Examining Network Routing Algorithm Efficiency

Tyler S. Worman
Dr. Ben Coleman - Project Adviser
Dr. Stephen Dunham - Honor's Liaison
Moravian College
Honors Fall 2006

Abstract

Computer networks are becoming increasingly more complex. Efficient data routing on complex networks is critical to ensure efficient use of a network. This paper details research on 3 different routing algorithms studied on networks with a ring-based topology at the center. A computer simulation was written to generate statistics on each othe algorithms. Using these statistics we show that different algorithms on this specific network layout are most efficient under varying traffic levels. A network layout with a central ring presents a problem for any routing algorithm to route around effectively.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The goal of a computer network is to allow information to travel from one computer to another. It is similar to a system of roads where vehicles move between destinations. Drivers have various paths they can take, and the travel time is dictated both by the layout of the roads and the quantity of traffic. For example, in Figure 1.1, a car has 3 paths it may take. The upper path is obstructed by a car crash that the driver is likely to see. The middle middle path also has an accident, but the driver is unlikely to be able to see this far ahead. The third path is unobstructed. In this situation, it is actually best for the driver to select the longer third path.

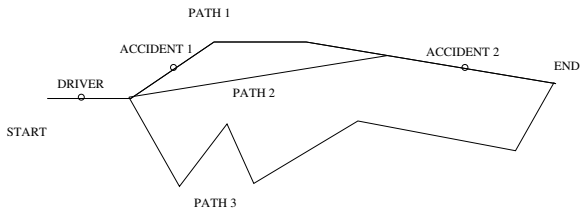


Figure 1.1: Driver in traffic with decisions to be made.

Unlike this real-world example, where decisions about routes are decided by the drivers, we are interested in the situation where decisions are made at a global perspective. Regardless of the decision-making model, the process used to decide a path is called routing[7].

Decisions upon how data should be routed can be made by examining available information. These choices are split into two categories, global and local decisions. To make a global decision, information for the overall status of a network is used. Figure 1.1 shows a global decision when the driver is able to take into account that accident 2 exists when making his

decision to take path 1, 2 or 3. When making a local decision for a routing path, only data that is immediate to a router is used. Figure 1.1 presents a driver with a local decision if his only knowledge is of accident 1. The driver's choice is made solely upon what is in view at present time.

A router uses a routing algorithm, which is a way to systematically determine a path between two points, to decide how data will travel[12]. The goal of routing within a network is ultimately for data to flow in the most efficient way between a start and end destination. Effective routing can produce not only satisfied end users, but it can also reduce the number of routers needed, and reduce costs for businesses installing networks.

As data flows across the network, it is sent in bursts of data called packets. Packets have varying sizes based on the amount of data that is carried within them. If connections on a path were to become unavailable, then a packet reaching a router may be redirected along another path that is complete. By utilizing information on the status of connections, congestion on these connections and also current network utilization, routers have a chance to optimize paths for new data flowing through them.

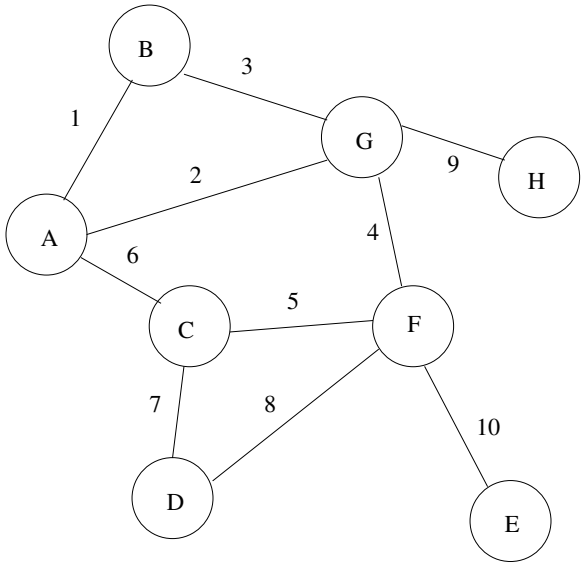


Figure 1.2: A simple graph

When studying how to route data packets efficiently on a network we are actually using tools from an area of mathematics called graph theory. A network is a special type of representation called a graph. A graph consists of nodes, and connections between them called edges[11]. Figure 1.2 shows

a sample graph. Nodes are designated by circles and the edges by lines. On a computer network, nodes represents computers and routers. Edges describe the connections between nodes and their associated transfer speeds. For example, if information at the computer labeled "E" needed to go to computer "H", it could go through computers "F" and "G" to reach "H". The shape of a network given by a graph is called a topology.

Topologies' unique shapes may have effects on how data will be routed across them. Using routing algorithms we can choose a path for the data to take. When choosing a path it is possible to make a decision to use a short path or a different longer, yet quicker path. As network conditions change, paths for data to take can be optimized so that data continues to flow smoothly.

