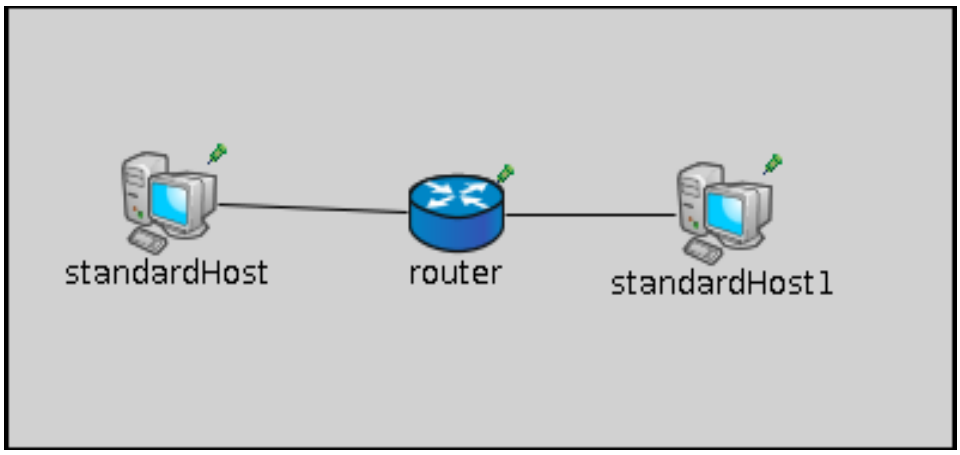


Fig. 3 A small network created in OMNET using INET modules Router and Standard Host

3. The actions of each module are programmed in C++. When OMNET simulations are run, each model in the network must first be initialized. The initialization phase of the simulation is where the simulation parameters for each module are initialized before the network communication and traffic flow actually begins. For every simulation, and initialization file "OMNET.ini" must be created to define the various parameters. Appendix A contains an example initialization file used to run the first simulation of the network in Fig. 3
4. Once an initialization file is created, simulations can be run. They can be run via either command line with only text output, or they can be run with a GUI in which network behavior can be observed throughout the simulation. Results are

recorded into output vector and output scalar files. The results are all text base and can be viewed with a variety of tools. The results for this project are processed with Kaledigraph. [11]

Running First Simulation

INET has modules and definitions for several communication protocols including TCP, UDP and a "PingApp." The INET PingApp is a module which the hosts use to send pings to one another. Initial simulations used the PingApp for communication between end hosts. Ping tests should be sufficient for testing whether or not the simulations of INET networks can scale well enough to display complex dynamics. The ping return times between hosts will be recorded in the output vector file for processing. The goal of this first test was to get familiarized with the OMNET tool and get a feel for what some of its limitations might be. The expected result is that the INET StandardHost modules will be able to send ping messages back and forth between a central INET Router.

Exp 1 Results

Initial conclusions and concerns

As was expected, there were no obvious bugs in the simple OMNET simulations. The INET "flatNetworkConfigurator" module assigned an IP address for every interface in the network, so the Router could route between hosts. The network is displayed in Fig. 4 below.

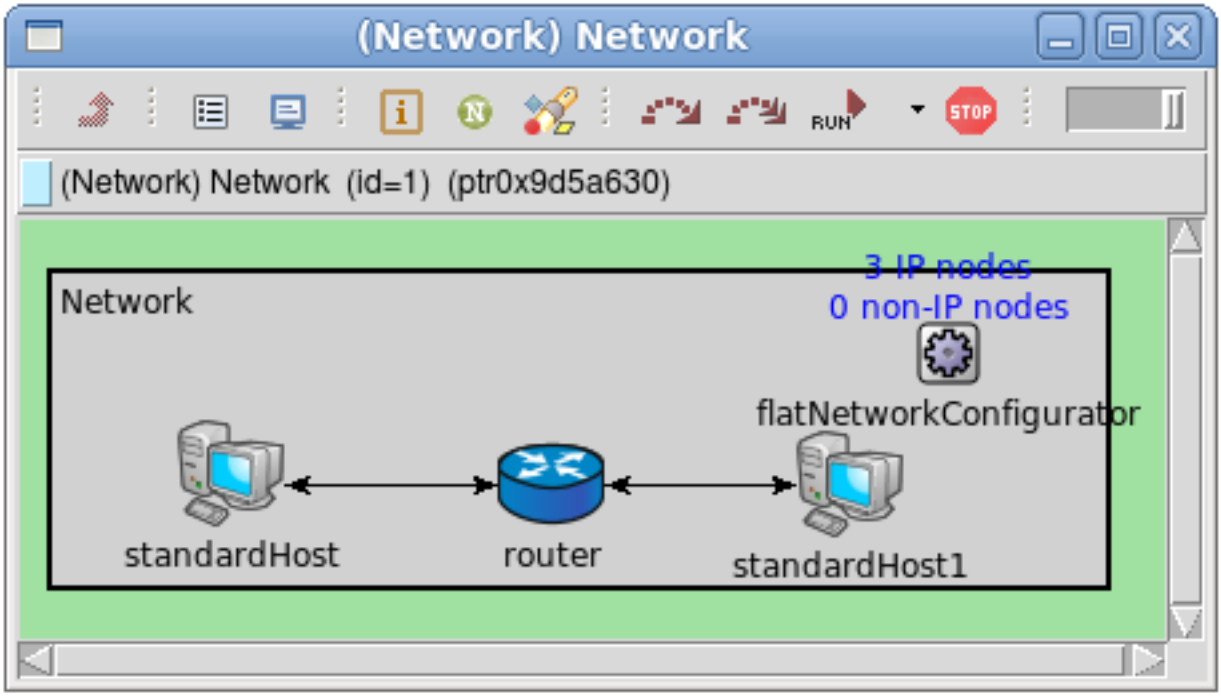
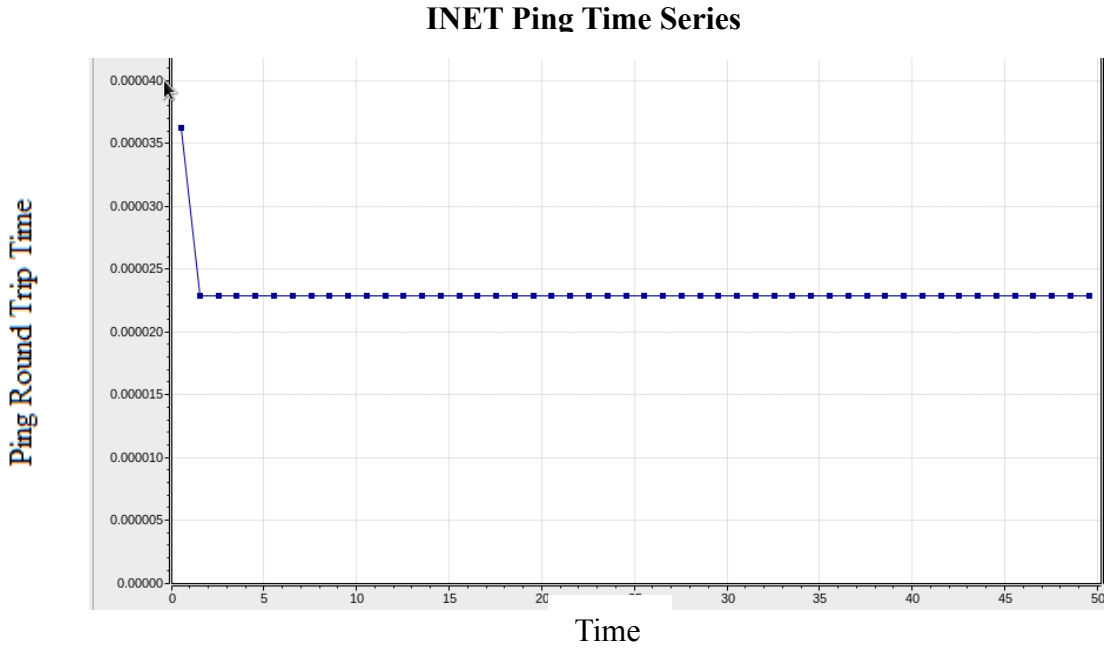


Fig. 4 The Network used in experiment one

The INET "pingApp" sent a ping to the other host every second, which the router was able to transport both ways successfully. Fig. 5 shows the initial results for the ping-times between the hosts.



Time

Fig. 5 Ping times between standardHost and standardHost1 from network in Fig. 4.

The ping times from the simulation showed almost no variance whatsoever. Almost every ping had the same return time. For the first experiment, though, it was a success to get a simulation up and running. Larger and more complicated networks would follow. Although this small network had no obvious problems, a few important questions and concerns arose.

First of all, can OMNET handle a large network? One of the functional requirements for this modeling tool is to provide the user with a way to create various sizes of networks. Specifically anywhere from a few nodes to the order of a few hundred, and possibly thousands of router and end hosts is desired. OMNET was able to run a small network just fine, but it must be able to handle large networks of various sizes.

Secondly, can it handle large amounts of traffic? Along with creating large networks, this modeling tool will have to be able to scale well enough to provide enough traffic interaction to push the network near its operational limit. This feature is necessary for providing a network model which is able to capture complex dynamics because one of the main causes of complexity in networks is the congestion. Providing OMNET with a way to scale to the levels required for this model turned out to be a large portion of the project.

ReaSE

The last main concern with the first experiment was that there might not be a way to create networks in OMNET quickly. The small network in this experiment was created quickly by hand, but its only three nodes. There needs to be a way to create large networks quickly. If network creation is the bottleneck, the simulator won't be too useful. Fortunately there is a tool called ReaSe [10] which provides a way to create large networks quickly. ReaSE automatically creates networks by connecting INET Router and StandardHost modules to one another. ReaSE also comes with a graphical user interface, as shown in Fig 6.

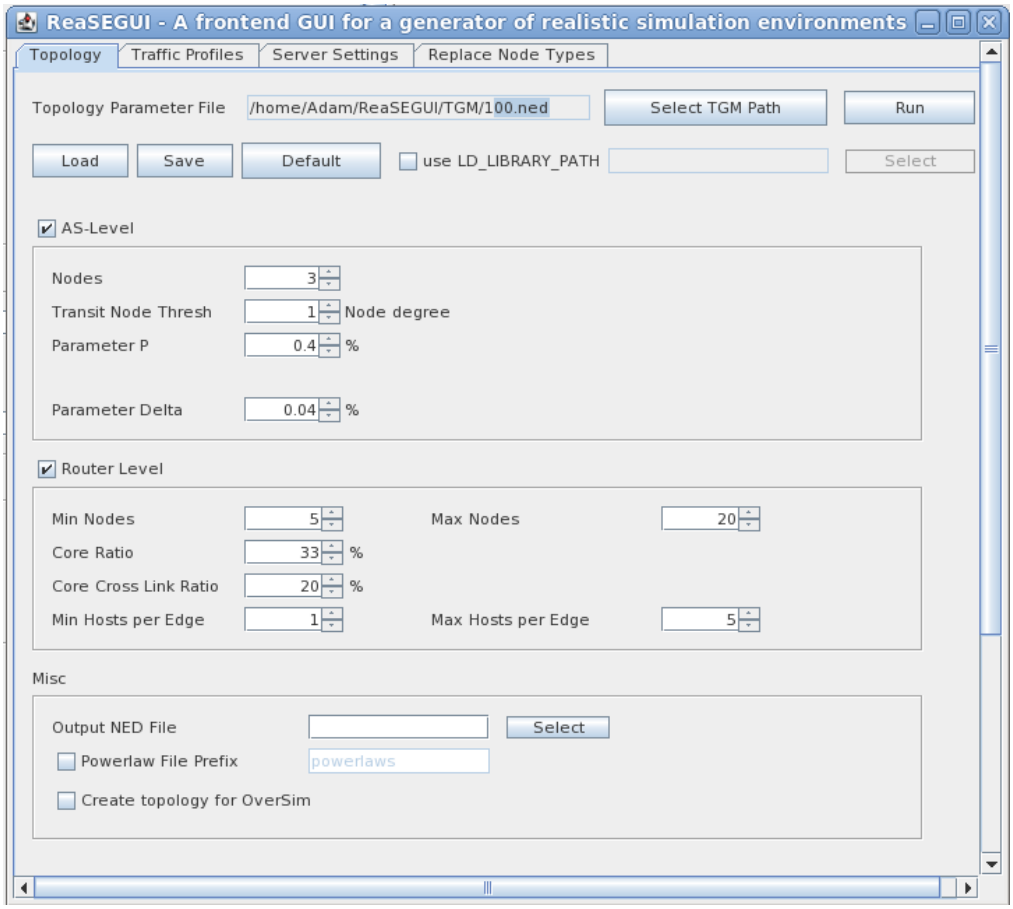


Fig. 6 ReaSE GUI

ReaSE can create many different networks based on a few parameters. It assigns each router a type depending on whether it is a core, gateway or edge router. The primary difference in each router is the bandwidth in the connections between interfaces. The routers in ReaSE can be set to represent single routers or autonomous systems (AS). These autonomous systems are just INET routers designed to represent the whole AS. Autonomous Systems are groups of routers or networks controlled by the same network administrator. [9] Traffic is routed between autonomous systems in a similar manner to the way traffic is routed between routers. Because the job of an AS is the same as a router, just on a different scale, using routers to represent entire autonomous systems will be sufficient for this project. Since ReaSE can create all the different sized networks with various characteristics, for instance the ratio of core to edge routers, needed

by this project using the INET modules, it will be used in the final model as the network generator. The rest of the process, however, still needs to be automated.

ReaSE does not automatically create the necessary initialization files in order to run simulations on the networks it creates. This step will need to be addressed when creating an automated process. ReaSE will output a network description file, the user will then run an automation script which will create the OMNET initialization file with parameters entered by the user. The final automation script and process is shown and outlined in Appendix E.

3.1 Exp. 2: Large Networks in OMNET

Testing OMNET Scalability

ReaSE is able to generate networks at the AS (autonomous system) level and the router level. AS's can be sufficiently simulated by an INET Router module. ReaSe was used to create the Router structure in the various sized networks for experiment 2, and then various StandardHost modules were connected with selected Routers to perform Ping Time tests. A display of part of the larger networks is shown in Fig 7 below:

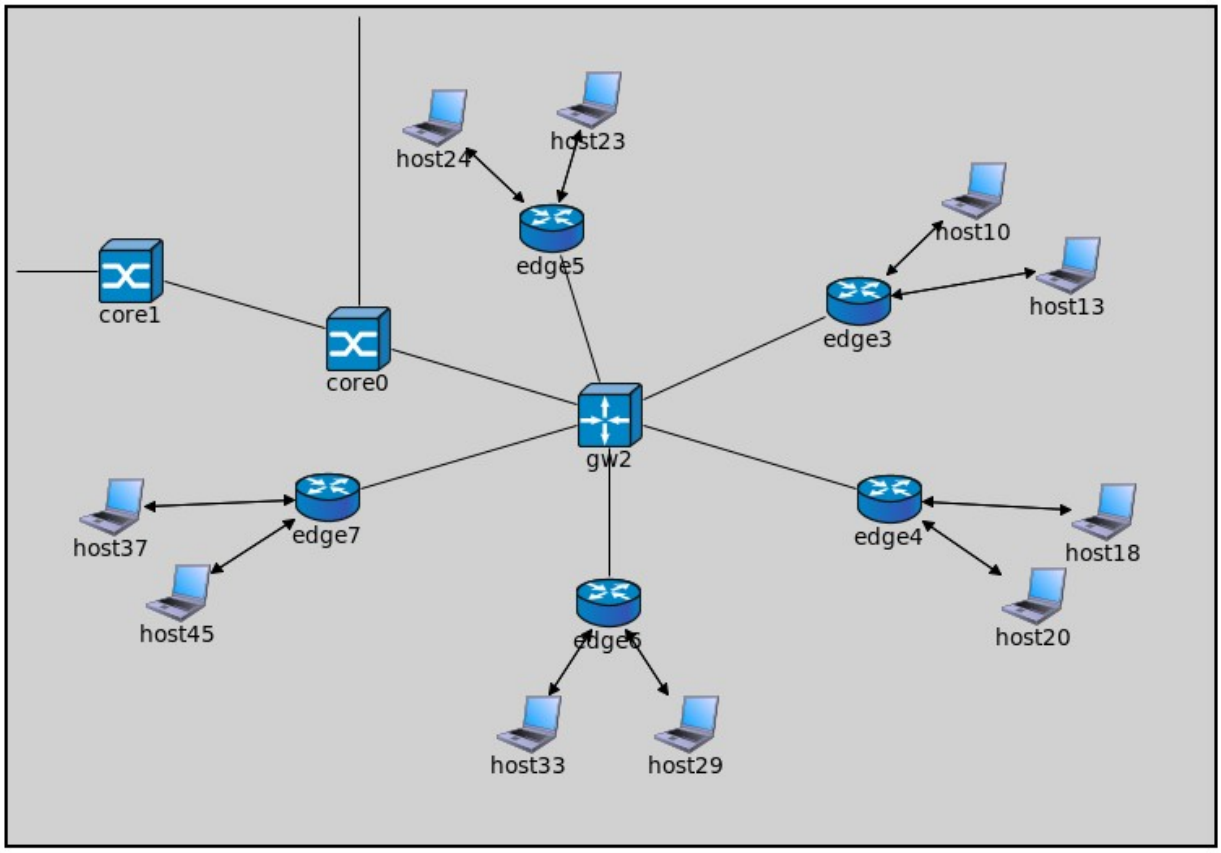


Fig 7 A small portion of the very large networks created with ReaSE

This second experiment was used to test whether OMNET could handle these large networks created by the ReaSE tool. Another functional requirement for the tool is that it must scale well both in the size of such networks and in the amount of traffic. This issue of scalability is one of the main problems of predictive modeling. In order to predict the growth and dynamical behavior of a network, the simulation must run much faster than real-time. The INET PingApp was again used as the traffic generator, congesting the network with communications between end hosts.

Exp. 2 Results

Size Scalability

OMNET handled the large networks with few problems. The initialization phase of each simulation took longer because of the large increase in the number of modules. The following

table shows the time for the initialization of each network and the time it took to run the ping tests (for 12490 simulated seconds).

	Initialization Time (s)	Simulation Time (s)	SimTime-Init (s)
200 Routers	15	238	223
400 Routers	33	335	302
800 Routers	50	418	368

The increase in time spent initializing the large networks should not become the bottleneck. The longer the simulations run for, the less relative time is spent initializing the network, even for the very large networks. Since the simulations are ultimately expected to be run much longer, on the order of hours at a time, the time spent initializing the large networks will become negligible. The simulation time was not noticeably affected by the increase in the orders of magnitude of the size of the network. The larger networks took a bit longer to process the increased amounts of traffic, as there were more hosts on each network sending out pings every second. These results suggest that the time to handle the network traffic will be where the bulk of the processing is done. The easy handling of a large number of nodes, however, is a favorable feature of OMNET because ultimately this modeling tool will have to handle networks of such size.

OMNET Limitations

When trying to get any sort of interaction between packets, there was no way to increase the network traffic with the current traffic generators of INET to the levels required to see any nonlinear dynamic behavior. Initially pings sent across the network never had any chance of being delayed or held up by any others. By increasing the amounts of traffic needed for any feedback to occur in the system, the simulations were running far too slow to be useful, running at around one one hundredth of real time. Predictive modeling tools try to capture the dynamic

behavior of modeled systems much faster than real time, so as to predict how a real system will react to certain events. INET traffic simulations as they stand now are just too slow.

OMNET simulations have a few problems preventing simple construction of a network model suitable for the study mentioned of exploring the dynamics of complex systems. The main issue, as previously mentioned, is that the simulations do not scale well enough to capture the nonlinear behavior faster than real time. In order to construct a valid model, there needs to be a way to work around this problem. The current traffic simulators clog the network simulations, although a way of providing background traffic to represent the social pressures which drive the system toward complex behavior is an essential part of the model.

3.2 Exp. 3: Providing Background Noise

Queues

Each interface in every router has an output queue. Messages get stored in such queues during congestion to allow packets a place to wait until the line is free and they can be sent on to their destination. After some deliberation, it was decided that these queues is a good level to control the background traffic. Queues are used in real networks and are an important cause of delay. Routers can only process packets so quickly, and the lines have limitations in bandwidth, so queues are placed in each interface to store traffic waiting to be put on a line. During congested times, the average queue length for each interface would be higher than during times of low congestion, leaving packets waiting longer at each hop to be put on their respective line. The idea was to create an OMNET module which will control the level of congestion by affecting the number of packets in the output queues at any given time.

Fake Packets

The system needs a way to simulate enough traffic to the system's threshold and also run at a reasonable rate. The initial experiment that we tried to accomplish this feat was to create a varying number of "FAKE" packets to reside in the system. These packets would have no destination, and would not be passed between routers. All they would do would sit inside the each output queue in every interface of all the routers, keeping the system at a certain level with respect to its threshold. Real traffic between hosts could then be run on a system which is already congested. The fake traffic will exist only to interfere with real traffic run on top of the system, by causing a delay in the time each real packet is processed by the router. This setup should provide a way to simulate a congested system and still run in a reasonable amount of time, because most of the packets will not be doing anything active, and only slow down the real packets with real destinations.

The implementation of the fake packets involved modifying the INET source code of the interface modules. Essentially, the fake packets would not be passed outside of the interface they exist within. The interface will take fake packets out of the queues, delete them, and then wait a given amount of time to take another packet from the queue until the real packet is sent on the line. In this manner, the routers do not really spend much time handling the fake traffic, but the real traffic is slowed down, simulating a congested network.

Coupling to a "Complex" Model

A control for the amount of background traffic is required, and will come in the form of a "Complex" model coupled to the OMNET network model. The "Complex" model generated complex time series, which are time series that will periodically push the congestion level from low congestion all the way past the system threshold. In order for these time series to "talk" to the OMNET model, a new OMNET module needed to be created to act as the communication

interface. Fig. 8 shows a flowchart of the communication between the time-series and the OMNET simulations.

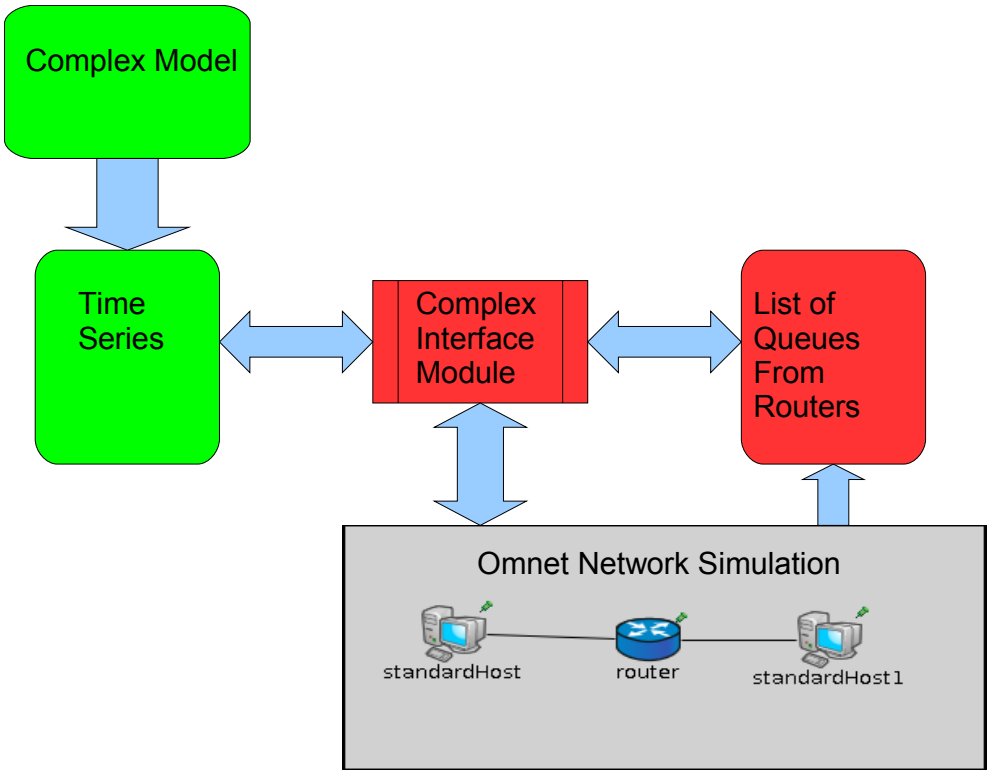


Fig. 8 A Flochart representing the use of the ComplexInterface Module to communicate beween the complex time series and an OMNET simulation.