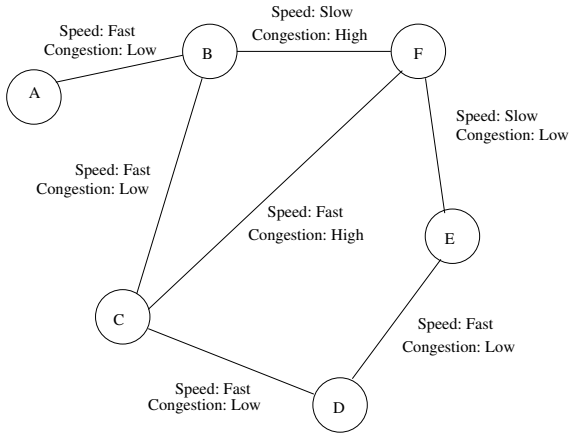
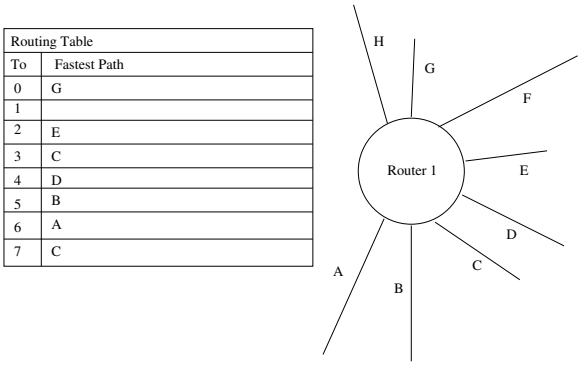


Figure 1.3: A Small Network

In Figure 1.3, a piece of data on the network is trying to get from router A to router F. When the data reaches router B, the path for the packet must be determined. With knowledge of the size of the packet an efficient route based upon the speed of the line can be calculated. The chosen path may direct the data across the shortest route on the network, which would be by traveling from A to B to F. An alternative option is for the router to have optimized the route based upon the path with the fastest interconnections, this time going from A to B to C to F. This route lets the data flow at a higher speed across the network, ultimately resulting in a quicker arrival rate. A third alternative is presented in Figure 1.3. By deciding upon a route using line speeds and congestion on the network, a better optimized route can be produced. Speed of lines is fixed on a network, however congestion is

dynamic, therefore routes can vary at all times. This optimized route might direct a packet from A to B, B to C, C to D, D to E, and finally E to F.

To implement algorithms, routers calculate a path and store it in a table so that when a packet is to be sent its route can quickly be looked up and the most efficient route can be chosen[6]. Figure 1.4 shows an example of a router and a possible routing table. The router's table contains information on how a packet of data will navigate through the network. To get from router 1 to router 5 the best way to travel would be via route B.



Routing Table	
To	Fastest Path
0	G
1	
2	E
3	C
4	D
5	B
6	A
7	C

Figure 1.4: A Sample Routing Table

Choosing an effective routing algorithm to generate paths for storage in routing tables, will result in an increased rate of flow on specific network structures[13].

PPL Corporation[2], a power company native to Allentown, PA uses a network layout which is quite different from the average local network topology. Most local networks are pyramid shaped with 1 node connecting the network out to a larger global network called the Internet. They are designed this way as it provides an easy and effective way of securing a network from exposure to the Internet. By securing one single computer you can now secure all computers from external connections. At PPL, the network layout must join several offices and power plants together, to ensure that each office has a fast and reliable connection to the others. Each larger site has a pyramid shape below it, which is connected to the other larger sites by first joining to the ring.

In Figure 1.5 the network resembles a ring in the center. From the nodes on the ring, structures which maintain a general pyramid shaped structure are connected. These pyramid-like topologies are present at an individual site. Essentially, each pyramid portion is a local network with a ring node at the top of it.

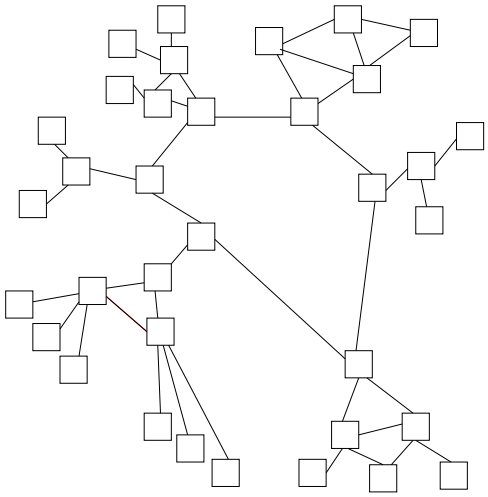


Figure 1.5: Network Ring with sub tree's

Many routing algorithms exist. and in our research we examined how very simple algorithms effect data flow on a very specific and unique network layout such as PPL's. To perform this research, we wrote and verified a computer simulation. Using this simulation software, we compared algorithms on a network with a topology like PPL's while varying traffic patterns. We used simple algorithms that optimized based on shortest path, on speed of connections, and also one that optimized for connection speed based upon congestion.

Our original thought was that the congestion optimization algorithm would be the best of the three, despite the network topology. After performing the simulation, we found that with small packet sizes. optimizing based upon shortest route was sufficient. When using varying packet sizes, the fastest path algorithm was the most efficient. The congestion optimization algorithm used was actually the least efficient under heavy traffic patterns. This is because routes generated by it were very long and the path wasn't dynamic as the congestion on the network was. Because of our specific topology, none of the algorithms can truly excel because of the ring. We concluded that the center ring in our topology using any routing algorithm would act as a bottleneck. The center ring cannot be easily routed around if it becomes congested. To reduce the chance of congestion on the ring the ring speeds must be sufficiently fast to handle all possible traffic loads put on it at ring nodes.

Chapter 2

Related Works

No single algorithm is the most effective on all network topologies, since the layout effects bottlenecks on networks. Due to this, research on new topologies is constantly being published. Algorithms are tested for increases in data transfer speeds on specific topologies. Companies like PPL, stand to gain by implementing a routing algorithm which is most effective on their specific network topology.

The efficiency of a computer network due to the use of a specific routing algorithm may be measured by monitoring statistics gathered on both routers and from PC's connected to the network. On computer networks, efficiency is measured by the time to transmit an individual packet of data. To determine an efficient network, a study should be done that looks at the packet flow per second onto the network, the amount of packets waiting to be processed by routers, the amount of routers a packet must travel through (commonly referred to as hops), and the speed, packet sizes and usage rates of the routes taken[6].

There have been many approaches to research in area of networking and routing on certain topologies. Our research is similar to much of the research found on the study of algorithms on network topologies. Our research surrounds the discovery of how specific routing algorithms work on the unique topology such as PPL Corporations network. In our simulation, we studied 2 algorithms which optimized solely on layout of the network and one algorithm which chose paths by taking network congestion into consideration. These two types of algorithms are commonly studied, but research of their use on our specific central ring based network topology has not been published.

Ishai Aroya, Ilan Newman and Assaf Schuster studied hot-potato routing algorithm on a hyper-cube and high-dimensional meshes[3]. In hot

potato routing, queues at each router are not implemented. A packet is constantly passed around the network until it reaches its destination so that clogs do not occur at the routers.

Work with specific algorithms was also performed by Michael Mitzenmacher. He researched Greedy Algorithms on array based networks[10]. Greedy Routing algorithms choose the shortest route between the start and end node on which a the path which packet will travel. A greedy algorithm is one of the simplest algorithms to implement. Our research included the study of greedy routing algorithms on our specific topology.

Similar research was done by Xiaofan Yang, Graham M. Megson and David J. Evans[14]. They looked at the same type of shortest path routing algorithms as Mitzenmacher, but observed the algorithm on a network topology based upon fully connected cubic networks.

Yossi Azar, Edith Cohen, Amos Fiat, Haim Kaplan and Harald Racke studied adaptive routing algorithms on several network layouts[4]. They did not however look at the use of these algorithms on our specific network topology.

Further work was done with adaptive algorithms by B. Zhang and H.T Mouftah. They studied a shortest path routing algorithm that also takes into account bandwidth constraints[16]. This routing algorithm used in their research is comparable to the first type of adaptive algorithm that we will study on our network topology. However, Zhang and Mouftah designed their algorithm with the assumptions that bandwidth restriction is in place lines due to the use of a protocol that attempts to ensure quality of service.

Adaptive algorithms were also examined by K. Chi, C. Yang and X. Wang. They wrote about using shortest path and maximum rate distribution and its effects on common Internet topologies[5]. The two adaptive routing algorithms which we will be studying are modeled closely around the same ideas presented by Chi, Yang and Wang. The topology which they performed their research on was common Internet architecture. Chi, Yang and Wang's research was used a specific type of data transmission called multicasting. In multicast transmissions, a single message is sent to the network and the network delivers a copy to each of the recipients of the multicast.

Research on networks that share similarities in layout to our specific topology were also found. Chun-yen Chou, D.J Guan, and Kuei-lin Wang looked at routing on double-loop networks[15]. Their research examines routing on a redundant ring structure. The research done by Chou, Guan and Wang looks to find an efficient way to balance data transfer about the double ring loop.

The pyramid shape structures that are connected to our central rings nodes have also been studied by M. R. HoseinyFarahabady and H. Sarbazi-Azad[8]. They examined a generalized pyramid-shaped network topology. The pyramid like structures are formed by connecting a single head node to an increasing number of devices as the network grows larger. M. R. HoseinyFarahabady and H. Sarbazi-Azad's research used algorithms that were optimized to take advantage of the pyramid shape of the network.

Overall, our research provides an in depth look at basic routing on a complex network topology. We present that on our specific topology under certain conditions it is more efficient to use a specific type of algorithm.

Chapter 3

Simulating a Network

3.1 Introduction

To run tests and gather statistics on this specific type of topology, experiments could be run on an actual network while varying different properties on the network. Since PPL would not permit experimentation on their own network and creating an actual network large enough to simulate this is cost prohibitive, a computer simulation was developed. The computer simulation enables us to model how a network functions. Using the simulation, we can run series of tests on our specific network topology and gather data on each of the tests for later comparison.