The OMNET module "ComplexInterface" is the interface module between the time series and the OMNET simulation. It works by reading in the time series produced from the complex model, and stores it in a table. The time series is then converted into a number representative of the congestion level, and fed into the OMNET simulation by extracting each output queue and controlling the proper number of fake packets. The congestion level is thus controlled by the complex model.

The fake packets congest the output queues where the real packets, such as the pings, wait for their turn to be put out on the line. The queues are each filled with a certain number of Fake packets, as determined by the Complex model, and this provides background traffic. The Fake packets sit in the queues until a real packet comes along, and then each in turn exit until the real packet is put out on the line. Thus the real packet is slowed down in accordance with the number of fake packets it had to wait for at each hop. This way, the real packets must interact with a congested system, but the fake packets don't slow down the simulation as much as trying to scale congestion with only real packets.

This third experiment was set up to test the dynamic behavior of networks congested using fake packets placed in the output queues. In order for this simulation to be run, various INET models needed to be compatible with the fake packets. Appendix B on page X shows the source code of the modules which were changed in order to properly handle fake packets. Experiment three ran ping times between hosts on various networks with different levels of congestion. The results were analyzed and compared to data from real networks obtained from PlanetLab [5]. PlanetLab is a research network of hundreds of nodes across the world, and have provided real ping times from these hosts (see Fig. 1).

Exp. 3 Results

Outliers

The initial results appeared to scale well. Ping Times were coming in clearly interacting with the base congestion level at a very reasonable rate. This was a good sign, but the concern was that there were no signs of complex dynamics. The results had a very normal distribution when looking at the PDF in Fig 9.

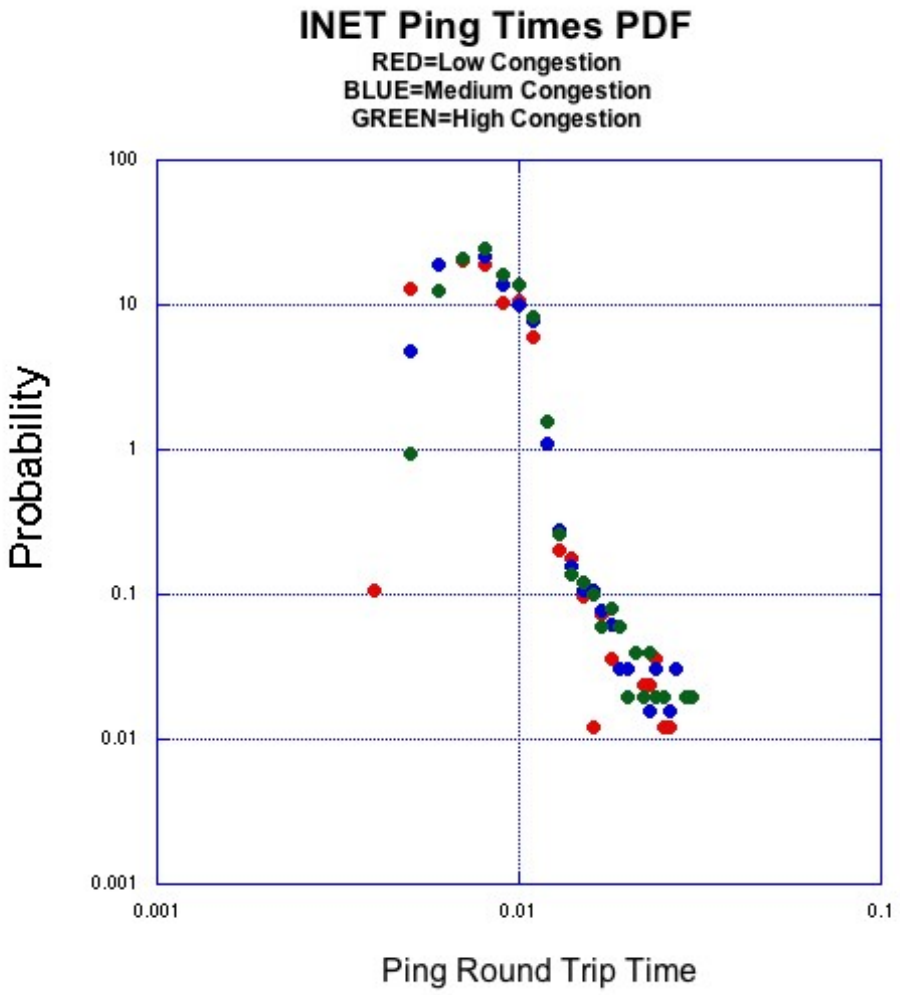


Fig. 9 Probability Distribution Function of Ping Times from Experiment Three at different levels of Congestion.

This distribution in the frequency of ping times for the simulation is very different from the distribution found in the real data gathered from PlanetLab shown in Fig. 1. Regardless of the level of congestion, the frequency of the ping times continued to show almost a normal distribution. One of the strangest results from this experiment was initially thought to be a bunch of outliers in the ping times. Fig. 10 shows a small portion of the ping times not as a PDF, but just a plot of the ping number on the x-axis and the return time on the y.

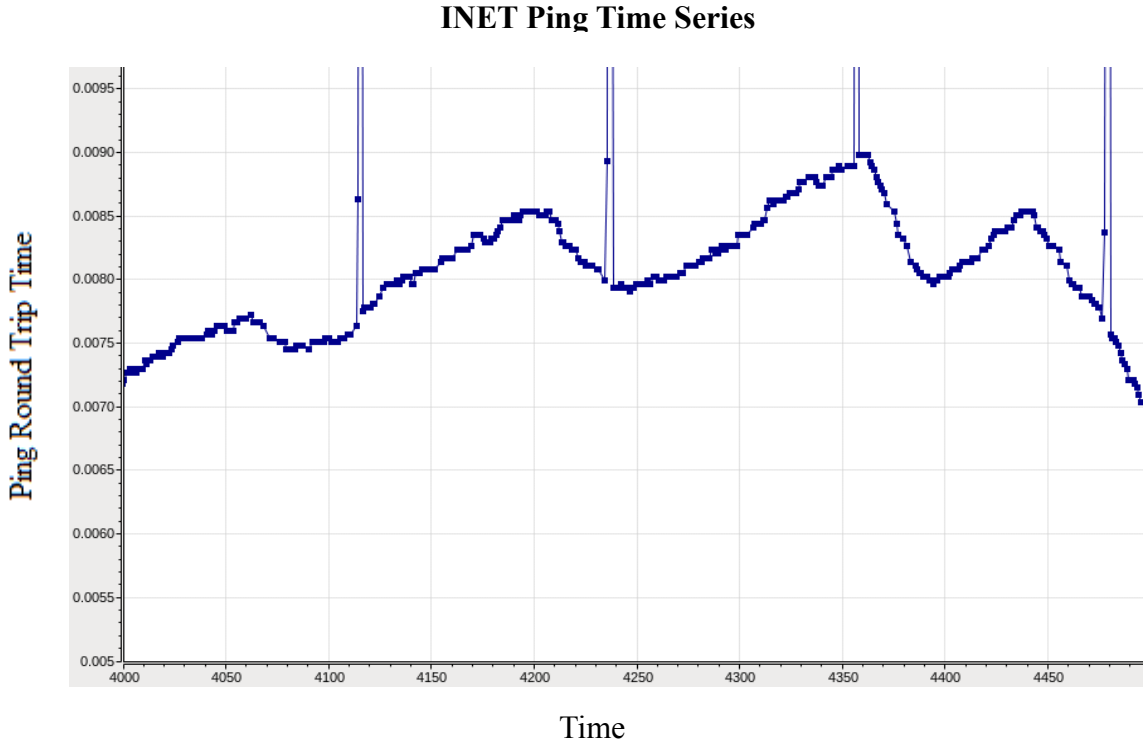


Fig 10 Chart of Ping Times between two hosts in OMNET Network. The off-the chart outliers were removed from the PDF in Fig. 7

Most of the ping times returned in a few milliseconds, but a small percentage returned on the order of seconds. The peaks and valleys in the plot represent the overall congestion in the system fluctuating. During periods of high congestion, the ping times increased on average, and during relaxed congestion the average ping times dropped. The few ping times coming in off the chart were deemed outliers. This huge disparity between pings was initially thought to be a programming error, so the outliers were thrown out when making the plots in Fig. 7. However, the times turned out to be part of the feedback which creates nonlinear dynamics.

ARP and rerun data

ARP (address resolution protocol) is a low-level protocol in the OSI model which discovers the MAC addresses of connected interfaces. It is one of the protocols which is

implemented by INET in an ARP module. Interfaces don't send information between one another unless the MAC address of the connecting interface is known. ARP requests are thus sent out to a neighbor to ask for its MAC address, which is returned in an ARP reply. If an ARP request does not receive a reply after a certain time-out period, then a new request will be issued. It was found that the default parameter in INET for ARP is to have a time-out of 1 sec. During times of congestion, an ARP request would occasionally get dropped by a queue, and the requesting interface would need to re-issue a request. A 1-second timeout for the ARP requests was a bit unrealistic for the time scales used in this model, where the response is expected on the order of a few hundred nanoseconds. After adjusting to a parameter of 10 milliseconds for the retransmission time-out, the simulation was re-run and the results, shown in Fig. 11, began to show what looks like a power law tail.

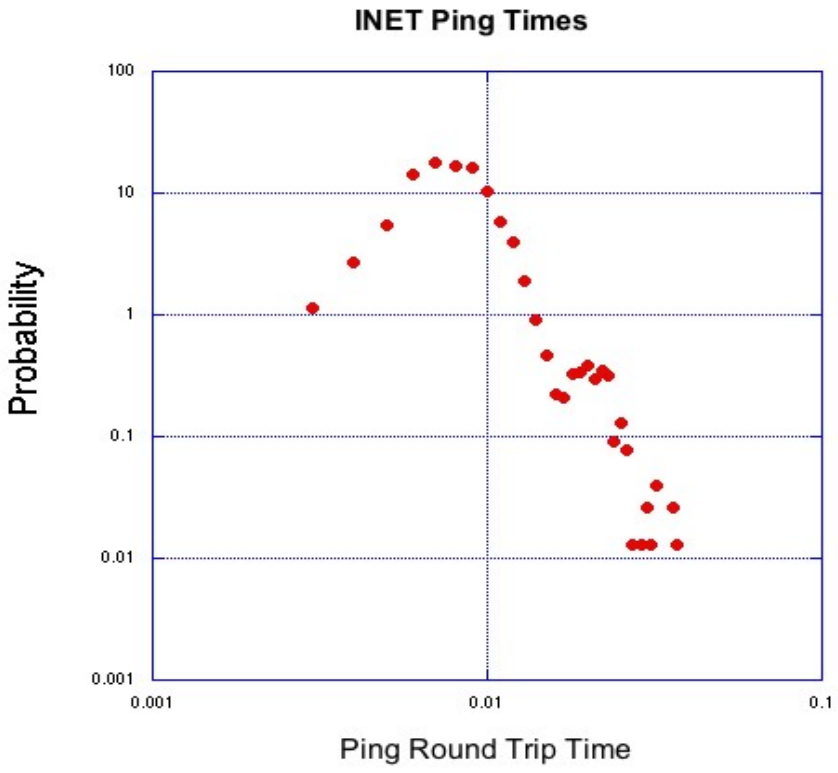


Fig. 11 A PDF of ping times with ARP timeouts at 10 milliseconds

When compared to the previous results (shown again in Fig 12), the distribution shown in Fig. 11 is visibly different.

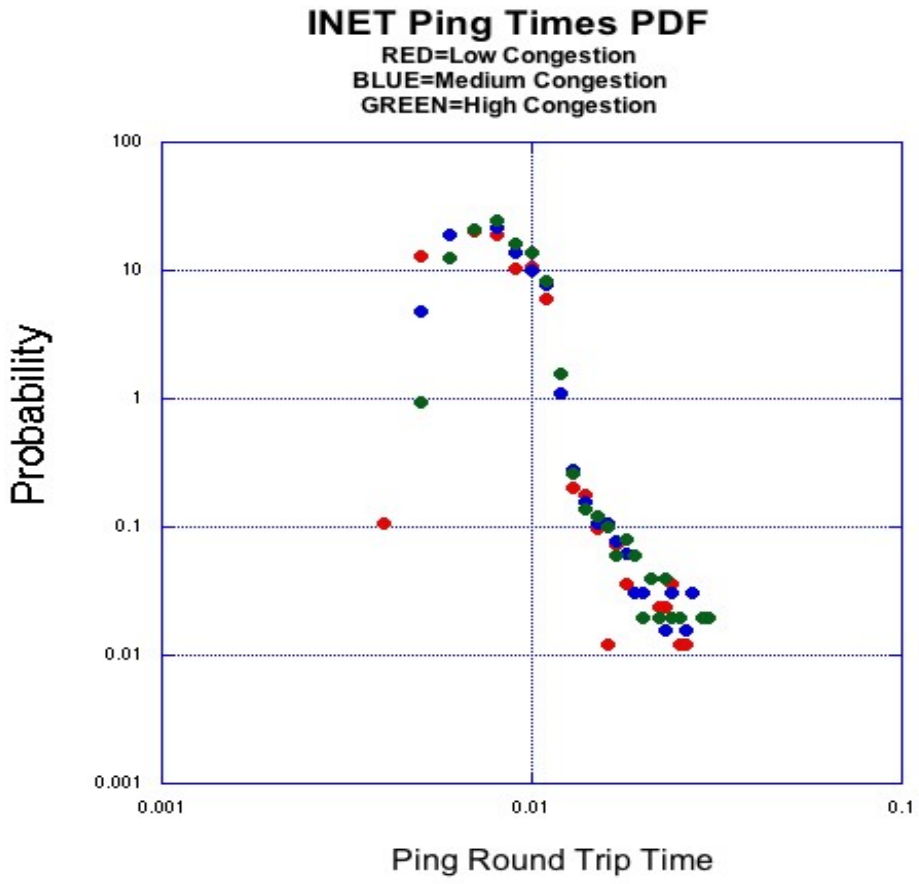


Fig 12 PDF of Ping Times with no power law tail

The ping times in Fig 12 have a much steeper drop off from the top before "leveling" off into a more linear relationship, whereas the ping times in Fig. 11 have a much closer to linear correlation starting from the top and moving right along the plot. The power law tail in Fig. 11 is not as strong as the distribution from the planet lab data in Fig. 1, but this experiment shows that even a small amount of feedback in the system will significantly affect the dynamic behavior.

Real networks have many factors affecting the dynamics, and this model needs to capture the characteristics which produce their nonlinear behavior.

3.3 Exp. 4: Adding Dynamic Routing

Dynamic Routing

Up until Experiment 5, the routing has been statically set up by the INET Routers during the initialization phase. Real networks and routers don't discover the absolute shortest paths between hosts. Real routers run dynamic routing protocols in order to make a best effort for routing data between hosts as efficiently as possible. Dynamic routing allows for new routes to be discovered on the fly as networks increase in size. They also allow ways to handle failure and discover when routes go down. Without dynamic routing, routes would have to be calibrated by hand and node failures could be extremely time consuming and difficult to fix. One of the functional requirements for this model is to have dynamic routing capabilities in order to simulate network growth and response to congestion and ultimately node failure.

There are many dynamic routing protocols in networks today, each running some variation of finding an efficient path between all interfaces on the network. RIP, OSPF and BGP are three of the main routing protocols used in today's networks.[9]

RIP, or Routing Information Protocol, is a routing protocol using the hop-count, number of connections between points, as the primary routing metric. This means that the lower the number of nodes a packet must pass between is indicative of a more efficient path than one with a higher count. The case that lower hop count means quicker or more efficient path is not always true because the time taken between hops is not factored. However, it usually gives a path that is good enough as part of the best-effort routing of large networks. The primary issue with RIP is

that alone it is unsuited for very large networks. To avoid infinite cycles RIP employs a max hop count which limits the size of the network for which it can be used as the stand-alone routing protocol. As an internal protocol either within an autonomous system or a small network it is well suited. [9]

OSPF, or Open Shortest Path First, is an extensively used interior routing protocol. Like RIP it is unsuited for a stand-alone protocol for a network so large as the Internet, but it is very effective for routing within an AS. OSPF gathers information from routers and uses algorithms to construct a shortest path tree containing the nodes within the AS. [9]

BGP, or Border Gateway Protocol, is the primary routing protocol between AS's. BGP uses different metrics than most interior protocols like RIP and OSPF. BGP can be used internally within an AS as well by coupling OSPF networks which couldn't scale on their own. BGP is arguably one of the most important protocols in the functionality of the Internet today. [9]

Quagga [8] is software which provides implementations of many routing protocols for unix-based systems. Quagga includes daemons for running OSPF, BGP and RIP. These, as discussed above are some of the primary protocols for routing today. They are implemented worldwide across various networks including both small local networks and wide area networks (WAN). Quagga provides a way for using such protocols on unix systems.

INET-Quagga

Inet-Quagga was developed to add dynamic routing capabilities to INET simulations. Modules for emulating Quagga were developed and the new QuaggaRouter modules were created to replace the older Router modules. INET-Quagga currently provides a daemon to run BGP, RIP or OSPF through any QuaggaRouter, providing dynamic routing abilities to OMNET simulations. Rather than calculating routes during the Initialization phase of the simulations,

now the routing daemons will “discover” routes by passing messages between one another and learn of new routes as the simulations progress. This feature helps create more realistic simulations and adds required functionality to the project.

ReaSE, the network generator, is not compliant with Inet-Quagga. ReaSE was created independently of the INET-Quagga project, and cannot create instances of Quagga-enabled routers. A script, as shown in Appendix E, was created to quickly turn a network created with REase to one with dynamic routing capabilities by turning every INET Router module into a QuaggaRouter. Inet-Quagga relies on each host and router having configuration files which define any default routes and also assign an IP address to each interface. Appendix B contains a sample configuration file for a QuaggaRouter.

Each interface is assigned an IP in the IFCONFIG portion of the file. The ROUTE portion defines the initial routes for the QuaggaRouter, more routes are added dynamically during the simulation. Along with these configuration files, each QuaggaRouter needs a file for each routing daemon. RIP, OSPF and BGP each have unique configuration files. There are examples of each in Appendix B. The format of the configuration files is the same as Quagga configuration files for real UNIX systems.

Creating these files by hand for a large network is far too time-consuming to be practical, so a way to automate this process is required. Two scripts were creating for automation, shown in Appendix E. The first will create initialization files for every host and router and create the proper INET-Quagga files for each daemon (OSPF,BGP,RIP), and the second creates the simulation’s initialization file by defining the paths to each routers and daemons configuration files.

A small test network using eight routers shown in Fig. 13 was used to test the INET-QUAGGA implementations of BGP, RIP, and OSPF.

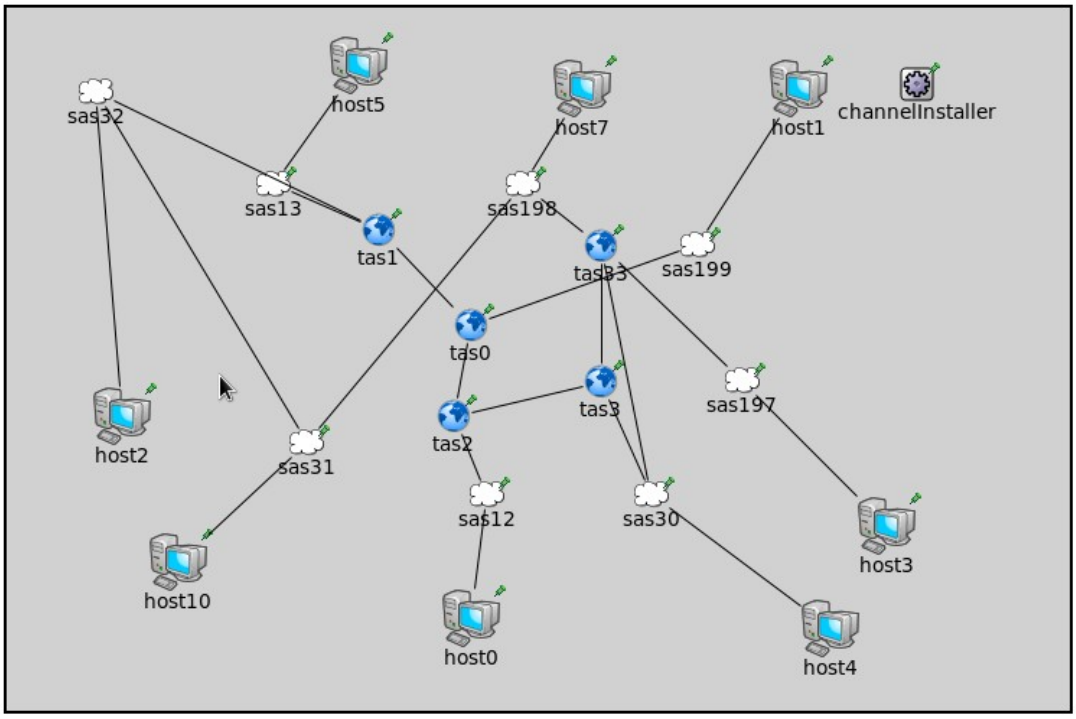


Fig 13. Test Network for INET-Quagga

The expectation is that this network will start with each router only knowing each interface's subnet. As the simulation runs, the dynamic protocols should create routes and allow the each host on the network to communicate to one another. Along with this expectation, the system should have more feedback, thus possibly increasing the power-law correlation in the ping times.

Exp 4 Results

Ping Results

Fig 14 below shows the results of the ping times for pings from host2 to host0 for the first 1200 seconds of simulation time.

INET Ping Time Series

Time

FIG 14: Ping Times from initial run with INET-Quagga dynamic routing

There is a noticeable lack of ping times for the first few seconds as the routing protocols built their complete routing tables. At around 5 seconds is where a path between Host4 and Host2 was finally discovered and traffic was able to resume. The routing tables will need to be maintained throughout the simulation, which means a lot of extra communication between neighboring routers, creating more feedback in the system.

Strong Power-Law tail

As expected, the extra feedback in the system from the routers dynamically building and maintaining routing tables led to a stronger power law correlation in the PDF of the ping times, as shown in Fig. 15.