3.2 Model Description

The simulation was designed with the intention of modeling a specific network topology. The simulation software can be altered to allow further experimentation on networks of the unique shape studied. The topology that we used in our simulation was designed to resemble the unique layout of a network like the PPL Corporation maintains.

PPL's network is built off a central ring that has a very high data transfer rate. In recent years, technologies that enable fast interconnection speeds have been developed[6]. Since not all systems attached to a network need such a high speed connection, the larger sites on the PPL network only use these high speed links to connect to other larger sites. The large sites link together to form a ring at the heart of PPL's layout. Data sent from one large location to another on the PPL network travels to the "top" of its network and then onto the ring. Once a data packet arrives on the ring, it is

transported at a high rate of speed to the node on the ring on which its destination is located beneath.

A generalized version of PPL's network topology is represented in Figure 3.1. The actual topology used in our simulation however was much larger and was more complex then the simple example shown in Figure 3.1. Our topology used in simulation has more nodes on the ring, with larger and deeper networks connected to the ring. The chance of an interconnection occurring between two routers within a network was 50%. The Line speeds were evenly split between 10 and 100 Mb/s connections.

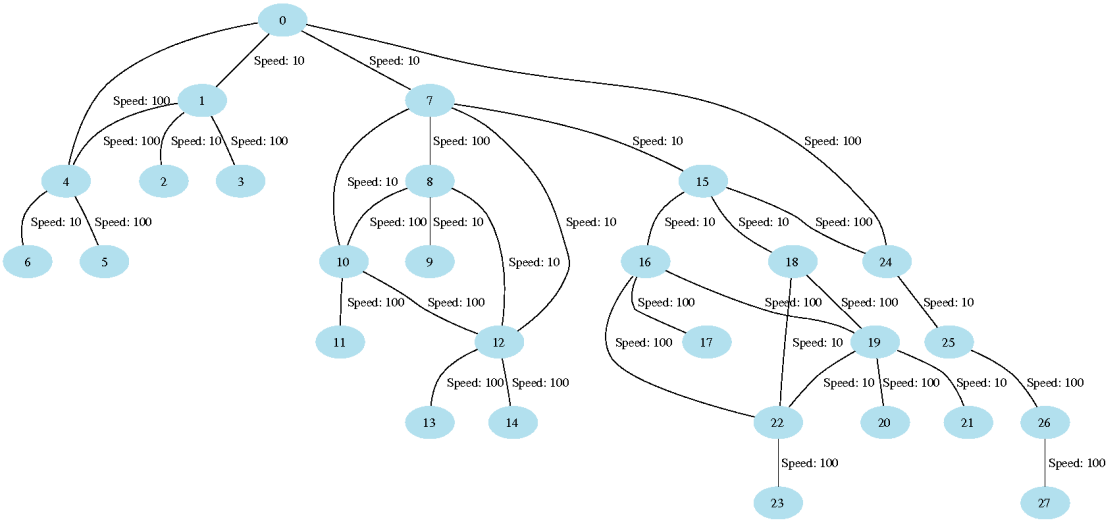


Figure 3.1: The Generalized Network Topology studied

By examining this specific topology, we determined which algorithm from a selected set was the most effective on networks with a layout of this type. The ring being located at the center of the network topology is what makes this a unique and interesting problem to research.

In Figure 3.1 computers are shown by having only one connection to a single router. This assumption in our network layout excludes computers with more than one network connection. Routers are all nodes in between that connect to 2 or more nodes.

We make the assumption that connections will not fail and data can always reach from any given node to any other node to which it is connected. In reality, connections can fail and nodes can be unroutable. In our research, meaningful data was gained without modeling these traits.

To model a network as realistically as possible, we considered what traffic on the network would look like. By using a log of traffic data from an actual network[1]. we were able to observe a general pattern that occurs on a network. The packet log shows the packets sizes that crossed the network. The maximum packet size was 1500 bytes. The log shows a large amount of very small packets, a small peak at around one third of the maximum packet size, and finally a large peak at the maximum packet size. Then we decided that for each algorithm tested we would run the simulation with two different forms of traffic. The first uses a varied packet size based directly upon the distribution shown in the log. The second is an average of the size of all packets found in the log. The distribution of packet sizes uses a function which closely models the actual data shown in the log. Figure 3.2 shows the sizes of packets found in the data set used to determine the values modeled.

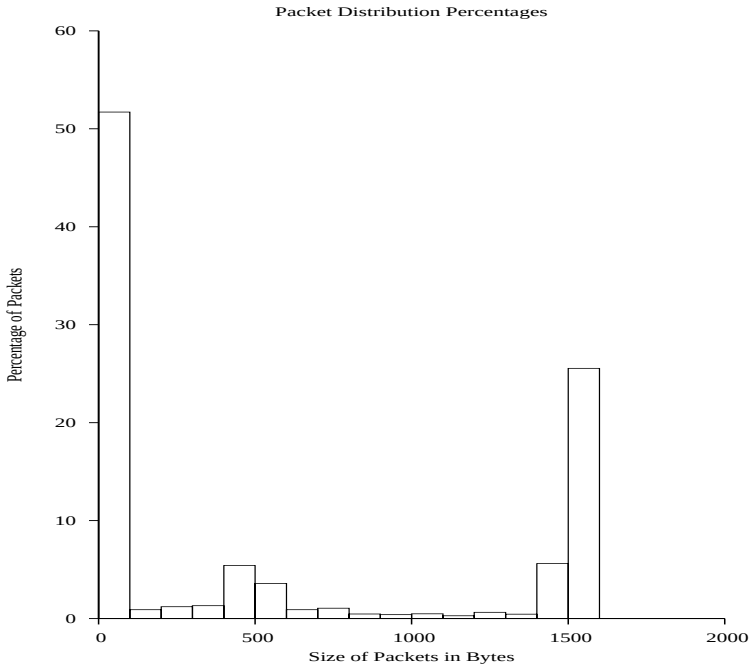


Figure 3.2: Logged network packet distribution

In the simulation, traffic moved from one computer to another. The source and destination we determined by selecting a computer from the set of all computers. Each computer had an equal chance of being selected.

The route for the data to travel is generated upon a specific algorithm that is being run that simulation. Since the study is of the unique network

topology and how routing algorithms effect it, we chose to use 3 simple routing algorithms. Two of our algorithms choose paths, which are independent of traffic on the network for all packets that go from point A to point B during a simulation. The third algorithm dynamically chooses paths along the network based upon traffic load for each of the packets sent.

The first algorithm dictated that data should always take a path in which it travels between the minimal number of routers to reach its desired destination. The second algorithm dictated data travels over the fastest connection available. The third algorithm finds a route with the least number of hops from start to destination over the least congested of the connections.

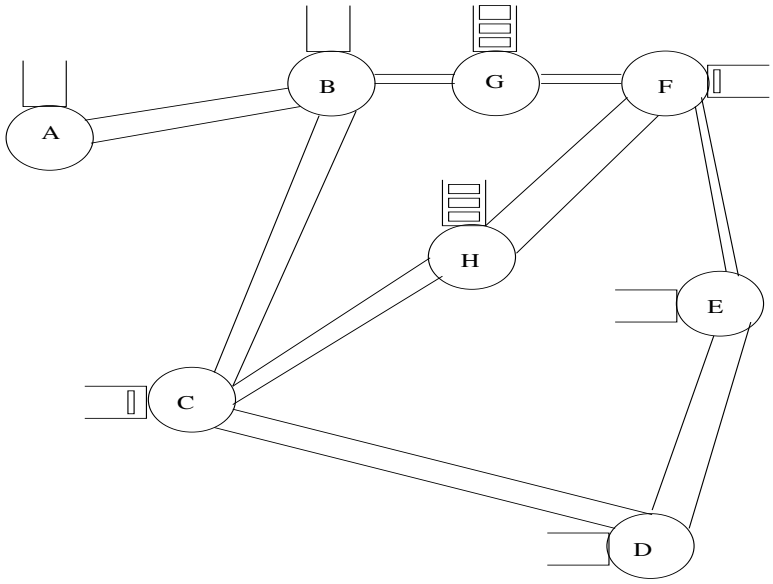


Figure 3.3: A sample network with various packet wait times.

Shortest Distance: In this routing algorithm, the path is selected based on the path with the shortest amount of hops (connection traverses from one router to another). It does not take into account the speed or wait at each router along the way. For example, in Figure 3.3 the shortest path algorithm will choose a route with 3 hops between router A and F. The path chosen would be from A to B, B to G and G to F.

Fastest Connections: In this routing algorithm, the path is determined by selecting connections based upon speed and then selecting the path that has the fewest slow connections. It does not take into account congestion on a connection or look to find the least number of hops from one destination to another. In Figure 3.3, faster connections are designated by larger lines

that connect them. A route from A to F by choosing the fastest route for the network (shown in Figure 3.3) will choose to route over the fastest connections with the shortest amount of hops. The path this algorithm will choose is from A to B, B to C and C to H, H to F.

Congestion Sensitive: In this routing algorithm, the path is determined by choosing a connection based upon speed of a connection and a comparison of its health to the health of other paths. "Healthiest" is defined by the fastest connections with the least backlog that might impede data transfer. Impeding data transfer was weighted by the number of packets waiting to be processed by the next router multiplied by that specific router's average rate at which it processes packets. Figure 3.3 shows a route from A to F that would be chosen when using this algorithm. Routers G and H are busy due to having 3 packets in queue and also because these packets are large. Since G and H have high congestion, the algorithm will attempt to avoid these nodes if another path with less "wait time" is found. An optimized route using this algorithm is traveling from A to B, B to C, C to D, D to E and finally E to F, successfully avoiding the congestion.

When a data packet reaches its destination, it is assumed received and removed from the network. We assume that all data is intact and that all transmissions are perfect. Because of this, we do not need to verify the data on arrival and retransmissions are not modeled. Our modeled routing protocol is a generalization of most networking protocols for this reason. This allowed us to abstract the details of a specific protocol out of our simulation.