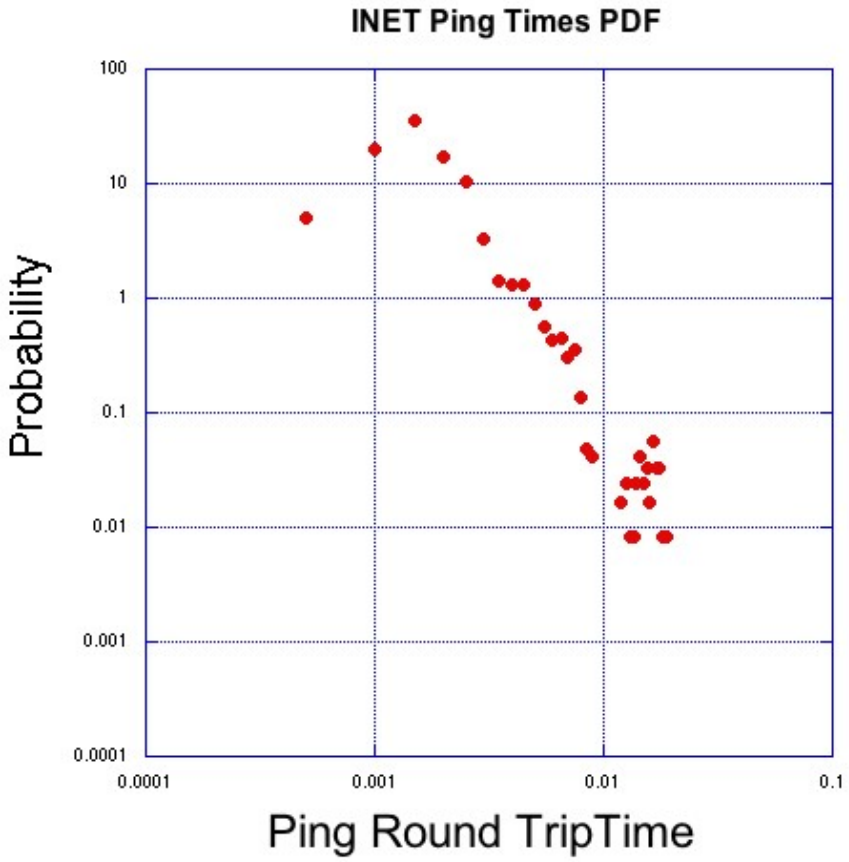


Fig. 15 Power Law Tail in Simulated Network for RIP dynamic routing

This correlation is characteristic of what has been observed in real networks. It's a stronger correlation than was previously seen in the static networks. This is likely due to the increased amount of feedback in the system due to the dynamic routing protocols. This stronger correlation was exhibited regardless of the routing protocol used.

3.4 Exp. 5: Constructing a Real Network

Construct UAA Network

The next test for the network model was to try to model an existing large network, and since we had access to the structure and data from the UAA network, it was a good choice for a test network. This network was constructed by hand creating modules to represent sub-networks,

such as the Homer LAN.

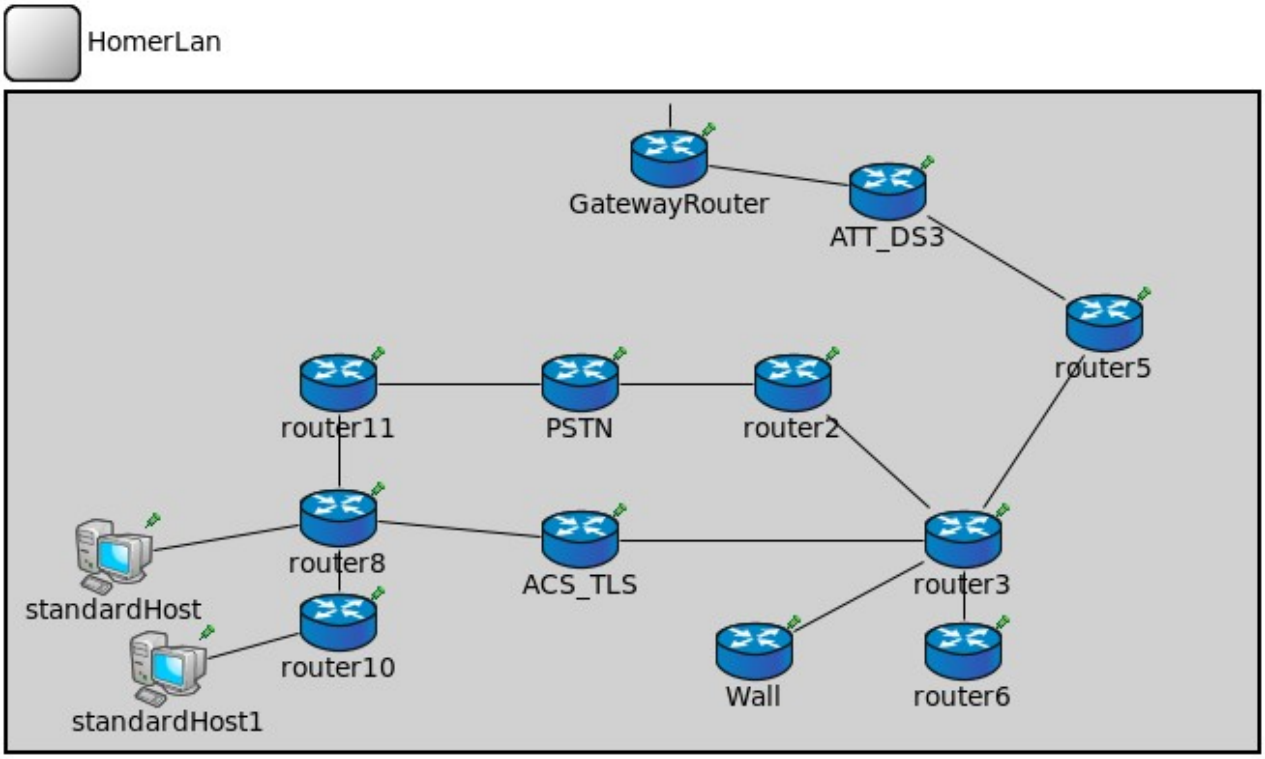


Fig16 HomerLan module created in INET

The real UAA network has a very low bandwidth connection to outlying networks such as those found in Homer. By using the INET PPP Interface module, it is possible to simulate the low bandwidth through initial network parameters. High bandwidth channels and low bandwidth channels can be defined and created in INET and used to create more accurate representations of real-world networks. The above network involves a low bandwidth channel in the gateway router named "GatewayRouter." Other modules like the HomerLan were connected in the same manner as the other networks, and we were able to easily run simulations in the same manner. A few "StandardHost" modules were hung off of some edge routers to run the ping tests.

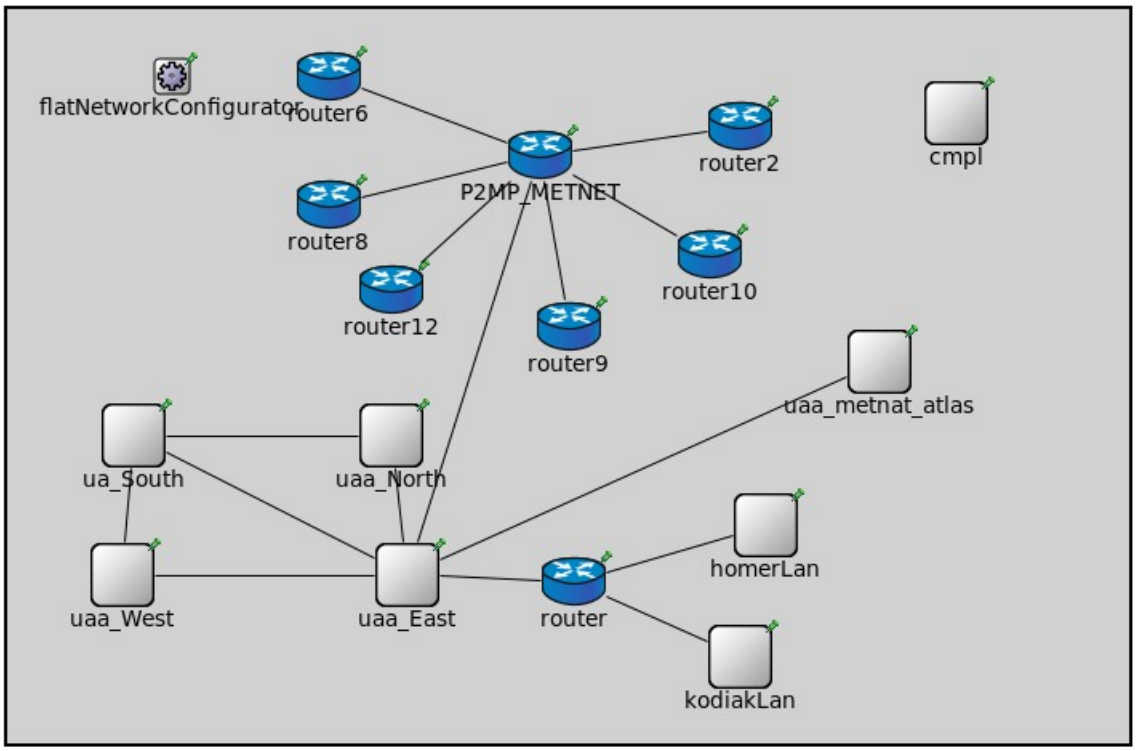


Fig 17: UAA Network created in INET

The UAA Network has been split up into smaller network then connected to create a larger network. This bottom up method of creating networks can be used to create models of real-world networks for simulations in OMNET. The coupling of small networks in INET

should work in a similar manner as coupling compound modules, and the simulations should route the same as if it were one large network.

Exp. 5 Results

UAA Network ping test results

The ping times of the UAA network showing the difference between pings going through the low bandwidth connections and those which avoid them. Fig 18 Below shows a ping times between Homer and the main network and Fig 19 shows those between hosts on UAA Main Campus.

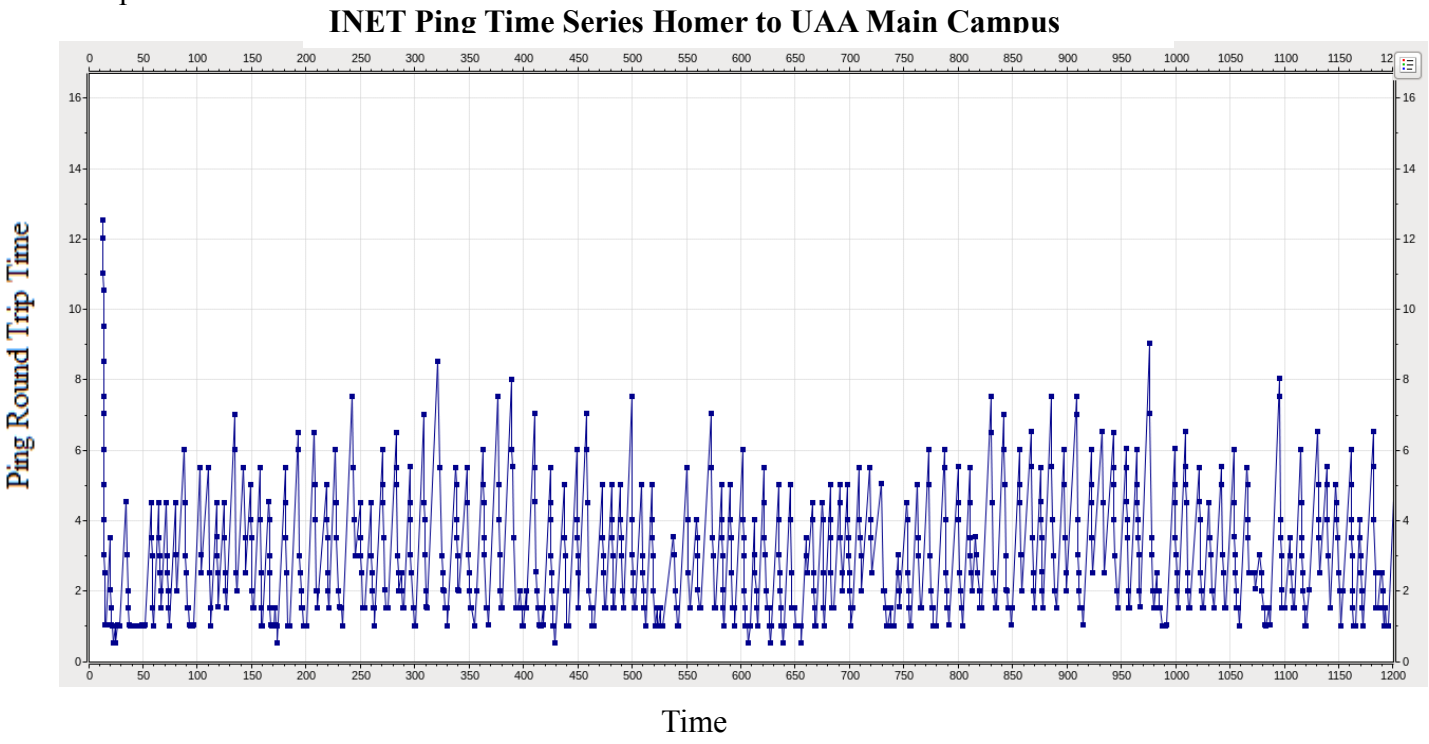


Fig. 18 Ping times between low bandwidth connection HomerLan and Main Campus

INET Ping Time Series Across UAA Main Campus

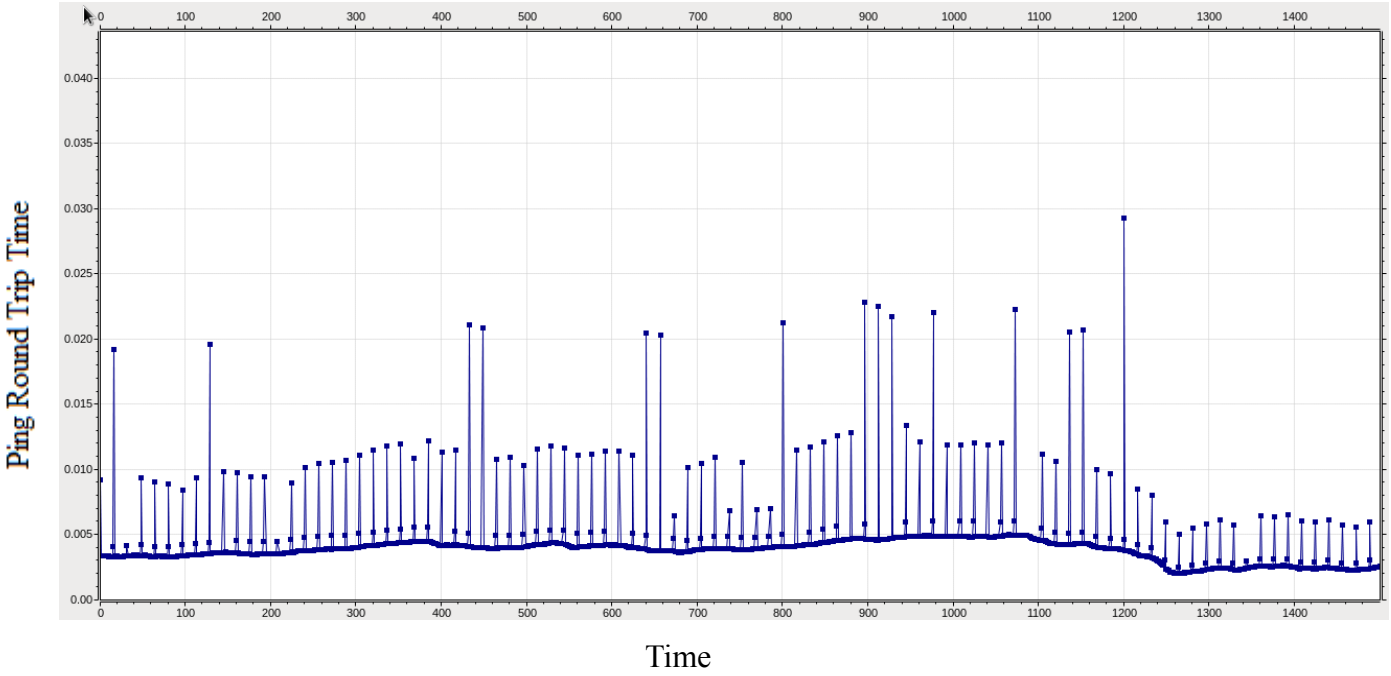


Fig. 19 Ping times on UAA Network cross-campus

The ping results from figures 18 and 19 show how the low bandwidth connection between the LAN in Homer and the main campus in Anchorage can be simulated in INET. The cross campus pings were coming in on the order of milliseconds where it took as many as eight seconds to get a response in Homer. The numbers from these runs may not be one hundred percent realistic but it is a more accurate simulation than the pings all coming in from low and high bandwidth connection on the exact same order of magnitude. Fig 20 below shows real ping data from cross campus pings and Fig 21 shows pings out to Bethel over a low bandwidth connection.

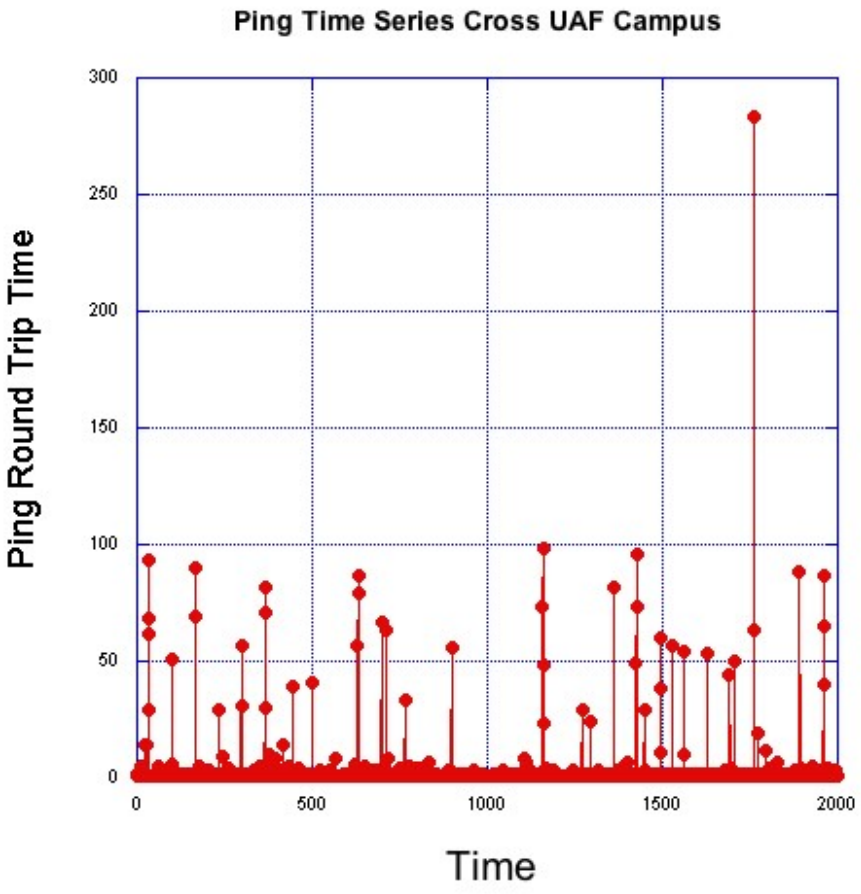


Fig 20 Four hop ping between two hosts across the UAF campus

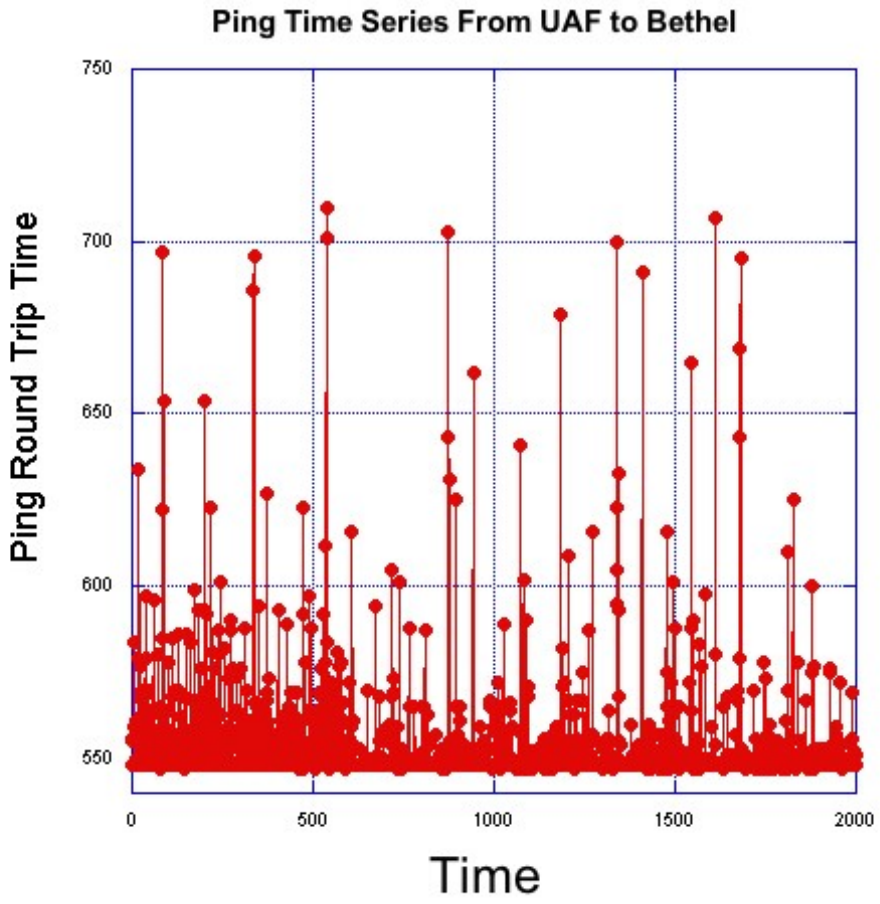


Fig 21 Four-hop ping times between UAF and Bethel over one low-bandwidth connection

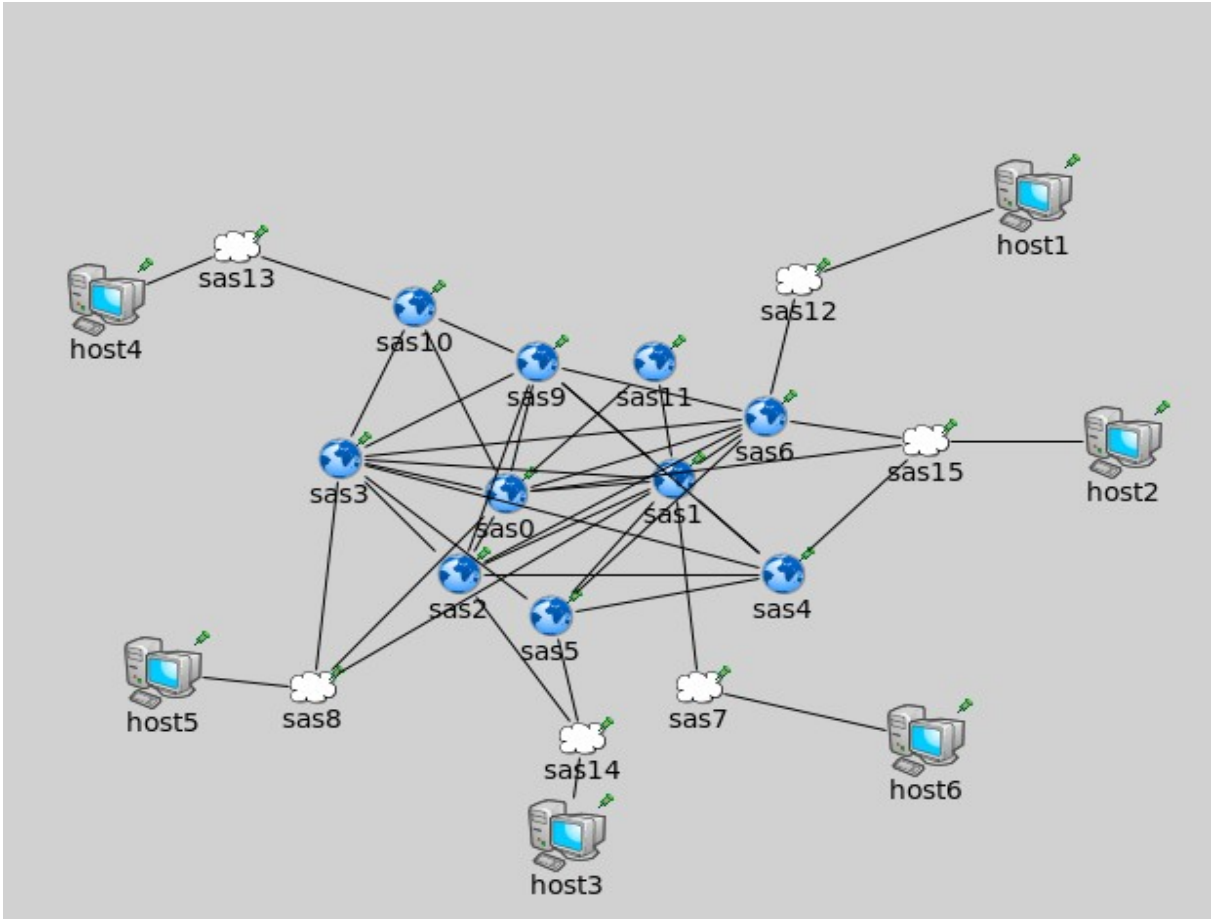
The ping times across two different real connections showed the role that one low-bandwidth connection will play in the network dynamics. Both Fig 20 and Fig 21 are four hops between hosts, or there were four routers in between. Just by going through one low bandwidth connection, the ping times from Bethel came back with far more variance, and on a different order of magnitude. Most of the pings across campus returned after only a few milliseconds, where no pings from Bethel returned under around five hundred and fifty milliseconds. The time taken to go through the one long hop to Bethel dominates the dynamic behavior of these pings,

much like in the simulated runs. This functionality of defining the bandwidth of channels in the INET networks provides a way to better simulate real networks.