To run the simulation, we generated a single-network topology that was representative of a network like PPL Corporation's and ran the simulation with the following parameters:

The length of the simulation was 30 seconds. Packets arrived on the network with a rate of 2857 packets per second. The rate of the packet arrival was chosen by looking for a distribution in which the center routers began to experience congestion. Packet size was varied between static and distributed sizes. A route's beginning and destination were chosen by a process in which all computers on the network have an equal chance of being selected.

3.3 Simulation Design

The simulation we designed is modeled after a discrete-event simulation[9]. In a discrete-event simulation, each change in the simulation occurs at a specific and unique time. Simulating systems in this manner makes it easy for

a computer to calculate effects of interactions since all items occur in a specific order. In our simulation, a packets arrival and departure times are each separate events. As packets are added to the network, the amount of events for the simulation to maintain grows.

A model of the network for the simulation was created. Using C++, we created a network topology data-file generator. Code for this piece of software is available in the first section of the appendix. The software takes basic parameters on the size of the network it should create and then builds a network around a central ring of user specified size. Upon completion, the network and its interconnections are written to a data file. The user has the ability to alter the amount of routers in the circle, the lower and upper limits on the width, the depth on each of the tree's that connect to the central ring, the probability of interconnection within trees, and the probability of connection speed's being designated slower then others.

Upon completion of the ring software, we had to create and choose a network size that both resembled PPL's topology and was small enough to run the simulation on in a feasible amount of time.

The simulation, also written in C++, was designed to keep track of all packets on the network and the order that events would occur to deliver a packet to its destination. The code for the simulation is shown in the second and third section of the appendix. It first loads the model of the network and then creates a queue in memory for every router on the network. In our simulation, when a packet is in the process of being transferred other packets queue up behind it and wait until the router is free to service the other packets. The simulation also manages an event queue. The event queue stores a list of all packets that are at the front of a routers queue on the network. These packets in the event queue are currently being used by the simulation and are being moved from router to router.

The simulation starts by generating a packet. Once its starting and ending computers are chosen, we generate a route for the packet to travel. We use Dijkstra's algorithm, which is used for finding paths on graphs, to generate the path the packet will take. Each routing algorithm is implemented based upon Dijkstra's algorithm.

To achieve variation in routes, each algorithm, gives different values to each path on the network. In the shortest distance algorithm each path is assigned a value of 1. To find the shortest path, Dijkstra's algorithm finds the path which the sum of values for each connection is the least. When generating a path looking for connection speed, the inverse of the connection speed is assigned to each part of the network. The path's speed values must be inverted, so that Dijkstra's algorithm can find the fastest path. The

slowest path would have the lowest speed values. The implementation of the congestion sensitive algorithm assigns values to each path that are summations of the connections speeds and the amount of wait time that would be encountered at each router in its path.

The packet is then inserted onto the network. The arrival time of the packet at the next router is generated based upon the size of the packet and the line speed it must cross at. This event is scheduled. We then choose a time for the next packet arrival and schedule that event. The event-list is sorted based upon time so that events will occur in the proper order. With 2 events now in the event queue, we choose the event which will occur first then process it. Packets progress through the simulation by moving from one queue to the next along its pre-calculated path. When a packet reaches its destination computer, it is removed from the event queue and statistics about that packet are gathered. This is repeated thousands of times for a large number of packets so that congestion on the network is created and the router queue's will have multiple items waiting to be processed.

As network simulations sizes grow, it is important to manage the data effectively by maintaining separate queues for events currently processing and events that are queued. The effective management of this data was crucial to allow the simulation to run in a reasonable amount of time.

Throughout the development of the simulation, we constantly verified that any changes made continued to ensure accuracy. When developing a large piece of software it is crucial, since not every single features will be tested during an actual simulation. A large simulation is also extremely hard to verify by hand. Because of this, we created small basic tests so we can ensure that each small portion of the simulation works as it should. When these smaller pieces inter-operate the simulation should then work fully as it should. Throughout the development process these tests proved crucial to detecting and correcting errors. The suite of tests we created derived from an original base case that was suggested in Leemis' book on discrete event simulation[9].

Figure 3.4 is a basic test that moves a single packet from one computer to another. The completion time for this packet was the size of the packet divided by the speed of the line.

Figure 3.4: A simple single transfer of a packet between 2 computers

Building upon this test, we next ensure that the sending and receiving of data by two different computers is actually a disjoint operation and that they do not interfere with one another. Figure 3.5 shows the original 2 computer network with traffic now moving in two directions.

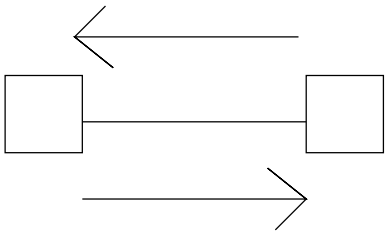


Figure 3.5: A simple 2 computer bidirectional packet transfer

The next step was to create a larger network that features a router in between the two computers. Figure 3.6 shows a central router which will be responsible for relaying packets between the two computers. As packets arrive closely in time, the packets will be queued at the central router. Once the central router completes sending a packet it will continue until its queue in clear.

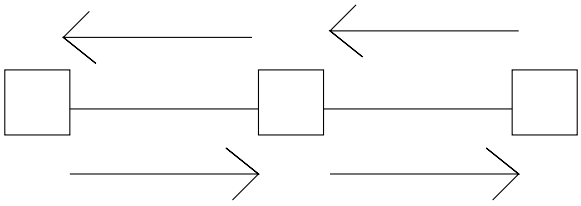


Figure 3.6: Packets are sent across a network in both directions at the same time

The test performed in the network in Figure 3.6 is the basis for all complex network layouts. We then increased the example shown in Figure 3.7

to have several additional computers with one central router. By adding additional central routers, this very basic network would become a basic ring topology network like the one which we studied.

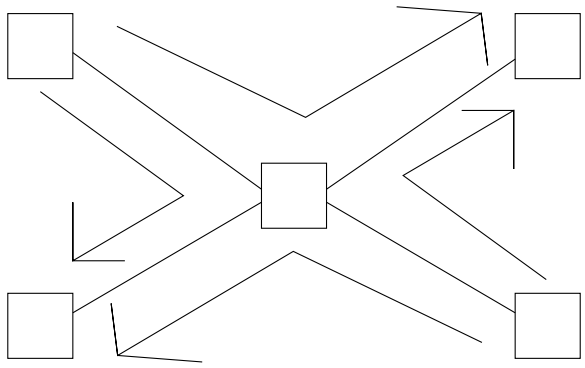


Figure 3.7: Packets will be queued at a single central router before reaching the destination

Figure 3.8 demonstrated queuing at multiple central routers. Packets have two places in which they can become congested based upon the flow of the data. This final test assures that the simulation is moving a packet across a multiple-hop path based upon a path chosen by the routing algorithm.

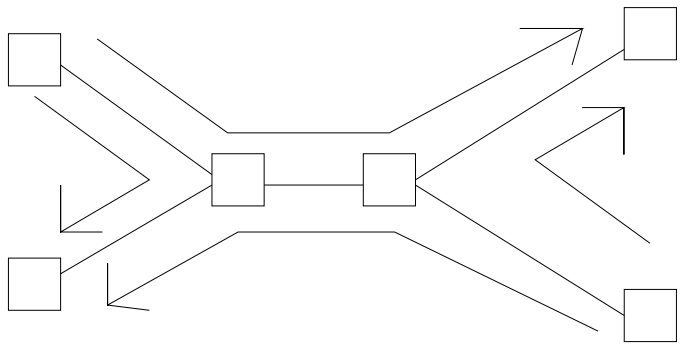


Figure 3.8: Packets will travel and be queued at multiple central routers before reaching their destination

The success of these tests insures that the basic way in which packets transfer from one router to another is correct. With the knowledge that basic transfers are correct, we state that the simulation is correct for larger more complex networks. Larger networks are derived from a combination of smaller networks.

These tests were designed to be simplistic so that the results could be checked and verified by a human. The final results from a complete running complex network simulation would not be easily verified by a human.

Creation and verification of a software simulation was crucial to accomplish the research. A large portion of the time to conduct this experiment was spent on developing the custom simulation software. The software development and verification life cycle for complex software is an important part of the process to ensure that the resulting software works as intended.

The simulation software underwent extensive debugging and modification session so that the final simulation could be run correctly in an optimal amount of time, with limited computing resources.

3.4 Results

A total of six experiments were conducted. The first set of three were conducted using varying algorithms and a static data-packet size which was chosen by averaging the size of all packets in the distribution presented in Figure 3.2. The second set of three used the same algorithms and packet sizes that conformed to the distribution seen in Figure 3.2.

First, we chose to use a large static packet size, since the abnormally large packet size would cause the network to become heavily congested. On a congested network, each router has a long wait queue till it can deliver a newly arrived packet. The congestion optimization algorithm should optimize around this if possible. The second varying packet size distribution closely mimics the type of traffic that would be seen on a real network. By carefully handpicking these two packet size distributions we are able to see the effects of the algorithms under both a normal and stressed network condition.

It is important to note that a network with a low level of traffic (such that no packet will get queued at router), both the line speed and congestion sensitive algorithm are equally effective. The congestion algorithm will only make optimizations over the line speed algorithm when when traffic on the network could hinder a packets delivery time.

If a network was to have a homogeneous connection speed, the best-case scenario when not considering traffic load is the shortest path. Because of the design of the algorithms, when the shortest path and line speed algorithms are used on a homogeneous network they will also be equally as effective. If the traffic level is low (such that no packet is being queued), then all 3 algorithms are equal.