3.5 Exp. 6: Testing Network Response to Failure/Congestion

Setting up experiment parameters

This final experiment will test the capabilities of the routers to discover new paths during periods of high congestion and node failure. Figure 22 below shows a network with INET-Quagga "QuaggaRouter" modules capable of running dynamic routing protocols.


host1
sas13
host4
sas10
sas12
sas9
sas11
sas6
sas15
host2
sas3
sas1
sas0
sas2
sas4
sas5
host5
sas8
sas7
sas14
host6
host3

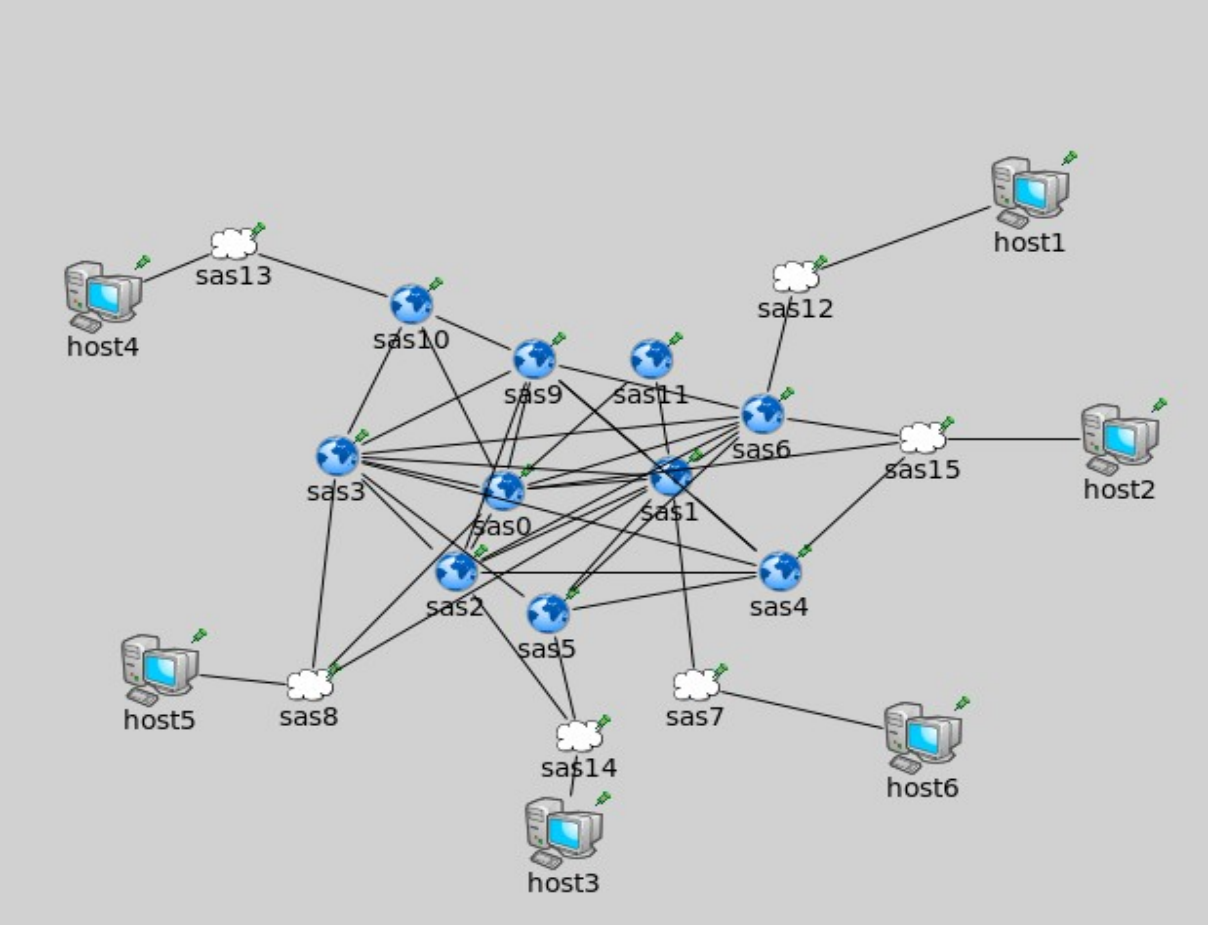


Fig. 22 Small Network used to test dynamic routing response to node failure and congestion

The test parameters are as follows:

The network simulation will be run and the routers will build their routing tables and the Host4 will ping Host2. The simulation will be stopped, and the route will be recorded. A second simulation will be run, this time after 150 seconds, one of the middle routers in the route recorded between Host4 and Host2 will be so congested by our Fake traffic, no traffic generated from INET and the INET-Quagga modules will be able to get in or out. This will simulate a router failure. The expectation is that the failed node will be detected and traffic between Host4 and Host2 will be rerouted. If another route is ultimately established between the two hosts, ping times should resume and the routers will be shown to effectively handle node failure.

In order to run a test against the network congestion, the simulation will continue and after 1100 seconds, the congestion on the node will drop, simulating the over congested router being relaxed and reentering the network. The routers are expected to reinstate the initial route, because it should be the shorter or better route as determined by the protocol, thus simulating the routers responding to high congestion and reestablishing a quicker route as congestion is relaxed.

Exp. 6 Results

The initial route

Fig. 23 below highlights the initial route between Host4 and Host2 as determined by the routers.

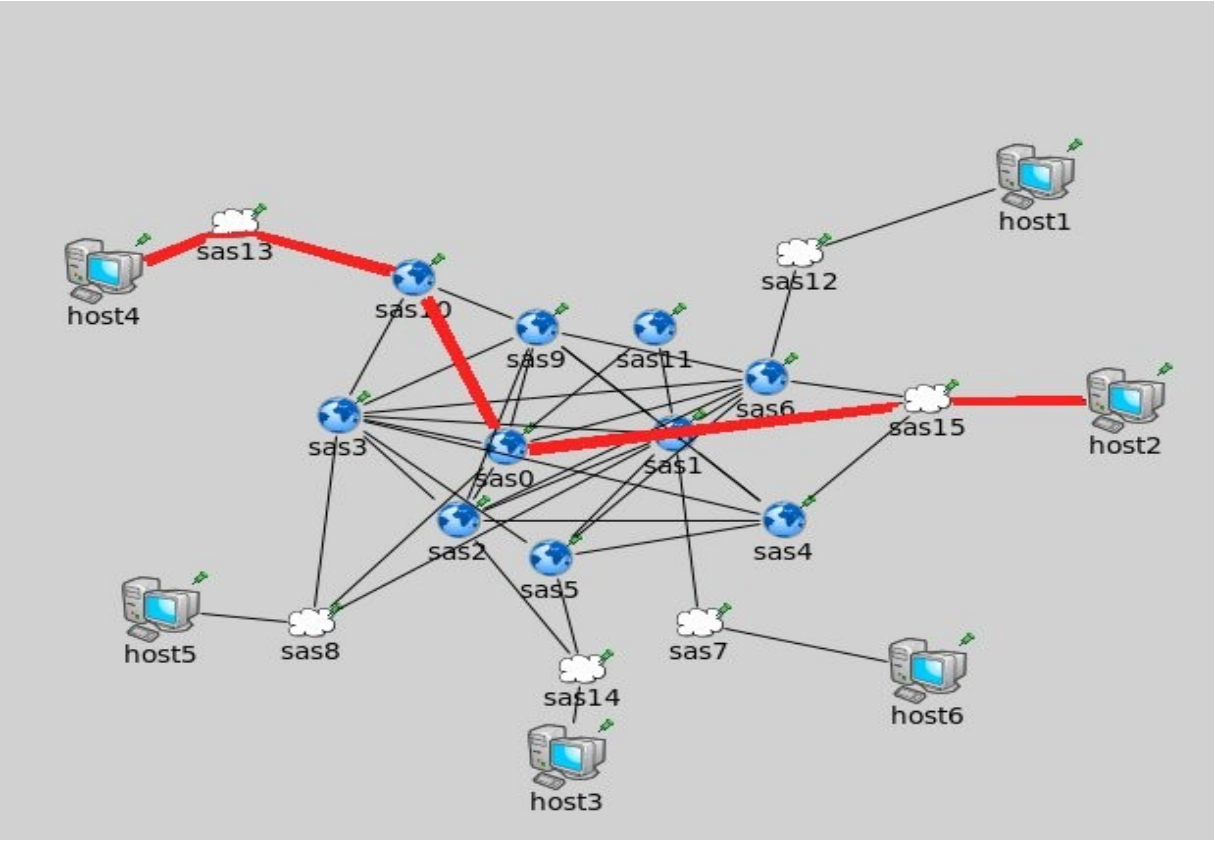


Fig 23 The initial route between Host4 and Host2 (highlighted in red)

After 200 seconds router "sas0" was over congested by the Complex module. All packets sent to this router were dropped, and ultimately traffic between Host4 and Host2 was rerouted through the path highlighted in Fig 24 below.

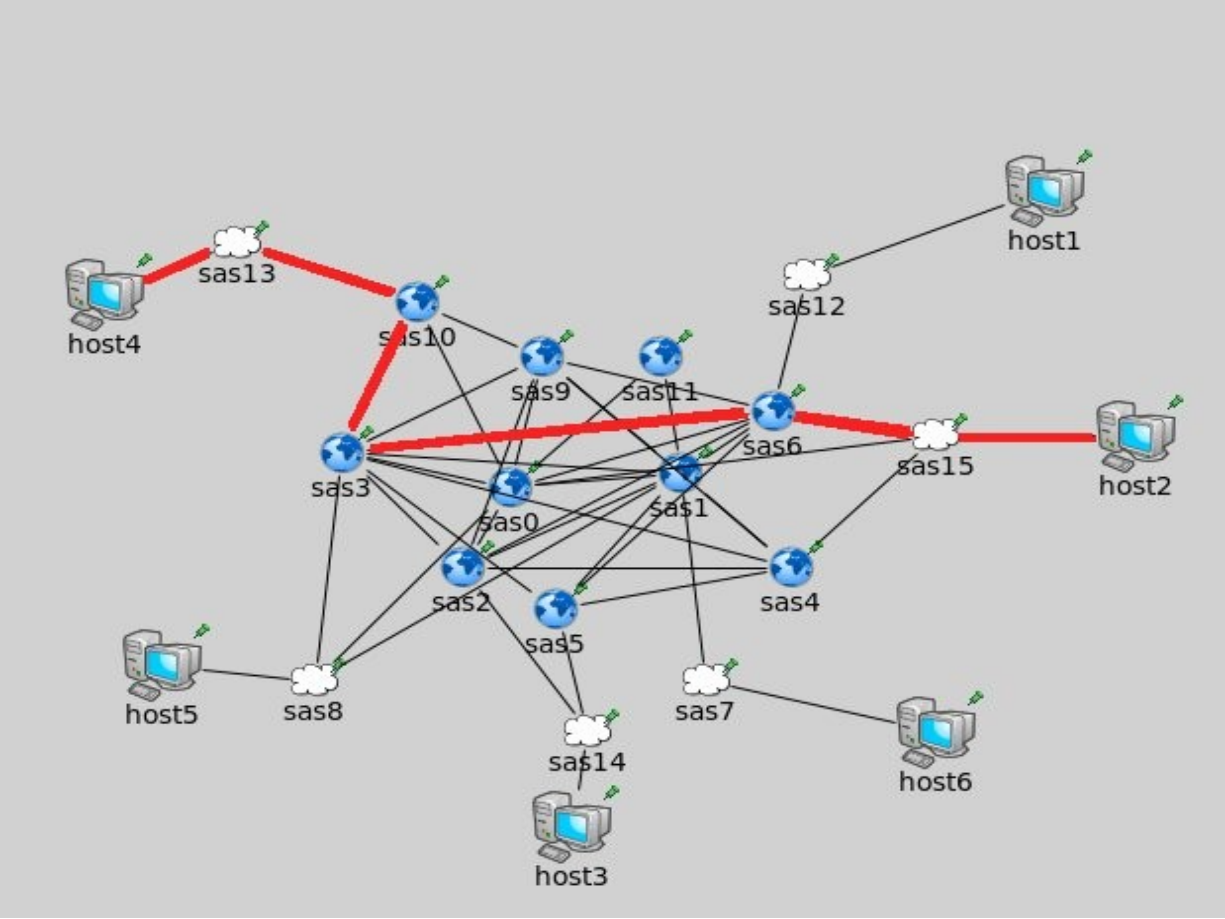


Fig 24 The route between Host4 and Host2 after sas0 was shutdown

As expected after a few minutes of lost traffic, the routers reconstructed their routing tables and sent the traffic from Host4 through "sas3" and then "sas6" to get to its final destination of Host2. Since the initial routing table set up traffic to go through "sas0," when the congestion was relieved the traffic ultimately rerouted back through "sas0," thus finishing a successful response to congestion. As discussed in Section 3, this test by no means proves that this representation of failing routers is one hundred percent accurate, but the response to any form of failure or congestion by the routers is important for the model to provide an understanding of the dynamic behavior of the system. Even though the system and the nonlinear dynamics will only be a reflection of reality, the model should provide a better understanding of

some specific causes of this behavior. Fig 25 below shows the ping times between Host4 and Host2.

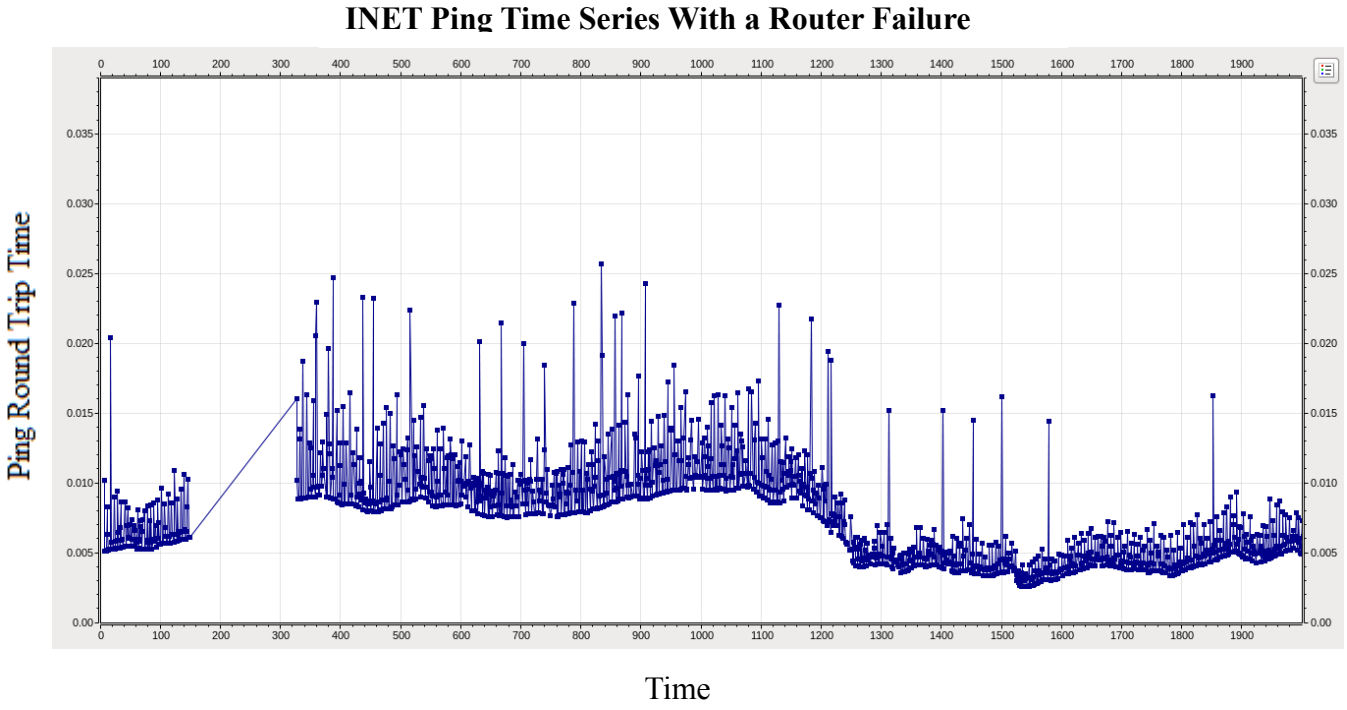


Fig. 25 Ping times during simulation of failure and congestion

There is a gap in the ping times between around 150 seconds and 300 seconds. This gap reflects the few minutes where the router was overly congested and dropping all incoming packets before a new route was established. Between 1100 seconds and 1200 there is a sharp decline in the ping times as the dynamic routing protocols rediscover the shortest route.

IV PROJECT SUMMARY

4.1 Project Summary

The network simulating tool has been constructed and tested against the fulfillment of the functional requirements. Using the INET framework on top of the OMNET network simulation software, the model reflects real network structure and protocols used to communicate and route traffic across the network. Because large communication networks are unique among complex systems in structure and communication protocols, it was a requirement for this model to capture these features unique to communication networks in order to study the unique dynamic behavior, to learn what features lead to the nonlinearity.

INET-Quagga was implemented to provide dynamic routing capabilities to INET. The dynamic routing provided capabilities for the simulation to capture more realistic response towards network congestion and node failure, allowing for the modeling of more long term network evolution that would not be available without dynamic routing. With the addition of the "Complex" module to provide background traffic in the network, the simulations can scale well enough to capture the nonlinear dynamic behavior used for validation, as shown with the power law in Fig. 26

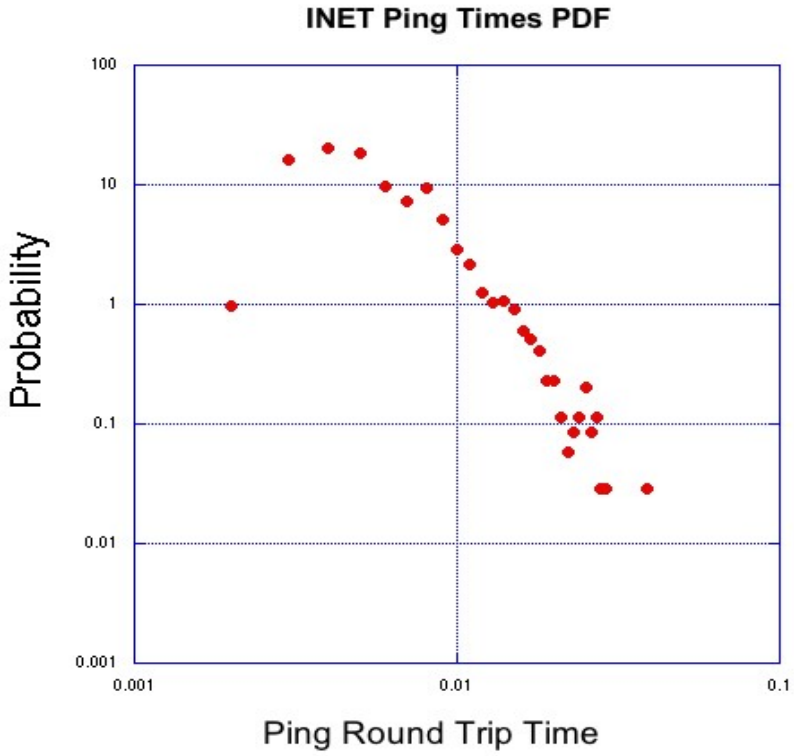


Fig 26 A Power Law correlation in the Ping Times of simulated network from Fig 17 with dynamic routing

The Ping return times for congested networks followed similar distribution to the power law seen with the PlanetLab data. The power law tail of the ping times captured by the simulator appears to be directly correlated with the congestion of the system and the amount of maintenance or feedback the system requires in order to remain active. The power laws were not seen in simulations with little or no congestion, or those networks which are set up completely static, with no dynamic routing or maintenance being performed at run time. This correlation helps to show some of the causes of nonlinear behavior in complex communication networks.

On top of the functional requirements, a number of scripts, shown in Appendix E, were written to automate the process from creating networks using the ReaSE network topology generator all the way to creating the necessary configuration files for INET-Quagga and the

initialization files to run the OMNET simulations. This automation will help simplify the use of the tool for creating and running network simulations. Along with the automation, a user manual (Appendix F) was written to outline how to install the software on either Fedora Linux or Ubuntu operating systems, and several tutorials on how to create and run different simulations. Along with instructional outlining the installation and use, the user manual includes explanation on the structure of the model, and additional parameter which can be controlled outside of the scope of the automation scripts in Appendix E.

4.2 Project Conclusions and Future Work

This project involved creating an initial network modeling tool to run simulations for studying the nonlinear dynamic behavior of communication networks. The tool was created using the OMNET network modeling software. The INET framework provided OMNET modules to include definitions of communication network components, and the INET-Quagga extension provided the necessary dynamic routing capabilities. The ReaSE topology generator is used to quickly create realistic networks, and the scripts outlined in Appendix E provide automation of the entire process of creating model networks and running simulations.

Now that the initial model has been created and validated against the data provided by PlanetLab, it is ready to begin running simulations. The main recommendation is to turn the "Complex" module into a more powerful traffic controller, and add the functionality of controlling events during run time. For instance, experiment six outlined a situation where the "Complex" module was used to simulate the failure of a router by congesting the router so much no traffic was allowed in or out. In a similar manner, the module could be increased to allow more simple control over failure or other events in a simulation, possibly providing a way to study the cascading failure of congested communication networks.

This network simulation tool, however, must undergo more tests in order become truly useful as a predictive model. Even though the simulated networks respond to dynamic changes such as high levels of congestion and failure, there are a few considerations which must be taken into account. One example might be the failure of a route. In this model there is no control over the fake background traffic in the network, it only exists to congest the network and slow down the real traffic. In real-world scenarios, however, this is not always the case. For instance, if a core router fails, all the traffic which usually passes through must find alternate routes, increasing the load in part of the network. Similarly when a gateway router between a local network and a wide area network fails, the wide area network may in fact receive a decrease in traffic, lowering its overall congestion. This example is just one consideration which could affect the accuracy of the model. Predictive models can only be a reflection of reality and cannot capture all the dynamic behavior of a system. They have inherent limitations because it is not possible to capture every factor which affects the system in the real world. This network modeling tool is the first stepping stone in the effort to construct a good predictive model for studying the complex dynamical behavior of congested communication networks.

List of References

[1] Newman, David, N. Sizemore, V.E. Lynch, and B.A. Carreras, "Growth and Propagation of Disturbances in a Communication Network Model." Presented at the 35th Hawaii International Conference on System Sciences, January 2002, Hawaii, USA. Unpublished conference paper, 2002.

[2] Back, Per. *How Nature Works: The Science of Self-organized Criticality*. New York: Copernicus Press 1996

[3] "OMNeT++ 4.2 documentation and tutorials." *OMNeT++*. 27 March 2012. Web. 01 January 2011. <http://www.omnetpp.org/documentation>

[4] "Exploring Networks of the Future" *GENI.* March 2012. National Science Foundation. <http://www.geni.net/>

[5] "PLANETLAB: An open platform for developing, deploying, and accessing planetary-scale services" *PlanetLab* March 2012. Princeton University <www.planet-lab.org>

[6] "INET Framework" *OMNeT++* March 21, 2012 <http://inet.omnetpp.org/ >

[7] "INET Quagga" *OMNeT++* January 2006 <http://www.omnetpp.org/pmwiki/index.php?n=Main.INETQuagga>

[8] "Quagga Routing Suite" *GNU Zebra.* January 4, 2011 <http://www.nongnu.org/quagga/>

[9] Kurose, James F., Ross, Keith W. *Computer Networking: A Top-Down Approach.* May 23, 2004

[10] "ReaSE – A generator for realistic simulation environments" *TeleMatics*. September 2011. Karlsruher Institut fur Technologie <https://i72projekte.tm.uka.de/trac/ReaSE>

[11] "KaleidaGraph – scientific graphing, curve fitting, data analysis software" *Synergy Software.* <http://www.synergy.com/>

[12] "Network Simulators" September 2010. National Science Foundation <http://www.icir.org/models/simulators.html>

Appendix A: OMNET Initialization Files

A.1: Simple initialization file for exp. 1:.

```
[General]
            network = Network
            result-dir = pingTEST
            *.standardHost.pingApp.destAddr = "standardHost1"
```

This is the initialization file for the network in Fig. 3. This file defines the simulation parameters, in this case "standardHost" will be using the pingApp to send pings to its destination of "standardHost1" and the results will be stored in the directory named "pingTEST."

A.2: OMNET Initialization file for Exp. 2:

```
[General]
network = Inet200
fname-append-host = false
output-scalar-file = ${resultdir}/${configname}-${runnumber}.sca
output-vectors-memory-limit = 16MB
record-eventlog = false
ned-path = /home/Adam/inet-framework-inet-a3307ab/src/.;/home/Adam/inet-
framework-inet-a3307ab/Adam/.
cmdenv-express-mode = true
result-dir = ders
sim-time-limit = 12490s
**.module-eventlog-recording = true
**.scalar-recording = true

*.host0.pingApp.destAddr = "host131"
*.host1.pingApp.destAddr = "host61"
*.host2.pingApp.destAddr = "host122"
*.host3.pingApp.destAddr = "host124"
*.host4.pingApp.destAddr = "host142"
*.host5.pingApp.destAddr = "host30"
*.host6.pingApp.destAddr = "host52"
*.host7.pingApp.destAddr = "host119"
*.host8.pingApp.destAddr = "host43"
*.host9.pingApp.destAddr = "host86"
*.host10.pingApp.destAddr = "host74"
*.host11.pingApp.destAddr = "host98"
*.host12.pingApp.destAddr = "host56"
*.host13.pingApp.destAddr = "host80"
*.host14.pingApp.destAddr = "host148"
*.host15.pingApp.destAddr = "host142"
*.host16.pingApp.destAddr = "host99"
*.host17.pingApp.destAddr = "host111"
*.host18.pingApp.destAddr = "host22"
*.host19.pingApp.destAddr = "host94"
*.host20.pingApp.destAddr = "host2"
*.host21.pingApp.destAddr = "host37"
*.host22.pingApp.destAddr = "host21"
*.host23.pingApp.destAddr = "host125"
*.host24.pingApp.destAddr = "host24"
*.host25.pingApp.destAddr = "host62"
*.host26.pingApp.destAddr = "host20"
*.host27.pingApp.destAddr = "host16"
*.host28.pingApp.destAddr = "host155"
*.host29.pingApp.destAddr = "host34"
*.host30.pingApp.destAddr = "host80"
*.host31.pingApp.destAddr = "host130"
*.host32.pingApp.destAddr = "host95"
*.host33.pingApp.destAddr = "host46"
*.host34.pingApp.destAddr = "host99"
*.host35.pingApp.destAddr = "host81"
*.host36.pingApp.destAddr = "host76"
*.host37.pingApp.destAddr = "host151"
*.host38.pingApp.destAddr = "host45"
*.host39.pingApp.destAddr = "host120"
*.host40.pingApp.destAddr = "host82"
*.host41.pingApp.destAddr = "host120"
*.host42.pingApp.destAddr = "host62"
*.host43.pingApp.destAddr = "host139"
*.host44.pingApp.destAddr = "host44"
*.host45.pingApp.destAddr = "host54"
```

```
*.host46.pingApp.destAddr = "host126"
*.host47.pingApp.destAddr = "host143"
*.host48.pingApp.destAddr = "host10"
*.host49.pingApp.destAddr = "host148"
*.host50.pingApp.destAddr = "host82"
*.host51.pingApp.destAddr = "host13"
*.host52.pingApp.destAddr = "host29"
*.host53.pingApp.destAddr = "host103"
*.host54.pingApp.destAddr = "host138"
*.host55.pingApp.destAddr = "host54"
*.host56.pingApp.destAddr = "host10"
*.host57.pingApp.destAddr = "host3"
*.host58.pingApp.destAddr = "host71"
*.host59.pingApp.destAddr = "host9"
*.host60.pingApp.destAddr = "host37"
*.host61.pingApp.destAddr = "host151"
*.host62.pingApp.destAddr = "host140"
*.host63.pingApp.destAddr = "host132"
*.host64.pingApp.destAddr = "host41"
*.host65.pingApp.destAddr = "host84"
*.host66.pingApp.destAddr = "host58"
*.host67.pingApp.destAddr = "host118"
*.host68.pingApp.destAddr = "host79"
*.host69.pingApp.destAddr = "host104"
*.host70.pingApp.destAddr = "host82"
*.host71.pingApp.destAddr = "host6"
*.host72.pingApp.destAddr = "host68"
*.host73.pingApp.destAddr = "host145"
*.host74.pingApp.destAddr = "host145"
*.host75.pingApp.destAddr = "host112"
*.host76.pingApp.destAddr = "host44"
*.host77.pingApp.destAddr = "host115"
*.host78.pingApp.destAddr = "host99"
*.host79.pingApp.destAddr = "host55"
*.host80.pingApp.destAddr = "host107"
*.host81.pingApp.destAddr = "host25"
*.host82.pingApp.destAddr = "host68"
*.host83.pingApp.destAddr = "host137"
*.host84.pingApp.destAddr = "host129"
*.host85.pingApp.destAddr = "host51"
*.host86.pingApp.destAddr = "host35"
*.host87.pingApp.destAddr = "host139"
*.host88.pingApp.destAddr = "host54"
*.host89.pingApp.destAddr = "host107"
*.host90.pingApp.destAddr = "host149"
*.host91.pingApp.destAddr = "host91"
*.host92.pingApp.destAddr = "host102"
*.host93.pingApp.destAddr = "host133"
*.host94.pingApp.destAddr = "host68"
*.host95.pingApp.destAddr = "host144"
*.host96.pingApp.destAddr = "host62"
*.host97.pingApp.destAddr = "host127"
*.host98.pingApp.destAddr = "host106"
*.host99.pingApp.destAddr = "host142"
*.host100.pingApp.destAddr = "host75"
*.host101.pingApp.destAddr = "host33"
*.host102.pingApp.destAddr = "host148"
*.host103.pingApp.destAddr = "host143"
*.host104.pingApp.destAddr = "host23"
*.host105.pingApp.destAddr = "host137"
*.host106.pingApp.destAddr = "host100"
*.host107.pingApp.destAddr = "host67"
*.host108.pingApp.destAddr = "host96"
```

```
*.host109.pingApp.destAddr = "host43"
*.host110.pingApp.destAddr = "host122"
*.host111.pingApp.destAddr = "host47"
*.host112.pingApp.destAddr = "host69"
*.host113.pingApp.destAddr = "host35"
*.host114.pingApp.destAddr = "host29"
*.host115.pingApp.destAddr = "host43"
*.host116.pingApp.destAddr = "host86"
*.host117.pingApp.destAddr = "host64"
*.host118.pingApp.destAddr = "host26"
*.host119.pingApp.destAddr = "host141"
*.host120.pingApp.destAddr = "host16"
*.host121.pingApp.destAddr = "host19"
*.host122.pingApp.destAddr = "host77"
*.host123.pingApp.destAddr = "host118"
*.host124.pingApp.destAddr = "host153"
*.host125.pingApp.destAddr = "host145"
*.host126.pingApp.destAddr = "host106"
*.host127.pingApp.destAddr = "host59"
*.host128.pingApp.destAddr = "host116"
*.host129.pingApp.destAddr = "host57"
*.host130.pingApp.destAddr = "host45"
*.host131.pingApp.destAddr = "host36"
*.host132.pingApp.destAddr = "host91"
*.host133.pingApp.destAddr = "host38"
*.host134.pingApp.destAddr = "host23"
*.host135.pingApp.destAddr = "host114"
*.host136.pingApp.destAddr = "host19"
*.host137.pingApp.destAddr = "host123"
*.host138.pingApp.destAddr = "host25"
*.host139.pingApp.destAddr = "host116"
*.host140.pingApp.destAddr = "host11"
*.host141.pingApp.destAddr = "host148"
*.host142.pingApp.destAddr = "host8"
*.host143.pingApp.destAddr = "host81"
*.host144.pingApp.destAddr = "host27"
*.host145.pingApp.destAddr = "host37"
*.host146.pingApp.destAddr = "host124"
*.host147.pingApp.destAddr = "host114"
*.host148.pingApp.destAddr = "host102"
*.host149.pingApp.destAddr = "host150"
*.host150.pingApp.destAddr = "host99"
*.host151.pingApp.destAddr = "host118"
*.host152.pingApp.destAddr = "host14"
*.host153.pingApp.destAddr = "host21"
*.host154.pingApp.destAddr = "host81"
*.host155.pingApp.destAddr = "host12"
*.host156.pingApp.destAddr = "host10"
```

This is one of the initialization files for Exp. 2. Each host is “pinging” a random host every second for the duration of the run.

A.3: OMNET Initialization file for Exp. 3:

```
[General]
network = Inet200
fname-append-host = false
output-scalar-file = ${resultdir}/${configname}-${runnumber}.sca
output-vectors-memory-limit = 16MB
record-eventlog = false
ned-path = /home/Adam/inet-framework-inet-
a3307ab/src/.;/home/Adam/inet-framework-inet-a3307ab/Adam/.
cmdenv-express-mode = true
result-dir = ders
sim-time-limit = 12490s
**.module-eventlog-recording = true
**.scalar-recording = true

*.host0.pingApp.destAddr = "host131"
*.host1.pingApp.destAddr = "host61"
*.host2.pingApp.destAddr = "host122"
*.host3.pingApp.destAddr = "host124"
*.host4.pingApp.destAddr = "host142"
*.host5.pingApp.destAddr = "host30"
*.host6.pingApp.destAddr = "host52"
*.host7.pingApp.destAddr = "host119"
*.host8.pingApp.destAddr = "host43"
*.host9.pingApp.destAddr = "host86"
*.host10.pingApp.destAddr = "host74"
*.host11.pingApp.destAddr = "host98"
*.host12.pingApp.destAddr = "host56"
*.host13.pingApp.destAddr = "host80"
*.host14.pingApp.destAddr = "host148"
*.host15.pingApp.destAddr = "host142"
*.host16.pingApp.destAddr = "host99"
*.host17.pingApp.destAddr = "host111"
*.host18.pingApp.destAddr = "host22"
*.host19.pingApp.destAddr = "host94"
*.host20.pingApp.destAddr = "host2"
*.host21.pingApp.destAddr = "host37"
*.host22.pingApp.destAddr = "host21"
*.host23.pingApp.destAddr = "host125"
*.host24.pingApp.destAddr = "host24"
*.host25.pingApp.destAddr = "host62"
*.host26.pingApp.destAddr = "host20"
*.host27.pingApp.destAddr = "host16"
*.host28.pingApp.destAddr = "host155"
*.host29.pingApp.destAddr = "host34"
*.host30.pingApp.destAddr = "host80"
*.host31.pingApp.destAddr = "host130"
*.host32.pingApp.destAddr = "host95"
*.host33.pingApp.destAddr = "host46"
*.host34.pingApp.destAddr = "host99"
*.host35.pingApp.destAddr = "host81"
*.host36.pingApp.destAddr = "host76"
*.host37.pingApp.destAddr = "host151"
*.host38.pingApp.destAddr = "host45"
*.host39.pingApp.destAddr = "host120"
```

```
*.host40.pingApp.destAddr = "host82"
*.host41.pingApp.destAddr = "host120"
*.host42.pingApp.destAddr = "host62"
*.host43.pingApp.destAddr = "host139"
*.host44.pingApp.destAddr = "host44"
*.host45.pingApp.destAddr = "host54"
*.host46.pingApp.destAddr = "host126"
*.host47.pingApp.destAddr = "host143"
*.host48.pingApp.destAddr = "host10"
*.host49.pingApp.destAddr = "host148"
*.host50.pingApp.destAddr = "host82"
*.host51.pingApp.destAddr = "host13"
*.host52.pingApp.destAddr = "host29"
*.host53.pingApp.destAddr = "host103"
*.host54.pingApp.destAddr = "host138"
*.host55.pingApp.destAddr = "host54"
*.host56.pingApp.destAddr = "host10"
*.host57.pingApp.destAddr = "host3"
*.host58.pingApp.destAddr = "host71"
*.host59.pingApp.destAddr = "host9"
*.host60.pingApp.destAddr = "host37"
*.host61.pingApp.destAddr = "host151"
*.host62.pingApp.destAddr = "host140"
*.host63.pingApp.destAddr = "host132"
*.host64.pingApp.destAddr = "host41"
*.host65.pingApp.destAddr = "host84"
*.host66.pingApp.destAddr = "host58"
*.host67.pingApp.destAddr = "host118"
*.host68.pingApp.destAddr = "host79"
*.host69.pingApp.destAddr = "host104"
*.host70.pingApp.destAddr = "host82"
*.host71.pingApp.destAddr = "host6"
*.host72.pingApp.destAddr = "host68"
*.host73.pingApp.destAddr = "host145"
*.host74.pingApp.destAddr = "host145"
*.host75.pingApp.destAddr = "host112"
*.host76.pingApp.destAddr = "host44"
*.host77.pingApp.destAddr = "host115"
*.host78.pingApp.destAddr = "host99"
*.host79.pingApp.destAddr = "host55"
*.host80.pingApp.destAddr = "host107"
*.host81.pingApp.destAddr = "host25"
*.host82.pingApp.destAddr = "host68"
*.host83.pingApp.destAddr = "host137"
*.host84.pingApp.destAddr = "host129"
*.host85.pingApp.destAddr = "host51"
*.host86.pingApp.destAddr = "host35"
*.host87.pingApp.destAddr = "host139"
*.host88.pingApp.destAddr = "host54"
*.host89.pingApp.destAddr = "host107"
*.host90.pingApp.destAddr = "host149"
*.host91.pingApp.destAddr = "host91"
*.host92.pingApp.destAddr = "host102"
*.host93.pingApp.destAddr = "host133"
*.host94.pingApp.destAddr = "host68"
*.host95.pingApp.destAddr = "host144"
*.host96.pingApp.destAddr = "host62"
```

```
*.host97.pingApp.destAddr = "host127"
*.host98.pingApp.destAddr = "host106"
*.host99.pingApp.destAddr = "host142"
*.host100.pingApp.destAddr = "host75"
*.host101.pingApp.destAddr = "host33"
*.host102.pingApp.destAddr = "host148"
*.host103.pingApp.destAddr = "host143"
*.host104.pingApp.destAddr = "host23"
*.host105.pingApp.destAddr = "host137"
*.host106.pingApp.destAddr = "host100"
*.host107.pingApp.destAddr = "host67"
*.host108.pingApp.destAddr = "host96"
*.host109.pingApp.destAddr = "host43"
*.host110.pingApp.destAddr = "host122"
*.host111.pingApp.destAddr = "host47"
*.host112.pingApp.destAddr = "host69"
*.host113.pingApp.destAddr = "host35"
*.host114.pingApp.destAddr = "host29"
*.host115.pingApp.destAddr = "host43"
*.host116.pingApp.destAddr = "host86"
*.host117.pingApp.destAddr = "host64"
*.host118.pingApp.destAddr = "host26"
*.host119.pingApp.destAddr = "host141"
*.host120.pingApp.destAddr = "host16"
*.host121.pingApp.destAddr = "host19"
*.host122.pingApp.destAddr = "host77"
*.host123.pingApp.destAddr = "host118"
*.host124.pingApp.destAddr = "host153"
*.host125.pingApp.destAddr = "host145"
*.host126.pingApp.destAddr = "host106"
*.host127.pingApp.destAddr = "host59"
*.host128.pingApp.destAddr = "host116"
*.host129.pingApp.destAddr = "host57"
*.host130.pingApp.destAddr = "host45"
*.host131.pingApp.destAddr = "host36"
*.host132.pingApp.destAddr = "host91"
*.host133.pingApp.destAddr = "host38"
*.host134.pingApp.destAddr = "host23"
*.host135.pingApp.destAddr = "host114"
*.host136.pingApp.destAddr = "host19"
*.host137.pingApp.destAddr = "host123"
*.host138.pingApp.destAddr = "host25"
*.host139.pingApp.destAddr = "host116"
*.host140.pingApp.destAddr = "host11"
*.host141.pingApp.destAddr = "host148"
*.host142.pingApp.destAddr = "host8"
*.host143.pingApp.destAddr = "host81"
*.host144.pingApp.destAddr = "host27"
*.host145.pingApp.destAddr = "host37"
*.host146.pingApp.destAddr = "host124"
*.host147.pingApp.destAddr = "host114"
*.host148.pingApp.destAddr = "host102"
*.host149.pingApp.destAddr = "host150"
*.host150.pingApp.destAddr = "host99"
*.host151.pingApp.destAddr = "host118"
*.host152.pingApp.destAddr = "host14"
*.host153.pingApp.destAddr = "host21"
```

```
*.host154.pingApp.destAddr = "host81"
*.host155.pingApp.destAddr = "host12"
*.host156.pingApp.destAddr = "host10"


*.sas*.eth[*].queueType = "REDQueue"
*.tas*.eth[*].queueType = "REDQueue"
*.tas*.eth[*].mac.txrate = 1Gbps
*.sas*.eth[*].mac.txrate = 10Mbps

**.cmpl.congestion = 30

*.sas*.networkLayer.arp.retryTimeout = .01s
*.tas*.networkLayer.arp.retryTimeout = .01s
*.**.arp.cacheTimeout = 10s
*.host*.networkLayer.arp.retryTimeout = .01s
```

This is an initialization file used in Exp. 3. The main difference between Exp. 2 and this one is the congestion level was added, also the ARP "cacheTimeouts" and "retryTimeouts" have been adjusted.

A.4: OMNET Initialization file for Exp. 4:

```
[General]
**.**.routingDaemon = "Bgpd"
network= Internet
total-stack = 30MB
**.sas2.routingFile = "sas2.irt"
**.sas4.routingFile = "sas4.irt"
**.sas5.routingFile = "sas5.irt"
**.sas6.routingFile = "sas6.irt"
**.sas7.routingFile = "sas7.irt"
**.sas8.routingFile = "sas8.irt"
**.sas9.routingFile = "sas9.irt"
**.sas10.routingFile = "sas10.irt"
**.sas11.routingFile = "sas11.irt"
**.sas12.routingFile = "sas12.irt"
**.sas13.routingFile = "sas13.irt"
**.sas14.routingFile = "sas14.irt"
**.tas0.routingFile = "tas0.irt"
**.tas1.routingFile = "tas1.irt"
**.tas3.routingFile = "tas3.irt"
**.host0.routingFile = "host0.irt"
**.host1.routingFile = "host1.irt"
**.host2.routingFile = "host2.irt"
**.host3.routingFile = "host3.irt"
**.host4.routingFile = "host4.irt"
**.host5.routingFile = "host5.irt"
**.host6.routingFile = "host6.irt"
**.host7.routingFile = "host7.irt"
**.host8.routingFile = "host8.irt"
**.host9.routingFile = "host9.irt"
**.host10.routingFile = "host10.irt"
**.host11.routingFile = "host11.irt"
**.sas2.*.fsroot = "sas2"
**.sas4.*.fsroot = "sas4"
**.sas5.*.fsroot = "sas5"
**.sas6.*.fsroot = "sas6"
**.sas7.*.fsroot = "sas7"
**.sas8.*.fsroot = "sas8"
**.sas9.*.fsroot = "sas9"
**.sas10.*.fsroot = "sas10"
**.sas11.*.fsroot = "sas11"
**.sas12.*.fsroot = "sas12"
**.sas13.*.fsroot = "sas13"
**.sas14.*.fsroot = "sas14"
**.tas0.*.fsroot = "tas0"
**.tas1.*.fsroot = "tas1"
**.tas3.*.fsroot = "tas3"
**.namid = -1
*.sas*.ppp[*].queueType = "REDQueue"
*.tas*.ppp[*].queueType = "REDQueue"
**.host11.pingApp.destAddr = "10.100.28.1"
**.host10.pingApp.destAddr = "10.100.29.1"
**.host9.pingApp.destAddr = "10.100.30.1"
**.host8.pingApp.destAddr = "10.100.31.1"
**.host7.pingApp.destAddr = "10.100.32.1"
```

```
**.host6.pingApp.destAddr = "10.100.33.1"
**.host5.pingApp.destAddr = "10.100.34.1"
**.host4.pingApp.destAddr = "10.100.35.1"
**.host3.pingApp.destAddr = "10.100.36.1"
**.host2.pingApp.destAddr = "10.100.37.1"
**.host1.pingApp.destAddr = "10.100.38.1"
**.host0.pingApp.destAddr = "10.100.39.1"
*.sas*.ppp[*].ppp.timePerFake = .00044800000000000000
*.tas*.ppp[*].ppp.timePerFake = .00004480000000000000
```

This initialization file was used in Exp. 4. The file was generated automatically by the scripts found in Appendix E. Each host and router has an according ".irt" file declaring the IP address of each interface and any initial static routes. Each router, or AS, has an according configuration file for the proper dynamic routing protocol-in this case OSPF.

A.5: OMNET Initialization file for Exp. 5:

```
[General]
network = Network1
fname-append-host = false
output-scalar-file = ${resultdir}/${configname}-${runnumber}.sca
output-vectors-memory-limit = 16MB
record-eventlog = false
ned-path = /home/Adam/inet-framework-inet-
a3307ab/src/.;/home/Adam/inet-framework-inet-a3307ab/Adam/.
cmdenv-express-mode = true
result-dir = 80
sim-time-limit = 12490s
**.module-eventlog-recording = true
**.scalar-recording = true
*.kodiakLan.SHost.pingApp.destAddr = "ua_South.server1"
*.homerLan.standardHost1.pingApp.destAddr = "ua_South.server2"

*.uaa_North.server1.pingApp.destAddr = "ua_South.server3"
*.uaa_North.server4.pingApp.destAddr = "ua_South.server1"
*.uaa_North.server2.pingApp.destAddr = "uaa_North.server5"

**.homerLan.GatewayRouter.ppp[*].ppp.timePerFake = .005

**.router*.eth[*].queueType = "REDQueue"
**.router*.ppp[*].queueType = "REDQueue"
*.ua_South.server.eth[*].queueType = "REDQueue"
*.uaa_North.server3.eth[*].queueType = "REDQueue"
**.**.arp.retryTimeout = .01s
**.**.arp.cacheTimeout = 15s
```

This file was the initialization file for the UAA network in Exp. 5. Only a few hosts were used to run ping times, mainly to compare the difference between the low bandwidth and the high bandwidth connections.

A.6: OMNET Initialization file for Exp. 6:

```
[General]
network=Inet300
total-stack = 20MB
**.sas0.routingFile = "sas0.irt"
**.sas1.routingFile = "sas1.irt"
**.sas2.routingFile = "sas2.irt"
**.sas3.routingFile = "sas3.irt"
**.sas4.routingFile = "sas4.irt"
**.sas5.routingFile = "sas5.irt"
**.sas6.routingFile = "sas6.irt"
**.sas7.routingFile = "sas7.irt"
**.sas8.routingFile = "sas8.irt"
**.sas9.routingFile = "sas9.irt"
**.sas10.routingFile = "sas10.irt"
**.sas11.routingFile = "sas11.irt"
**.sas12.routingFile = "sas12.irt"
**.sas13.routingFile = "sas13.irt"
**.sas14.routingFile = "sas14.irt"
**.sas15.routingFile = "sas15.irt"
**.host6.routingFile = "host6.irt"
**.host1.routingFile = "host1.irt"
**.host2.routingFile = "host2.irt"
**.host4.routingFile = "host4.irt"
**.host5.routingFile = "host5.irt"
**.host3.routingFile = "host3.irt"
**.sas0.*.fsroot = "sas0"
**.sas1.*.fsroot = "sas1"
**.sas2.*.fsroot = "sas2"
**.sas3.*.fsroot = "sas3"
**.sas4.*.fsroot = "sas4"
**.sas5.*.fsroot = "sas5"
**.sas6.*.fsroot = "sas6"
**.sas7.*.fsroot = "sas7"
**.sas8.*.fsroot = "sas8"
**.sas9.*.fsroot = "sas9"
**.sas10.*.fsroot = "sas10"
**.sas11.*.fsroot = "sas11"
**.sas12.*.fsroot = "sas12"
**.sas13.*.fsroot = "sas13"
**.sas14.*.fsroot = "sas14"
**.sas15.*.fsroot = "sas15"
**.host1.pingApp.destAddr = "10.100.45.0"
**.host4.pingApp.destAddr = "10.100.41.0"
*.sas*.eth[*].queueType = "REDQueue"
[Config RIP]
**.**.routingDaemon = "Ripd"
[Config BGP]
**.**.routingDaemon = "Bgpd"
[Config OSPF]
**.**.routingDaemon = "Ospfd"
```

This file was used to initialize Exp. 6. The main focus was host4, which the Complex Module shut down partway through the simulation to test the dynamic routing.

Appendix B: “Complex Module” Source Code

B.1: Complex Module “cmpl.ned”

```
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
// along with this program.  If not, see http://www.gnu.org/licenses/.
//
//

package inet.complex;

simple cmpl
{
    parameters:
              int congestion;
}
```

B.2: Complex Module Header File “cmpl.h”

```
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
// along with this program.  If not, see http://www.gnu.org/licenses/.
//

#ifndef __CMPL_H__
#define __CMPL_H__
#include <iostream>
#include <fstream>
#include <omnetpp.h>
#include "IPvXAddress.h"
#include "PassiveQueueBase.h"
#include <vector>

using namespace std;
class REDQueue;
class cModule;
class cSimulation;

/**
 * Class: cmpl
 * Complex Module Interface between Time Series
 * and OMNET Simulations
 *
 * Extracts all REDQueues from Network and controls
 * amount of FAKE packets in each queue to
 * simulate network congestion
 *
 */
class cmpl: public cSimpleModule {
protected:
   std::vector<std::string> rvec;
   const static int arrsize = 10;
   int counter;
   float cxnumbers[12490][16];

   int count;
   int congestion;


   cTopology testtop;
   cTopology topo2;

   virtual void initialize();
   virtual void handleMessage(cMessage *msg);
   virtual void finish();
   virtual void extractTopology(cTopology& topo);
```

```
public:

        //Updates all REDQueus with new minimum number of
    //Fake packets
    virtual void push();

    //Scedules the next time at which push() will be called
    //DEFAULT is 1 sec
    virtual void scheduleNextPush(cMessage *timer);

   //Deprecated
    virtual void update();
};


#endif
```

B.3: Complex Module “cmpl.cc”

```
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
// along with this program.  If not, see http://www.gnu.org/licenses/.
//

#include "cmpl.h"
#include "REDQueue.h"
#include <stdio.h>
#include <iostream>
#include <fstream>
#include "ARP.h"
using namespace std;

Define_Module(cmpl);

//Initialization Method
//Reads in Congestion Level from .ini file
//Extracts the Network Topology
//Reads time series into array for quicker access at run-time
void cmpl::initialize()
{

    congestion = par("congestion");
    printf("Congestion=%d\n",congestion);
       FILE *fp;

       //Path will have to be Parameter eventually
       fp = fopen("/home/Adam/inet-framework-inet-
a3307ab/src/complex/Data_for_adam.txt", "r");

//Size of time series will need to be a parameter eventually
       int i = 0;
       for (i; i < 12490; i++) {
              int j = 0;
              for (j; j < 16; j++) {
                     fscanf(fp, "%f", &cxnumbers[i][j]);
                     cxnumbers[i][j] = cxnumbers[i][j];
              }
       }


       fclose(fp);

        cTopology topo("topo");


     rvec.push_back("inet.networklayer.queue.REDQueue");

     extractTopology(topo);
     topo2 = topo;
```

```
     topo.clear();

     counter=0;
     cMessage *timer = new cMessage("cmplxMod");
     simtime_t start = SimTime();// + 1.5;
     scheduleAt(start,timer);

}

//Extracts all the REDQueues from the network topology and stores in topo
void cmpl::extractTopology(cTopology& topo){
  topo.extractByNedTypeName(rvec);//topo now holds all the redqueues
}

void cmpl::handleMessage(cMessage *msg)
{

  if (msg->isSelfMessage())
  {
             push();
             scheduleNextPush(msg);
  }
}

void cmpl::push()//updates all of the cQueues
{

   // printf("%d\n",topo2.getNumNodes());

       //For each interface
       for(int i=0;i<topo2.getNumNodes();i++)
       {

              cTopology::Node *node = topo2.getNode(i);

              REDQueue *temp = (REDQueue*)node->getModule();


        float numup = cxnumbers[counter][0]*congestion + (cxnumbers[counter]
[(i+16)%16]*30)-15;
        if(numup<0) numup =0;
        if(temp->type==5){
            temp->updatePackets(numup*1.5);//more fake packets for router.type==5
        }
        else{
             temp->updatePackets(numup);
        }

       }
       counter++;
       if(counter>=12490) counter = 0;//reset counter if run too long to avoid error

printf("\n");

return;
}

void cmpl::scheduleNextPush(cMessage *timer)
{
       simtime_t start = simTime();
                    double interv = 1;
                    scheduleAt(start+interv,timer);
}
```

```
void cmpl::update(){}

void cmpl::finish(){}
```

Appendix C: Modified INET source code

C.1: Modified Source code for "Queue" modules

C.1.1: "PassiveQueueBase.cc"

```
void PassiveQueueBase::handleMessage(cMessage *msg)
{

//ALWAYS ENQUEUE OUTGOING PACKET
    numQueueReceived++;
    if (packetRequested>0)
    {
        packetRequested--;
        bool dropped = enqueue(msg);
           if (dropped){

               numQueueDropped++;
           }

                requestPacket();

    }
    else
    {
        bool dropped = enqueue(msg);
        if (dropped)
            numQueueDropped++;
    }

    if (ev.isGUI())
    {
        char buf[40];
        sprintf(buf, "q rcvd: %d\nq dropped: %d", numQueueReceived, numQueueDropped);
        getDisplayString().setTagArg("t",0,buf);
    }

}

void PassiveQueueBase::requestPacket()
{
    Enter_Method("requestPacket()");

    cMessage *msg = dequeue();
    if (msg==NULL)
    {
        packetRequested++;
    }
    else
    {
        sendOut(msg);
    }
}
```

The two methods "void PassiveQueueBase::handleMessage(cMessage *msg)" and "void PassiveQueueBase::requestPacket()" were modified such that all outgoing traffic from the router was "enqueued", not just in cases where there was a collision.

C.1.2: "REDQueue.h"

```
    void updatePackets(int numPckts){


//calculates the new average for random dropping
       if (queue.empty())
              {

                     double m = SIMTIME_DBL(simTime()-q_time) * pkrate;
                     avg = pow(1 - wq, m) * avg;

              }

       minFakePackets = numPckts;

//if below the minimum number of packets, add more
       if (currFakePackets < minFakePackets) {
              while (currFakePackets < minFakePackets) {
                     EV<< "\n\n\noldQueue Length = " << queue.length() << "\n\n\n";

                     cMessage *mess = new cMessage("FAKE");

                     enqueue(mess);
                     currFakePackets++;
                     EV << "INSERTEDFAKEQUEUE";

                     EV << "\n\n\nnewQueue Length = " << queue.length() << ":" <<
currFakePackets << "\n\n\n";
              }
       }
      timeOfLastReal = simTime();
    }
       void callThis(int i){
                printf("%d successfully called\n",i);
         }
       bool haveRealPackets(){
              if(numRealPackets>0) return true;
              else return false;
       }
```

The "void updatePackets(int nmPckts)" method was added to the REDQueue class. This method is used by the "Complex" Module to tell the REDQueue the minimum number of FAKE packets occupying the queue. These packets make up the background "noise" or congestion level of the simulation.

C.1.3 “REDQueue.cc”

```
void REDQueue::initialize()
{
//"type" parameter isn't really used yet
//A placeholder to define router types for use with Traffic Controller
  type = par("type");

       numRealPackets = 0;//no real packets to start with
    PassiveQueueBase::initialize();
    queue.setName("l2queue");
    //##########VECTORS############
    //Commented these out to speed up
    //uncomment to write values to record data
//    avgQlenVec.setName("avg queue length");
//    qlenVec.setName("queue length");
//    dropVec.setName("drops");



    wq = par("wq");
    minth = par("minth");
    maxth = 50.00;//60.00;//par("maxth");
    maxp = par("maxp");
    pkrate = par("pkrate");
    outGate = gate("out");
    currFakePackets = 0;
    minFakePackets = 0;
    cntrlFake = 0;
    extraFake = 0;
    avg = 0;
    q_time = 0;
    count = -1;
    numEarlyDrops = 0;
    WATCH(avg);
    WATCH(q_time);
    WATCH(count);
    WATCH(numEarlyDrops);

    timeOfLastReal = simTime();
}



bool REDQueue::enqueue(cMessage *msg)
{
       //Check added to insure Queues that weren't fake were dropped at max threshold
       std::string y1;
    y1= msg->getName();
     if(y1.compare("FAKE")==0) {}
     else{
       if(minFakePackets>=(int)maxth){
              delete msg;
            return true;

       }}

       //FLAG to check if the queue is at max threshold
       //if queue is at max, then figure out the number of packets to throw away
         //before attempting to enqueue a packet
       if(currFakePackets >= (int) maxth && currFakePackets > minFakePackets){
              currTime = simTime();
              SimTime elapsed = currTime - timeOfLastReal;
```

```
            //divide by pause period, used .00003 miliseconds (about the time for the
            //eth interface to put a packet on the wire. Not a good implemetation
            //because could be a PPP interface
            SimTime numTimeSteps = elapsed/(.00003);

            //for each time step since last real packet or update, get rid of one
            //fake packet, ASSUMES TIME STEP in CMPL IS 1 second!!!
            for( SimTime i=0;i<numTimeSteps;i = i+1.0)
                                        {
                  if(currFakePackets <= minFakePackets)
                  {
                        break;
                  }
                  else{
                        cMessage *temp = (cMessage *)queue.pop();
                        if (temp != NULL) {

                                    delete temp;

                              avg = (1 - wq) * avg + wq * minFakePackets;

                              currFakePackets--;


                        }

                  }
            }

      }

      if (!queue.empty())//calculates the new average
          {
                    avg = (1-wq)*avg + wq*queue.length();

          }
          else
          {

              double m = SIMTIME_DBL(simTime()-q_time) * pkrate;
              avg = pow(1-wq, m) * avg;

          }

          // statistics, uncomment to record
//          avgQlenVec.record(avg);

          bool mark = false;
          std::string s1;
          s1= msg->getName();
          bool real;
          if(s1.compare("FAKE")==0)real = false;
          if (minth<=avg && avg<maxth)
          {
              count++;
              double pb = maxp*(avg-minth) / (maxth-minth);
              double pa = pb / (1-count*pb);
             double ran = dblrand();
                // if is supposed to be if(dblrand() < pa)
              if (ran < pa)
              {
```

```
                    EV << "Random early packet drop (avg queue len=" << avg << ", pa="
<< pa << ")\n";
                    mark = true;
                    if(s1.compare("FAKE")==0) {
                     mark = false;numEarlyDrops--;
                     }
                    else{
                     printf("\n");
                    }//always enqueue FAKE packets
                    count = 0;
                    numEarlyDrops++;
                }
            }
            else if (maxth <= avg)
            {
                EV << "Avg queue len " << avg << " >= maxth, dropping packet.\n";
                mark = true;
                if(s1.compare("FAKE")==0) mark = false;//always enqueue FAKE packets

                count = 0;
            }
            else
            {
                count = -1;
            }

            //insert all FAKE packets nomatter what
            // carry out decision
            if (mark)
            {
                delete msg;
//uncomment to record
//              dropVec.record(1);

                return true;
            }
            else
            {
                queue.insert(msg);
//uncomment to record
//              qlenVec.record(queue.length());
              if(real == true)numRealPackets++;
                return false;
            }

       }



cMessage *REDQueue::dequeue()
{
       if(numRealPackets<=0){
              return NULL;
       }
       int incr = queue.length(); //go thru each packet only oner
              int i=0;
              bool done = false;
              if(incr <=0) return NULL;
              cMessage *pk = (cMessage *)queue.pop();

              //cMessage dumCheck = *pk;
              std::string s1;
              s1 = pk->getName();
```

```
            while(!done){

            if(i>0){
               if(queue.length()<=0) {
                      if(s1.compare("FAKE")==0)
                             return pk;
                      else
                             return pk;
               }
                      pk = (cMessage *)queue.pop();
                      s1= pk->getName();
                      std::cout << "s1 = " << s1 << "\n";
            }
               if(s1.compare("FAKE")==0){//if fake reenqueue
                  if(minFakePackets<currFakePackets){//if there are more fake packets

                           delete pk;
                       currFakePackets--;

                       cMessage *temp = new cMessage("FAKE");
                            return temp;
                  }
                  else{
                       enqueue(pk);
                            cMessage *temp = new cMessage("FAKE");
                            return temp;
                  }
               }
               else{//real packet so return

                                                done = true;
                                                 if (queue.length()==0)
                                                        q_time = simTime();

                                                    //uncomment to record
//                                                  qlenVec.record(queue.length());
                                                    timeOfLastReal = simTime();
                                 numRealPackets--;
                                                    return pk;
               }
               i++;
            }

      }
```

A few changes needed to be made to make the REDQueue modules compatible with the fake traffic. The “bool enqueue()” method was required to always enqueue fake packets regardless of congestion, while recalculated the new weight used to randomly delete packets. A loop was set to make sure that if enough time had passed between receiving a real packet, the right number of fake packets had “left” and moved on.

The “cMessage dequeue()” method needed to be altered to create new fake packets as old ones left, keeping the queue with the proper minimum number of fake packets. The fake packets which leave the queue will get destroyed by whatever interface, PPP or ETH, is used with the queue.

C.1.4 “REDQueue.ned”

```
package inet.networklayer.queue;

simple REDQueue like OutputQueue
{
    parameters:
        double wq = default(0.002);  // queue weight
        double minth = default(5);  // minimum threshold for avg queue length
        // maximum threshold for avg queue length (=buffer capacity)
        double maxth = default(50);
        double maxp = default(0.02);  // maximum value for pb
        double pkrate = default(150);  // arrivals per sec (see comment above)

        int type = default(0); //type parameter was used for testing with complex
module
        @display("i=block/queue");
    gates:
        input in;
        output out;
}
```

The “REDQueue.ned” class was only slightly altered. The “type” parameter is used only with the “Complex” module and in fact has been deprecated. It is a placeholder for where more information could be passed between the “Complex” module and each interface. It was used in experiment 6 to fail the queue with a certain type.

C.2: ETH Interface

C.2.1: "EtherMAC.cc"

```
void EtherMAC::handleMessage (cMessage *msg)
{
       std::string test = msg->getName();


    if (disabled)
    {
        EV << "MAC is disabled -- dropping message " << msg << "\n";
        delete msg;
        return;
    }
    if (autoconfigInProgress)
    {
        handleAutoconfigMessage(msg);
        return;
    }

    printState();
    // some consistency check
    if (!duplexMode && transmitState==TRANSMITTING_STATE && receiveState!
=RX_IDLE_STATE)
        error("Inconsistent state -- transmitting and receiving at the same time");

//Check for FAKE packets and Handle them
    if(test.compare("FAKE")==0){
//If message came from upperlayer-- This is where they should all come from
       if (msg->getArrivalGate() == gate("upperLayerIn")){
              processFrameFromUpperLayer((EtherFrame *)msg);
       }
//Fake packets should not come from network but handle just in case
       else
              processMsgFromNetwork((cPacket *)msg);
       printState();

           if (ev.isGUI())
               updateDisplayString();
       return;
    }
    if (!msg->isSelfMessage())
    {
        // either frame from upper layer, or frame/jam signal from the network
        if (msg->getArrivalGate() == gate("upperLayerIn"))
            processFrameFromUpperLayer(check_and_cast<EtherFrame *>(msg));
        else
            processMsgFromNetwork(PK(msg));
    }
    else
    {
//IF Message was created from a fake packet, request a new packet from queue
       std::string comp = msg->getName();
       if(comp.compare("REQUEST")==0){
              delete msg;
            beginSendFrames();
       }
       else{
         // Process different self-messages (timer signals)
                    EV << "Self-message " << msg << " received\n";
                    switch (msg->getKind())
```

```
                        {
                                case ENDIFG:
                                        handleEndIFGPeriod();
                                        break;

                                case ENDTRANSMISSION:
                                        handleEndTxPeriod();
                                        break;

                                case ENDRECEPTION:
                                        handleEndRxPeriod();
                                        break;

                                case ENDBACKOFF:
                                        handleEndBackoffPeriod();
                                        break;

                                case ENDJAMMING:
                                        handleEndJammingPeriod();
                                        break;

                                case ENDPAUSE:
                                        handleEndPausePeriod();
                                        break;

                                default:
                                        error("self-message with unexpected message kind %d",
msg->getKind());
                        }
                }
        }
        printState();

        if (ev.isGUI())
            updateDisplayString();
}


void EtherMAC::processFrameFromUpperLayer(EtherFrame *frame)
{
        std::string test = frame->getName();
        EtherMACBase::processFrameFromUpperLayer(frame);
        //FAKE Packets should just return, handled by EtherMACBase
        if(test.compare("FAKE")==0){
           return;
        }

        if (!autoconfigInProgress && (duplexMode || receiveState==RX_IDLE_STATE) &&
transmitState==TX_IDLE_STATE)
        {
            EV << "No incoming carrier signals detected, frame clear to send, wait IFG
first\n";
            scheduleEndIFGPeriod();
        }
}
```

Both methods “void processFrameFromUpperLayer(EtherFrame *fram)” and “void handleMessage()” were altered to handle fake packets. If any fake traffic gets in it should be deleted. If a “REQUEST” message is received, this means the interface should request another packet from the queue.

C.2.2: “EtherMacBase.cc”

```
void EtherMACBase::processFrameFromUpperLayer(EtherFrame *frame)
{
    EV << "Received frame from upper layer: " << frame << endl;


    //CHECK FOR FAKE PACKETS AND HANDLE ACCORDINGLY
    std::string test = frame->getName();
    if(test.compare("FAKE")==0) {
        //create message to request a new packet .00003 seconds from now
       cMessage *msg = new cMessage("REQUEST");
        //again using .00003 for time to process each fake packet in eth interface
        scheduleAt(simTime()+.00003,msg);
       delete frame;

        return;
    }
    if (frame->getDest().equals(address))
    {
        error("logic error: frame %s from higher layer has local MAC address as dest
(%s)",
              frame->getFullName(), frame->getDest().str().c_str());
    }

    if (frame->getByteLength() > MAX_ETHERNET_FRAME)
        error("packet from higher layer (%d bytes) exceeds maximum Ethernet frame size
(%d)", (int)(frame->getByteLength()), MAX_ETHERNET_FRAME);

    // must be EtherFrame (or EtherPauseFrame) from upper layer
    bool isPauseFrame = (dynamic_cast<EtherPauseFrame*>(frame)!=NULL);
    if (!isPauseFrame)
    {
        numFramesFromHL++;

        if (txQueueLimit && txQueue.length()>txQueueLimit)
            error("txQueue length exceeds %d -- this is probably due to "
                  "a bogus app model generating excessive traffic "
                  "(or if this is normal, increase txQueueLimit!)",
                  txQueueLimit);

        // fill in src address if not set
        if (frame->getSrc().isUnspecified())
            frame->setSrc(address);

        // store frame and possibly begin transmitting
        EV << "Packet " << frame << " arrived from higher layers, enqueueing\n";
        txQueue.insert(frame);
    }
    else
    {
        EV << "PAUSE received from higher layer\n";

        // PAUSE frames enjoy priority -- they're transmitted before all other frames
queued up
        if (!txQueue.empty())
            txQueue.insertBefore(txQueue.front(), frame);  // front() frame is
probably being transmitted
        else
            txQueue.insert(frame);
    }
```

```
}

void EtherMACBase::processMsgFromNetwork(cPacket *frame)
{
    EV << "Received frame from network: " << frame << endl;
    std::string test = frame->getName();
//NETWORK should not send a FAKE packet, but don't do anything if it does
//probably should delete any rouge fake packets
    if(test.compare("FAKE")==0)  return;


    // frame must be EtherFrame or EtherJam
    if (dynamic_cast<EtherFrame*>(frame)==NULL &&
dynamic_cast<EtherJam*>(frame)==NULL)
        error("message with unexpected message class '%s' arrived from network
(name='%s')",
                frame->getClassName(), frame->getFullName());

    // detect cable length violation in half-duplex mode
    if (!duplexMode && simTime()-frame->getSendingTime()>=shortestFrameDuration)
        error("very long frame propagation time detected, maybe cable exceeds maximum
allowed length? "
              "(%lgs corresponds to an approx. %lgm cable)",
              SIMTIME_STR(simTime() - frame->getSendingTime()),
              SIMTIME_STR((simTime() - frame->getSendingTime())*SPEED_OF_LIGHT));
}
```

The two methods “void processMsgFromNetwork(cPacket *frame)” and “void processFrameFromUpperLayer(EtherFrame *frame)” were altered to handle fake traffic. In this case, if a fake packet arrived from the upperlayer, a “REQUEST” message was scheduled. When the “REQUEST” message comes in to be handled by “EtherMac.cc,” another packet will be requested from the queue. This delay between receiving the fake packet and the following “REQUEST” simulates the time taken for a FAKE packet to be transmitted.

C.3: PPP Interface

C.3.1: PPP Class file “PPP.h”

```
void PPP::initialize(int stage)
{
    // all initialization is done in the first stage
    if (stage==0)
    {
       //FOR USE WITH FAKE PACKET, see handleMessage() and startTransmitting() methods
       //REQUEST MESSAGES TELL WHEN TO REQUEST A NEW PACKET FROM THE QUEUE TO PUT ON
NET
       timePerFake = par("timePerFake");
        txQueue.setName("txQueue");
        endTransmissionEvent = new cMessage("pppEndTxEvent");
        requestPACKET = new cMessage("REQUEST");

        txQueueLimit = 1000000;//par("txQueueLimit");

        interfaceEntry = NULL;

        numSent = numRcvdOK = numBitErr = numDroppedIfaceDown = 0;
        WATCH(numSent);
        WATCH(numRcvdOK);
        WATCH(numBitErr);
        WATCH(numDroppedIfaceDown);

        // find queueModule
        queueModule = NULL;
        if (par("queueModule").stringValue()[0])
        {
            cModule *mod = getParentModule()-
>getSubmodule(par("queueModule").stringValue());
            queueModule = check_and_cast<IPassiveQueue *>(mod);
        }

        // remember the output gate now, to speed up send()
        physOutGate = gate("phys$o");

        // we're connected if other end of connection path is an input gate
        bool connected = physOutGate->getPathEndGate()->getType()==cGate::INPUT;

        // if we're connected, get the gate with transmission rate
        datarateChannel = connected ? physOutGate->getTransmissionChannel() : NULL;
        double datarate = connected ? datarateChannel->par("datarate").doubleValue() :
0;

        // register our interface entry in IInterfaceTable
        interfaceEntry = registerInterface(datarate);

        // prepare to fire notifications
        nb = NotificationBoardAccess().get();
        notifDetails.setInterfaceEntry(interfaceEntry);
        nb->subscribe(this, NF_SUBSCRIBERLIST_CHANGED);
        updateHasSubcribers();

        // display string stuff
        if (ev.isGUI())
        {
            if (connected) {
                oldConnColor = datarateChannel->getDisplayString().getTagArg("o",0);
            }
```

```
            else {
                // we are not connected: gray out our icon
                getDisplayString().setTagArg("i",1,"#707070");
                getDisplayString().setTagArg("i",2,"100");
            }
        }

        // request first frame to send
        if (queueModule)
        {
            EV << "Requesting first frame from queue module\n";
            queueModule->requestPacket();
        }
    }

    // update display string when addresses have been autoconfigured etc.
    if (stage==3)
    {
        updateDisplayString();
    }
}



void PPP::startTransmitting(cPacket *msg)
{
    // if there's any control info, remove it; then encapsulate the packet

       //First handle fake packets
       std::string test = msg->getName();
       if(test.compare("FAKE")==0) {
              delete msg;

        cMessage *nmsg = new cMessage("REQUEST");

        //SWITCHED FROM THIS METHOD TO timePerFake parameter
        /*
                  simtime_t endTransmissionTime = datarateChannel-
>getTransmissionFinishTime();
            //         std::cout << "endTrsTime = "<<endTransmissionTime<<"\ncurrTime
= "<<simTime()<<"\n";
                  scheduleAt(simTime()+endTransmissionTime, nmsg);
//             queueModule->requestPacket();
 */
        //timePerFake variable determines processing time for FAKE packet on each PPP
interface
           simtime_t endTransmissionTime = timePerFake;
           scheduleAt(simTime()+endTransmissionTime, nmsg);

              return;
       }

       delete msg->removeControlInfo();
    PPPFrame *pppFrame = encapsulate(msg);
    if (ev.isGUI()) displayBusy();

    if (hasSubscribers)
    {
        // fire notification
        notifDetails.setPacket(pppFrame);
        nb->fireChangeNotification(NF_PP_TX_BEGIN, &notifDetails);
    }
```

```
    // send
    EV << "Starting transmission of " << pppFrame << endl;
    send(pppFrame, physOutGate);

    // schedule an event for the time when last bit will leave the gate.
    simtime_t endTransmissionTime = datarateChannel->getTransmissionFinishTime();
    scheduleAt(endTransmissionTime, endTransmissionEvent);
////    simtime_t endTransmissionTime = datarateChannel->getTransmissionFinishTime();
////       std::cout << "endTrsTime = "<<endTransmissionTime<<"\ncurrTime =
"<<simTime()<<"\n";
//    scheduleAt(simTime()+endTransmissionTime, requestPACKET);
}

void PPP::handleMessage(cMessage *msg)
{

       std::string test = msg->getName();
       if(test.compare("FAKE")==0) {
            delete msg;
            //Using timePerFake for processing time, probably a much better way to do
this
            //perhaps using datarateChannel->getTransmissionFinishTime(); or method at
the channel level
            double datarate = timePerFake;

            scheduleAt(simTime()+datarate,new cMessage("REQUEST"));
        }
       else{

          //request a new packet after FAKE packet processing time is through
    if(test.compare("REQUEST")==0){
       queueModule->requestPacket();
       delete msg;
       return;
    }

//     if(msg==requestPACKET){
//
////          if(queueModule->numreal()>0)
////          {
////                     simtime_t endTransmissionTime = datarateChannel-
>getTransmissionFinishTime();
////                //         std::cout << "endTrsTime =
"<<endTransmissionTime<<"\ncurrTime = "<<simTime()<<"\n";
////                     cancelEvent(msg);
////                     scheduleAt(simTime()+endTransmissionTime, requestPACKET);
////          }
//           return;
//     }

    if (datarateChannel==NULL)
    {
        EV << "Interface is not connected, dropping packet " << msg << endl;
        delete msg;
        numDroppedIfaceDown++;
    }
    else if (msg==endTransmissionEvent)
    {
        // Transmission finished, we can start next one.
        EV << "Transmission finished.\n";
        if (ev.isGUI()) displayIdle();

        if (hasSubscribers)
```

```
        {
            // fire notification
            notifDetails.setPacket(NULL);
            nb->fireChangeNotification(NF_PP_TX_END, &notifDetails);
        }

        if (!txQueue.empty())
        {
            cPacket *pk = (cPacket *) txQueue.pop();
            startTransmitting(pk);
            numSent++;
        }
        else if (queueModule)
        {
            // tell queue module that we've become idle
            queueModule->requestPacket();
        }
    }
    else if (msg->arrivedOn("phys$i"))
    {

        if (hasSubscribers)
        {
            // fire notification
            notifDetails.setPacket(PK(msg));
            nb->fireChangeNotification(NF_PP_RX_END, &notifDetails);
        }

        // check for bit errors
        if (PK(msg)->hasBitError())
        {
            EV << "Bit error in " << msg << endl;
            numBitErr++;
            delete msg;
        }
        else
        {
            // pass up payload
            cPacket *payload = decapsulate(check_and_cast<PPPFrame *>(msg));
            numRcvdOK++;
            send(payload,"netwOut");
        }
    }
    else // arrived on gate "netwIn"
    {
        if (endTransmissionEvent->isScheduled())
        {
            // We are currently busy, so just queue up the packet.
            EV << "Received " << msg << " for transmission but transmitter busy,
queueing.\n";
            if (ev.isGUI() && txQueue.length()>=3)
getDisplayString().setTagArg("i",1,"red");

            int check = txQueue.length();
            if (txQueueLimit && txQueue.length()>txQueueLimit)
                error("txQueue length exceeds %d -- this is probably due to "
                      "a bogus app model generating excessive traffic "
                      "(or if this is normal, increase txQueueLimit!)",
                      txQueueLimit);

            txQueue.insert(msg);
        }
        else
```

```
        {
            // We are idle, so we can start transmitting right away.
            EV << "Received " << msg << " for transmission\n";
            startTransmitting(PK(msg));
            queueModule->requestPacket();
//            scheduleAt(simTime()+.5,new cMessage("REQUEST"));
            numSent++;
        }
    }

    if (ev.isGUI())
        updateDisplayString();
}
}
```

The PPP.cc file was altered to handle fake packets. The “void handleMessage(cMessage *msg)” and “void startTransmitting(cPacket *msg)” methods were made to handle FAKE traffic. Much like the Ethernet interfaces, the PPP interfaces delete any FAKE traffic, and issue a REQUEST packet used to get another packet from the queue. Again the lag time between deleting a FAKE packet and waiting for a “REQUEST” simulated the time used to process a FAKE packet. The main difference is that this processing time is read in as a parameter from the OMNET initialization file. This way each PPP interface can have a different processing time per packet.

C.3.2 PPP.ned

```
package inet.linklayer.ppp;

//
// \PPP implementation.
//
// Packets are encapsulated in PPPFrame.
//
// \PPP is a complex protocol with strong support for link configuration
// and maintenance. This model ignores those details, and only performs
// simple encapsulation/decapsulation and queuing.
//
// In routers, \PPP relies on an external queue module (see OutputQueue)
// to model finite buffer, implement QoS and/or RED, and requests packets
// from this external queue one-by-one.
//
// In hosts, no such queue is used, so \PPP contains an internal
// queue named txQueue to queue up packets waiting for transmission.
// Conceptually, txQueue is of infinite size, but for better diagnostics
// one can specify a hard limit in the txQueueLimit parameter -- if this is
// exceeded, the simulation stops with an error.
//
// There is no buffering done on received packets -- they are just decapsulated
// and sent up immediately.
//
// @see PPPInterface, OutputQueue, PPPFrame
//
simple PPP
{
    parameters:

        int txQueueLimit = default(1000);  // only used if queueModule==""; zero means
infinite
        string queueModule = default("");  // name of external (QoS,RED,etc) queue
module
        int mtu = default(4470);

        //A roundabout way to get the time to Process each FAKE Packet, 1ms is default
        //Time is calculated by the nettoini script
        double timePerFake = default(.001);
        @display("i=block/rxtx");
    gates:
        input netwIn;
        output netwOut;
        inout phys @labels(PPPFrame);
}
```

The only addition to the PPP.ned file was to add the parameter “timePerFake” to determine how long each FAKE packet takes to process.

Appendix D: QuaggaRouter Initialization and Configuration Files

D.1 QuaggaRouter Initialization File "RouteFile"

```
IFCONFIG:
name: eth0
inet_addr: 10.100.1.0
Mask: 255.255.255.255
MTU: 1500
Metric: 1
BROADCAST MULTICAST

name: eth1
inet_addr: 10.100.1.1
Mask: 255.255.255.255
MTU: 1500
Metric: 1
BROADCAST MULTICAST

IFCONFIGEND.

ROUTE:
10.100.1.1 * 255.255.0.0 H 1 eth0
11.100.1.2 * 255.255.255.255 H  1 eth0
11.100.1.2 * 255.255.255.0 H  1 eth0
10.100.1.4 * 255.255.255.255 H 1 eth1
ROUTEEND.
```

Each "QuaggaRouter" and "StandardHost" module in every simulation needed one of the above files. They defined the network address for each interface and declared the initial routes used to talk to the neighboring module.

D.2 QuaggaRouter Configuration file: RIP Protocol

```
hostname sas2
password zebra
debug rip packet
debug rip events
debug rip zebra
log stdout
router rip
 network eth0
 network eth1
 network eth2
redistribute connected
redistribute kernel
```

D.3 QuaggaRouter Configuration file: OSPF Protocol

```
hostname sas2
password zebra
debug ospf packet all send
debug ospf packet all recv
debug ospf ism
debug ospf nsm
debug ospf lsa
debug ospf zebra
log stdout
router ospf

 ospf router-id 10.100.5.1
 ospf rfc1583compatibility
network 10.100.1.0/24 area 0.0.0.1
network 10.100.2.0/24 area 0.0.0.1
network 10.100.3.0/24 area 0.0.0.1
network 10.100.4.0/24 area 0.0.0.1
network 10.100.5.0/24 area 0.0.0.1
network 10.100.6.0/24 area 0.0.0.1
redistribute connected
redistribute kernel
redistribute static
interface eth0
  ip ospf cost 1
  ip ospf priority 1
interface eth1
  ip ospf cost 1
  ip ospf priority 1
interface eth2
  ip ospf cost 1
  ip ospf priority 1
interface eth0
  ip ospf retransmit-interval 5
  ip ospf transmit-delay 1
  ip ospf dead-interval 40
  ip ospf hello-interval 10
interface eth1
  ip ospf retransmit-interval 5
  ip ospf transmit-delay 1
  ip ospf dead-interval 40
  ip ospf hello-interval 10
interface eth2
 ip ospf retransmit-interval 5
 ip ospf transmit-delay 1
  ip ospf dead-interval 40
  ip ospf hello-interval 10
```

D.4 QuaggaRouter Configuration file: BGP Protocol

```
hostname sas2
password zebra
debug bgp events
debug bgp filters
debug bgp fsm
debug bgp keepalives
debug bgp updates
log stdout
router bgp 1
 neighbor 10.100.1.1 remote-as 1
 neighbor 10.100.2.0 remote-as 1
 neighbor 10.100.5.0 remote-as 1
redistribute connected
```

Appendix E: Automated Scripts for simulation

E.1: ReaSE to INET-Quagga

```
#! /bin/bash
#TAKES A NETWORK CREATED BY REASE (ONLY AS LEVEL, NO ROUTER LEVEL)

#first argument is reaseNetwork
REASENET=$1

#second argument is target network
TARGETNET=$2

#third argument is package
PACKAGE=$3

#fourth is Network Name
NETWORK=$4

echo "REASENET = $REASENET
TARGETNET = $TARGETNET
PACKAGE = $PACKAGE"

#CREATE NEW NED FILE, WRITE HEADER AND PACKAGE INFO
echo "package $PACKAGE;
import org.omnetpp.inet.networklayer.quagga.QuaggaRouter;
import inet.complex.cmpl;
import inet.nodes.inet.StandardHost;
">$TARGETNET

#PRINT CHANNELS
NUMCHAN=0
grep channel $REASENET | while read channel
do
NUMDEL=0
NUMRATE=0
echo "$channel
{
   parameters:">>$TARGETNET


grep "delay =" $REASENET | while read delay
do
if [ $NUMDEL = $NUMCHAN ]; then
echo "      $delay">>$TARGETNET
fi
let NUMDEL+=1
done
grep "datarate =" $REASENET | while read datarate
do
if [ $NUMRATE = $NUMCHAN ]; then
echo "      $datarate">>$TARGETNET
fi
let NUMRATE+=1
done
echo "}">>$TARGETNET
let NUMCHAN+=1
done


#DEFINE NETWORK MODULE
```

```
echo "module MOD_$NETWORK
{
   parameters:
      @MOD_$NETWORK();
   submodules:">>$TARGETNET


#GET ROUTER MODULES AND HOSTS
HOST=0
grep sas[0-9]*: $REASENET | cut -d: -f1| while read sas
do
  echo "      $sas: QuaggaRouter;">>$TARGETNET
  echo "      host$HOST: StandardHost;">>$TARGETNET
  let HOST+=1
done


grep tas[0-9]*: $REASENET | cut -d: -f1| while read tas
do
  echo "      $tas: QuaggaRouter;">>$TARGETNET
done


#PUT IN CONNECTIONS

echo "connections:">>$TARGETNET

grep "<-->" $REASENET | while read conn
do
echo "      $conn" >>$TARGETNET
done

HOST=0
grep sas[0-9]*: $REASENET | cut -d: -f1 | while read sas
do
echo "host$HOST.pppg++ <-->stub2stub <--> $sas.pppg++;">>$TARGETNET
let HOST+=1
done

echo "}

network $NETWORK extends MOD_$NETWORK
{
   parameters:
}">>$TARGETNET

delte $REASENET
```

This script will take in a network created in ReaSE and created the same network using QuaggaRouters from INET-Quagga.

E.2: NET to Quagga Configurations

```
#!/bin/bash

#SCRIPT TO CREATE NECESSARY QUAGGA INITIALIZATION FILES
#CREATES A .irt FOR EACH HOST AND ROUTER
#CREATES AN FSROOT DIRECTORY FOR EACH ROUTER
#EACH FSROOT CONTAINS zebra,ripd,ospfd, and bgp .conf files

#TAKES IN PATH TO NED FILE AS PARAMETER

rm *.rid
rm netListTAS
rm netListSAS
rm netList
rm *.route
rm *.ifnumber
rm *.ifn
rm -rf ConnectionArrays
mkdir ConnectionArrays
mkdir INIFiles
rm MODULES

rm ROUTERMODULES

NEDFILE=$1
echo "NEDFILE = $NEDFILE"

NET=`grep network[\ ] [0-9].*.ned | cut -d\  -f2`
NETWORK=0
TASNET=0

echo 'Network =' $NET
echo $INT

# Store routers in file
#  grep *** | cut *** | tr***
grep router[0-9]*: $NEDFILE | cut -d: -f1 | tr -d [:blank:] > ROUTER
grep router[0-9]*: $NEDFILE | cut -d: -f1 | tr -d [:blank:] >> MODULES
grep router[0-9]*: $NEDFILE | cut -d: -f1 | tr -d [:blank:] >> ROUTERMODULES

# store SAS in file
#  grep *** | cut *** | tr***
grep sas[0-9]*: $NEDFILE | cut -d: -f1 | tr -d [:blank:] > SAS
grep sas[0-9]*: $NEDFILE | cut -d: -f1 | tr -d [:blank:] >> MODULES
grep sas[0-9]*: $NEDFILE | cut -d: -f1 | tr -d [:blank:] >> ROUTERMODULES

# store TAS in file
#  grep *** | cut *** | tr***
grep tas[0-9]*: $NEDFILE | cut -d: -f1 | tr -d [:blank:] > TAS
grep tas[0-9]*: $NEDFILE | cut -d: -f1 | tr -d [:blank:] >> MODULES
grep tas[0-9]*: $NEDFILE | cut -d: -f1 | tr -d [:blank:] >> ROUTERMODULES

# store Host in file
#  grep *** | cut *** | tr***
grep Host[0-9]*: $NEDFILE | cut -d: -f1 | tr -d [:blank:] > HOST
grep Host[0-9]*: $NEDFILE | cut -d: -f1 | tr -d [:blank:] >> MODULES

grep host[0-9]*: $NEDFILE | cut -d: -f1 | tr -d [:blank:] > HOST
grep host[0-9]*: $NEDFILE | cut -d: -f1 | tr -d [:blank:] >> MODULES

# store standardHost in file
```

```
#  grep *** | cut *** | tr***
grep standardHost[0-9]*: $NEDFILE | cut -d: -f1 | tr -d [:blank:] > STANDARDHOST
grep standordHost[0-9]*: $NEDFILE | cut -d: -f1 | tr -d [:blank:] >> MODULES


# reset Network
# initialize all .irt files for sas and host modules
let NETWORK=0
grep sas.* SAS | while read line
do
echo "ifconfig:" > $line.irt
done

let NETWORK=0
grep tas.* TAS | while read line
do
echo "ifconfig:" > $line.irt
done

let NETWORK=0
grep h.* HOST | while read line
do
echo "ifconfig:" > $line.irt
done

#Go through each connection array for each interface and make subnet
#echo "name: ppp$NUM
#inet_addr: 10.100.$THIRDBYTE.$FOURTHBYTE
#Mask: 255.255.255.0
#MTU: 1500
#Metric: 1
#BROADCAST MULTICAST
#">>$sas.irt

#For each connection, will append an ip address to ead ifconfig files, and will keep
track of number of interface of each router in a file
LINE=0
THIRDBYTE=0
FOURTHBYTE=0
grep "<-->" $NEDFILE | cut -d. -f2 | cut -d\  -f5 | while read RIGHTSIDE
do
let THIRDBYTE+=1
FOURTHBYTE=0
echo "Line1 $LINE RIGHTSIDE: $RIGHTSIDE"
LINE2=0
grep "<-->" $NEDFILE | cut -d. -f1 | tr -d [:blank:] | while read LEFTSIDE
do
  if [ $LINE = $LINE2 ]; then
    echo "Line2 $LINE2 LEFSIDE: $LEFTSIDE"
   let LINE2+=1
   let FOURTHBYTE+=1
   #append to $LINE2.ifnumber
   #then read lines until get ifnumber, do same down below
   echo "1">>$LEFTSIDE.ifnumber
   IFNUM=0;
   grep [0-9].* $LEFTSIDE.ifnumber | while read line
   do
   let IFNUM+=1
   echo "$IFNUM">$LEFTSIDE.ifn
   done

   grep [0-9].* $LEFTSIDE.ifn | while read pppnum
   do
```

```
   let pppnum+=-1
   echo "name: ppp$pppnum
   inet_addr: 10.100.$THIRDBYTE.$FOURTHBYTE
   Mask: 255.255.255.0
   MTU: 1500
   Metric: 1
   POINTTOPOINT MULTICAST">>$LEFTSIDE.irt
   echo "10.100.$THIRDBYTE.$FOURTHBYTE">$LEFTSIDE.rid
   let FOURTHBYTE+=-1
   #now write routes to route file to append later
   echo "10.100.$THIRDBYTE.$FOURTHBYTE * 255.255.255.255 H 1 ppp$pppnum" >>
$LEFTSIDE.route
   done
   let LINE2+=-1
  fi
let LINE2+=1
done
   echo "1">>$RIGHTSIDE.ifnumber

   IFNUM=0;
   grep [0-9].* $RIGHTSIDE.ifnumber | while read line
   do
   let IFNUM+=1
   echo "$IFNUM">$RIGHTSIDE.ifn
   done

   grep [0-9].* $RIGHTSIDE.ifn | while read pppnum
   do


   let pppnum+=-1
   echo "name: ppp$pppnum
   inet_addr: 10.100.$THIRDBYTE.$FOURTHBYTE
   Mask: 255.255.255.0
   MTU: 1500
   Metric: 1
   POINTTOPOINT MULTICAST">>$RIGHTSIDE.irt
   echo "10.100.$THIRDBYTE.$FOURTHBYTE">$RIGHTSIDE.rid
   let FOURTHBYTE+=1
   echo "10.100.$THIRDBYTE.$FOURTHBYTE * 255.255.255.255 H 1 ppp$pppnum" >>
$RIGHTSIDE.route
   echo "10.100.$THIRDBYTE.0">>netList
   done
let LINE+=1
done

#add default route to host modules
grep [0-9].* HOST | while read host
do
echo "default: * 0.0.0.0 H 0 ppp0" >> $host.route
done


#finish .IRT files for each MODULE
grep [0-9].* MODULES | while read module
do
echo "ifconfigend.

route:">> $module.irt
grep [0-9].* $module.route | while read route
do
echo "$route">> $module.irt
done
```

```
echo "routeend." >> $module.irt
done

#####################################################
#WRITE QUAGGA CONFIG FILES FOR EACH ROUTER

#RIP
grep [0-9].* ROUTERMODULES | while read router
do
mkdir $router
echo "hostname $router
password zebra
debug rip packet
debug rip events
debug rip zebra
log stdout
router rip  ">$router/_etc_quagga_ripd.conf

LINE=0
grep .*[0-9] $router.ifnumber | while read iface
do
echo " network ppp$LINE">>$router/_etc_quagga_ripd.conf
  let LINE+=1

done
echo "redistribute connected
redistribute kernel">>$router/_etc_quagga_ripd.conf
echo "19">$router/_var_run_quagga_ripd.pid
echo "hostname $router
password zebra
enable password zebra
log stdout
" > $router/_etc_quagga_zebra.conf
echo "19"> $router/_var_run_quagga_zebra.pid
done

#BGP
grep [0-9].* ROUTERMODULES | while read line
do
  mkdir $line
  echo "hostname $line
password zebra
debug bgp events
debug bgp filters
debug bgp fsm
debug bgp keepalives
debug bgp updates
log stdout
router bgp 1
  ">$line/_etc_quagga_bgpd.conf
  lin=0
  grep [0-9].* $line.route | cut -d\  -f1 | while read ip
  do
     echo " neighbor $ip remote-as 1">>$line/_etc_quagga_bgpd.conf
  let lin+=1
  done
echo "redistribute connected">>$line/_etc_quagga_bgpd.conf
echo "19">$line/_var_run_quagga_bgpd.pid

done

#OSPF
grep [0-9].* ROUTERMODULES | while read line
```

```
do
  mkdir $line
  echo "hostname $line
password zebra
debug ospf packet all send
debug ospf packet all recv
debug ospf ism
debug ospf nsm
debug ospf lsa
debug ospf zebra
log stdout
router ospf
  ">$line/_etc_quagga_ospfd.conf
grep [0-9].* $line.rid | while read line1
do
echo " ospf router-id $line1">>$line/_etc_quagga_ospfd.conf

done
echo " ospf rfc1583compatibility">>$line/_etc_quagga_ospfd.conf
grep [0-9].* netList | while read network
do
echo "network $network/24 area 0.0.0.1">>$line/_etc_quagga_ospfd.conf
done
echo "redistribute connected">>$line/_etc_quagga_ospfd.conf
echo "redistribute kernel">>$line/_etc_quagga_ospfd.conf
echo "redistribute static">>$line/_etc_quagga_ospfd.conf

pppNUM=0
grep [0-9].* $line.ifnumber | while read num
do
let pppNum+=1
let pppNum+=-1
echo "interface ppp$pppNum
  ip ospf cost 1
  ip ospf priority 1">>$line/_etc_quagga_ospfd.conf

let pppNum+=1
done

pppNUM=0
grep [0-9].* $line.ifnumber | while read num
do
let pppNum+=1
let pppNum+=-1
echo "interface ppp$pppNum
  ip ospf retransmit-interval 5
  ip ospf transmit-delay 1
  ip ospf dead-interval 40
  ip ospf hello-interval 10">>$line/_etc_quagga_ospfd.conf

let pppNum+=1
done
done

#Creates an incomplete sample INI FILE
#echo "[General]"> tester.ini
#grep [0-9].* MODULES | while read mod
#do
#echo "**.$mod.routingFile = \"$mod.irt\""">>tester.ini
#done
#grep [0-9].* ROUTERMODULES | while read router
#do
#echo "**.$router.*.fsroot = \"$router\""">>tester.ini
```

```
#done


#echo "**.namid = -1">>tester.ini

#grep [0-9].* HOST | while read host
#do
#echo "**.$host.pingApp.destAddr = \"10.100\"">>tester.ini
#done

#echo "*.sas*.ppp[*].queueType = \"REDQueue\"">>tester.ini
#echo "*.tas*.ppp[*].queueType = \"REDQueue\"">>tester.ini

#echo "[Config RIP]
#**.**.routingDaemon = \"Ripd\"">>tester.ini

#echo "[Config BGP]
#**.**.routingDaemon = \"Bgpd\"">>tester.ini

#echo "[Config OSPF]
#**.**.routingDaemon = \"Ospfd\"">>tester.ini

#DELETE UNNECESSARY FILES
rm *.ifn
rm *.route
rm *.ifnumber
rm *.rid
```

This script will take a network defined in an NED file and create the necessary INET-Quagga initialization files to use dynamic routing. The script only accepts PPP interfaces, since these are the interfaces ReaSE uses when creating topologies.

E.3: Create a default *.ini file to run ping tests

```
#! /bin/bash
#

#CREATES A DEFAULT INITIALIZATION FILE
#READY TO RUN PING TIMES AT MED CONGESTION

#SPECIFY NAME OF NEW .INI FILE
INIFILE=$1

#ROUTING DAEMON WILL THIS INI FILE RUN
DAEMON=$2

#NETWORK NED FILE WILL THIS INI RUN
NED=$3

echo "FILE = $INIFILE"

echo "[General]">$INIFILE
if [ $2 = "RIP" ]; then
echo "**.**.routingDaemon = \"Ripd\""">>$INIFILE
fi
if [ $2 = "BGP" ]; then
echo "**.**.routingDaemon = \"BGPd\""">>$INIFILE
fi
if [ $2 = "OSPF" ]; then
echo "**.**.routingDaemon = \"Ospfd\""">>$INIFILE
fi

grep "network " $NED | cut -d\  -f2 | while read net
do
echo "network= $net">>$INIFILE
done

echo "total-stack = 30MB">>$INIFILE


grep [0-9].* MODULES | while read mod
do
echo "**.$mod.routingFile = \"$mod.irt\""">>$INIFILE
done
grep [0-9].* ROUTERMODULES | while read router
do
echo "**.$router.*.fsroot = \"$router\""">>$INIFILE
done


echo "**.namid = -1">>$INIFILE


echo "*.sas*.ppp[*].queueType = \"REDQueue\""">>$INIFILE
echo "*.tas*.ppp[*].queueType = \"REDQueue\""">>$INIFILE

#create ping tests
HOST=0

grep [0-9].* HOST | while read host
do
echo "$HOST">numhost
let HOST+=1
done
```

```
grep [0-9].* numhost | while read numhosts
do
HOST=$numhosts
grep [0-9].* HOST | while read host
do
grep inet_addr $host.irt | cut -d: -f2 | while read ip
do
echo "**.host$HOST.pingApp.destAddr = \"$ip\""">>$INIFILE
done
let HOST+=-1
done
done

#ORDER SHOULD BE STUB TO STUB, TRANS TO TRANS, STUB TO TRANS, calculate time for FAKE
Packet
grep "datarate = " $NED | cut -d= -f2 | cut -dM -f1 | tr -d [:blank:] | while read
datarate
do
NUM=0
echo "56*8/($datarate * 1000)" | bc -l > float
grep [0-9].* float | while read flt
do
if [ $NUM = 0 ]; then
echo "*.sas*.ppp[*].ppp.timePerFake = $flt">>$INIFILE
fi
if [ $NUM = 1 ]; then
echo "*.tas*.ppp[*].ppp.timePerFake = $flt">>$INIFILE
fi
done
let NUM+=1
done
```

This script takes a network assuming all the Quagga files exist, and creates an OMNET initialization file ready to execute ping tests between hosts. The OMNET file will have only one dynamic routing protocol defined, so this script may be run multiple times to configure for different protocols.

Appendix F: User Manual

Welcome to the "NETWORK SIMULATOR" user manual. This short manual is designed to take you through the steps of installing all of the necessary tools, and a brief tutorial on running simulations.

CH 1. Installation

This chapter takes you through the steps of installing the NETWORK SIMULATOR on both Ubuntu and Fedora operating systems. It assumes that you start with a fresh operating system, and the compressed file ADAMPOJ_V_1.0 in either a .tar or .zip format. If the operating system already has many packages installed, then some of the steps may be skipped.

OMNET++ Installation

The first and usually trickiest part is the installation of OMNET++ version 4.1. A copy of OMNET is found in the ADAMPROJ_V_1.0 compressed file. This step is the only part that may

be different whether using Ubuntu or Fedora. To install on other flavors of Linux or for more help in this part of the installation, go to http://omnetpp.org/doc/omnetpp/InstallGuide.pdf for another written installation guide, or check out http://www.youtube.com/watch?v=gz0BKhrbbXQ for a video tutorial.

Installation on Ubuntu (10.04 and 11.04)

The first step is to install the packages necessary to run OMNET. Open a terminal (Applications → Accessories → Terminal) and go through the following steps:

Step 1: Installing necessary packages

Update apt-get database (don't type the dollar sign)

$ sudo apt-get update

Install necessary packages. Issue the following command and answer *Y* when prompted if you are sure you want to install all the packages.

$ sudo apt-get install build-essential gcc g++ bison flex perl tcl-dev tk-dev blt libxml2-dev zlib1g-dev openjdk-6-jre doxygen graphviz openmpi-bin libopenmpi-dev libpcap-dev

Step 2: Configuration

If you haven't already done so, unzip the ADAMPROJ_V1 file and change into the directory: omnetpp-4.0p1

UNZIP if in .zip format:
$ gunzip ADAMPROJ_V1.zip
Or if it is in .tar:
$ tar –xvf ADAMPROJ_V1.tar
Change into the omnet directory:
$ cd ADAMPROJ_V1/omnetpp-4.0p1
Once you are in the directory, issue the following command:
$. setenv
Next, open your bashrc file in any text editor, for example using nano:
$ nano ~/.bashrc
Add the following line to the end of your bashrc file:

export PATH=$PATH:$HOME/omnetpp/omnetpp-4.0p1/bin

export PATH=$PATH:$HOME/Desktop/omnetpp-4.0p1

Save the new bashrc file. Restart your terminal and change back into the omnetpp-4.0p1 directory. Once in the directory, issue the following command:

$ ./configure

If all of the necessary packages from *Step 1* were installed, then there shouldn't be any errors. If the output tells you to add any additional lines to your .profile or .bash_profile, add the lines to the bottom of your .bashrc file, same way as before:

$ nano ~/.bashrc

Make sure to restart your terminal if you altered your .bashrc file and change back into the omnetpp-4.0p1 directory. If you altered your .bashrc file reissue the *./configure* command in the omnetpp-4.0p1 directory.

Step 3 Building OMNET

Inside the OMNET directory, issue one of the following commands:

$ make MODE=release

$ make MODE=debug

Depending on whether or not you want the release version or the debug. The release version is recommended unless you are planning on altering much of the source code. If any errors occur during this phase, they are most likely due to the TCL library. Refer to the OMNET installation guide online to run through more troubleshooting issues.

To start the IDE, you can issue the following command:

$ omnetpp

Select a workspace:

/home/test/Desktop/omnetpp-4.0p1/

F.2 Installation of INET

If you are still in the OMNET directory go back to the ADAMPROJ directory:

$ cd ..

To install INET first extract the components from the compressed file:

$ tar –xfj inet-framework-inet-a3307ab

Now change into the INET directory

$ cd inet-framework-inet-a3307ab

Build the INET framework

$ oppmakemake

$ cd src

$ make

To install INET graphically, open the OMNET IDE:

$ omnetpp

Delete any older versions of INET in the IDE (right click → delete)

Import INET into the workspace (File → Import → Existing Projects into Workspace)

Select the directory containing INET (probably ADAMPROJ)

Only check **inet**, make sure "copy projects into workspace" is checked.

To check if INET is successfully installed try running a program with the command

$ "command to run program"

If not try opening a new terminal

F.3 Installation of INET-Quagga

To install INET first extract the components from the compressed file:

$ tar –xfj inet-framwork-ajlkjadsg;aj;aj

Now change into the INET directory

$ cd inet-asdgkajg;jdfkljh

Configure and build the INET framework

$ ./configure

$ make

F.4 Installation of ReaSE

F.5 Step by step ReaSE to Simulation to Data

Step 1, Create Network

Change into the directory with the ReaSEGUR.jar file:

$ cd ReaSEGUI/ReaSEGUI/dist/

Execute the jar file with:

$ java –jar alskdg.jar

This activates the ReaSE GUI for creating topologies. You will first need to declare the path to the tgm:

DECLARE TGM PATH

The tgm by default is in the ReaSEGUI/TGM/ directory. Select this path as your tgm path. Once the tgm path is set, select the name of the output topology.ned file. You can select to output the topology to any directory you wish. Now select the parameters used to create the network. Make sure router-level is unselected, as this network tool will be simulating at the AS level exclusively. When you click run, you may be prompted to save your network. Save your topology in whatever directory you want, with the desired name of the file.

Congratulations! your first network has been constructed using ReaSE.

Step 2, Create Quagga Network

The next step towards running your first simulation will be to translate the network from ReaSE over to a network with QuaggaRouters. For this step, copy the script entitled: "reasetoquagga" from the "scripts" folder into the same directory as your ReaSE network. Execute the script:

$ ./reasetoquagga REASE.ned OUTPUT.ned PACKAGENAME NETWORKNAME

Replace the "REASE.ned" with the name of the ned file created by ReaSE. OUTPUT.ned should be replaced by whatever you want your new file to be called. PACKAGENAME should just be the name of the directory which you are in. NETWORKNAME should be replaced by a unique name for your network.

Step 2 is now complete. See "Troubleshooting" at the end for some common problems associated with this step.

Step 3, Create Quagga Files

To create all of the Quagga initialization files for each module, you must execute the "ppptest" script. This script assumes all of the interfaces are PPP and not ETH. This assumption is made

because ReaSE generates PPP interfaces at the AS level. Execute the script with:

$ ./ppptest TOPOLOGY.ned

The only parameter this script needs is the name of your topology. This should be the file you created with the script in step 2.

Step 3 is now complete. See "Troubleshooting" at the end for some common problems associated with this step.

Step 4, Create a default Initialization file

To run an OMNET simulation, you will need not only the .ned file, but also an initialization file, or .ini, to outline the necessary rum-time parameters. The script "nettoini" will create a default initialization file, ready to be executed to run pings between the hosts on the network.

$ ./nettoini INIFILENAME ROUTINGDAEMON NEDFILE

The first parameter will be the name of the initialization file you are creating. The ROUTINGDAEMON can be either Ripd or Ospfd, at this point Bgpd is not fully supported. The final parameter is the .ned file with the description of the network for which you would like to run simulations.

Step 4 is now complete. You now should have a default initialization file to run simulations. You will most likely want to run many simulations with all different parameters. See the section "INI Files" to learn more about various parameters and how to alter them.

Step 5 Analyzing Data

The OMNET documentation provides a lot of information on the format of the data outputted. The path to the directory to store the data can be set in the initialization file

Result-dir = "PATH/TO/RESULTS"

The default is to store the data in the "Results" directory. To analyze the data in the OMNET IDE, go the directory storing the results and right click, selecting "new analysis file (anf)." This will create a new file to analyze your results. Data can be exported to various formats by right clicking and selecting "export as," which can then be analyzed using various other tools.

PART 3, Troubleshooting guide

Troubleshooting Installation:

OMNET

OMNET can be tricky to install. Since the errors can be so vast, the most help I can say here is to check out the OMNET website for help installing. The version of OMNET included in the compressed file is version 4.0, which can also be downloaded from their website.

The part which was most-likely screwed up during installation was the installation of every required package required by OMNET. A few tricks if OMNET isn't working:

If you don't want or need the GUI associated with OMNET runs, issue the line:

$ NO_TCL=true

Before configuring and running. The TCL library is often a cause of failure during the building of OMNET, and if you aren't going to use it, there's no reason to include it in the building.

Another issue is that your packet manager may not install every component required by OMNET. If there is a way for you to graphically install the packages through the package manager, do it this way. Along the way make sure to install any packages recommended to you by the package manager at any step. This should help to insure that all the necessary packages get installed on your system.

INET

The INET installation should go smoothly, if it doesn't, the best recommendation is to install the package through the OMNET IDE (Eclipse). Choose

FILE→ IMPORT → EXISTING PROJECTS INTO WORKSPACE

Make sure that only INET is checked, and import it into the workspace. Then build the project with ctrl+b. This should help to insure that INET is properly linking the all the necessary libraries needed to compile.

INET-Quagga

Again, the INET-Quagga installation should be handled in a similar manner as INET. If the project doesn't build, make sure you are linking to the correct and necessary libraries, one way of doing this is by building through the Eclipse IDE, as outlined in the steps above under INET.

ReaSE

ReaSE is a bit trickier to install. First change into the directory TGM and try to make. If this works, ReaSE can at least be used to create topologies without the GUI. Just run the "tgm" script with a param file (there is a sample one in the TGM directory). Make sure to only create network topologies at the AS level, others may not be compatible with INET-Quagga or the automation scripts.

Troubleshooting Network Creation:

Assuming that the networks were properly created through ReaSE, the scripts should create all the necessary files for running simulations without error. There are however, a few things to make sure the networks can run.

PACKAGE

The package must be set correctly. This is the first line in the network .ned file. Typically this will just be the name of the directory which the network is in, but in some cases you will need a longer path name for your package. Using the IDE, it will usually tell you what the package name should be in an error report. Also if you get a run-time error with an incorrect

package, the error outputted should say that the ned package doesn't match the expected package and will usually output what the package name is expected to be.

NETWORK AND MODULE NAME

Network and module names must be unambiguous. If you have two ned files defining networks with the same name, this can cause errors if they are within the same ned path. This problem should never happen unless you try to create a network with the same name as one which has been created.

NON-UNIQUE IP ADRESSES

The script is assuming the use of only 512 interfaces in the entire network. If your network is too large, modify the script to handle more interfaces. Other than that you can manually change the IP and network addresses of each interface in the proper .irt files. If you do this make sure to manually update any necessary Quagga-conf files.

Troubleshooting Initialization Errors:

NO NETWORK OR DUAL NETWORK

Each ini file can only run simulations for one network. If the file is having trouble, make sure the network you are trying to run simulations on exists, and also make sure it is unique (see NETWORK AND MODULE NAME) above in the "troubleshooting network creation" section.

UNSET PARAMETERS

If any parameter, defined in .ned files, of any module does not have a *default* value, then the parameter *must* be set in the initialization file. If the parameter is not set, then you will be prompted to set the parameter before the simulation starts if trying to run through the Tkenv GUI, but through the command-line interface, the simulation will not run. Make sure your parameters are set and set right. See PART 4 for more information on simulation parameters.

Troubleshooting Run-Time errors:

Run time errors can be caused by a great many issues. The simulations as set up by the automation scripts have not had any run time errors, but as you change parameters you may run into a few. These can often be solved using the IDE debugging tool, but that takes some practice learning how to use most effectively. Often times the problems are with parameters set to default values which are unreasonable, or will set the simulation up to fail. Many problems with INET and OMNET can be searched for effectively online, as both are large open-source projects. INET-Quagga, ReaSE, and this tool are smaller and have less documentation and support. The best bet is to figure out what parameter caused what error, and use some kind of debugging tool to find what the problem is.

PART 4, INI Files and run time parameters

COMPLEX MODULE

The complex module "cmpl.ned" has a few necessary parameters in order to run. The complex module expects a file of numbers between 0 and 1. The numbers should be in a row and column format. The path to this file must be set with the line:

**.cmpl.file = “PATH/TO/file.txt”

Along with this file, the module will need to know what the number of rows and columns are in order to properly read the file. Each column should represent a time series. A more involved discussion of the complex module is found in part 5. Set number of rows and columns with:

**.cmpl.rows = number of rows
**.cmpl.cols = number of cols

The values should be set based on input to the nettoini file, but if you change your time series, you can just rewrite these values in the .ini file.

PING TIMES

The hosts can ping any valid IP address on the network. The pingApp has many parameters which can be changed, for instance the time between pings can be increased or decreased, the default value is one second. For more information about the pingApp, see the INET documentation at www.INET.com

TCP/UDP Protocol

INET supports both TCP and UDP communication traffic, none of which has been tested with this model. The INET documentation should provide you with information on how to run TCP and UDP traffic through your simulation.

SIMULATION LENGTH

The duration of the simulation can be set with:

Simulation-length = #s

RECORD VECTORS

Scalar and vector recording can be turned on and off:

Record-event-vector = true
Record-event-vector=false

ARP TIMEOUTS

The ARP protocol was found to have a direct effect on the nonlinear dynamics of the system, so playing around with ARP requests and timeouts is probably important.

..arp.Cache_timeout = 10s
..arp.request_timeout = .5s

There are many more parameters in the ARP protocol which can be found in the INET documentation or examining the ARP.ned file itself.

PART 5, Model Discussion and TODOs

Complex Module
The Complex module controls the background traffic in the system. It works by reading in a time series, stored in a separate file. The time series should be a list of numbers between 0 and 1, split up into rows and columns. Each column is its own time series. The module reads the entire time series file into a very large array for quicker access at run time.

At each time step, default is every second, the complex module will grab a list of every queue in the network, and based on the time series tell that queue how many fake packets minimum must be in the queue. At this point this is all the complex module does, so its just the beginning of a full blown traffic controlling module.

It is recommended that any special control over routers should be added to the complex module, allowing it to control failures and re-instantiation of nodes, as well as the redistribution of fake traffic during a failure. These specific simulations were out of scope for this project, as this project only needed to build the initial network modeling tool and examine a few possible causes of nonlinear behavior. The module has been tested, and the "type" parameter in the queues was used to test its control over the system.

BGP and DYNAMIC ROUTING
The BGP protocol is not working properly. The simulation will crash after about 123 seconds consistently, due to an error at the TCP level. Since both RIP and OSPF worked, the root cause of BGP's tendency to crash has not yet been fully analyzed. The problem very well could exist in the TCP definition, since TCP traffic has not been fully utilized as well. The rest of the dynamic routing protocols can't seem to handle more than around 60 interfaces without issue. Although this means network of upwards of 15 to 20 routers can be handled, this number will ultimately be insufficient for running many of the simulations desired by the model. This issue needs to be addressed as well.

ETH and PPP
It is recommended to continue the use of the PPP or point to point interface. This interface allows a bit more control over the simulation, in that it can allow for different times between hosts, simulating various bandwidth in the connections. Also the automated scripts to take one from ReaSE to a runnable simulation with dynamic routing is only compatible with the PPP interface.