Figures 3.9, 3.10, and 3.11 show the number of hops that each packet had to take under the shortest path, quickest path, and congestion sensitive quickest path algorithms

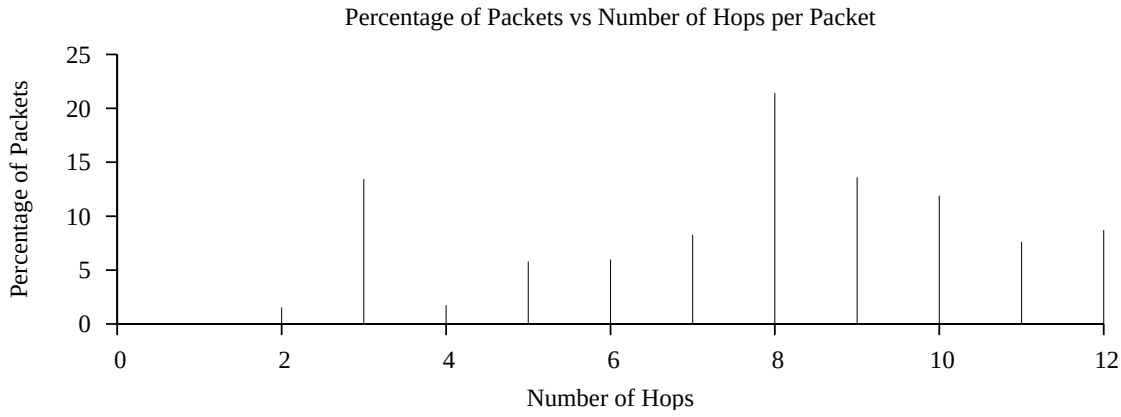


Figure 3.9: Packet hop percentages using shortest distance algorithm and fixed packet size

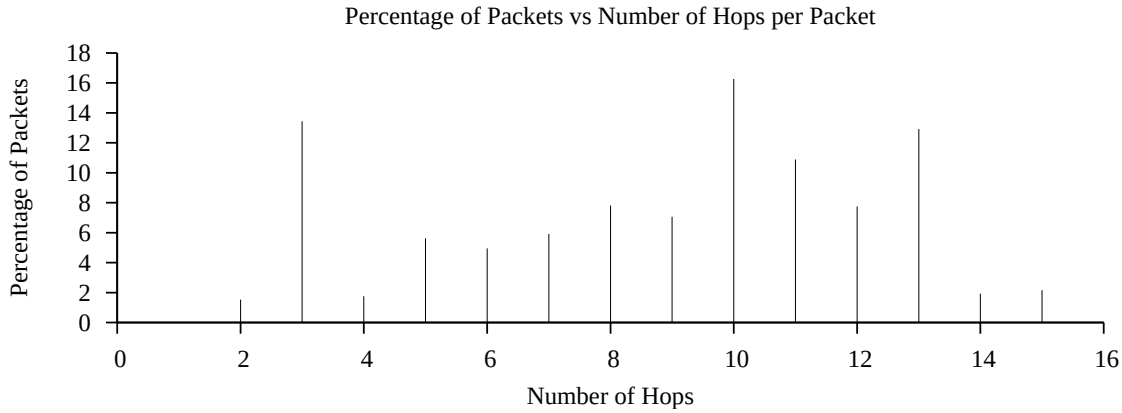


Figure 3.10: Packet hop percentages using line speed optimized algorithm and fixed packet size

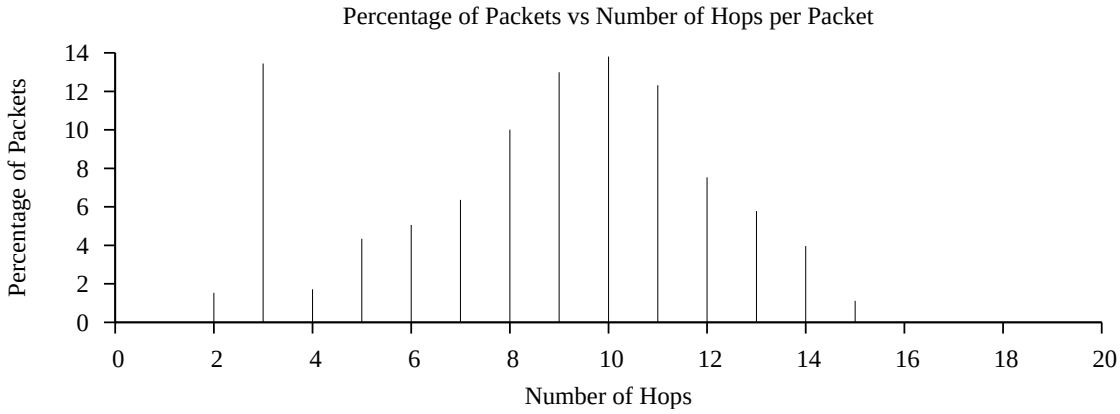


Figure 3.11: Packet hop percentages using congestion-optimization algorithm and fixed packet size

We can see a variation in the number of hops between all three algorithms. The shortest hop and line speed algorithms have a very similar shape while speed optimization is only heavy in its upper range. The second peak that occurs is pushed farther out in the speed optimization and congestion optimization algorithms indicating a longer path taken by each packet. The first peak, which occurs at three hops for each of the algorithms, is representative of packets traveling to neighbors on the same local network.

Figures 3.12, 3.13, and 3.14 show the time it took for a packets to complete its travel across the network using the shortest path, quickest path, and congestion sensitive quickest path algorithms

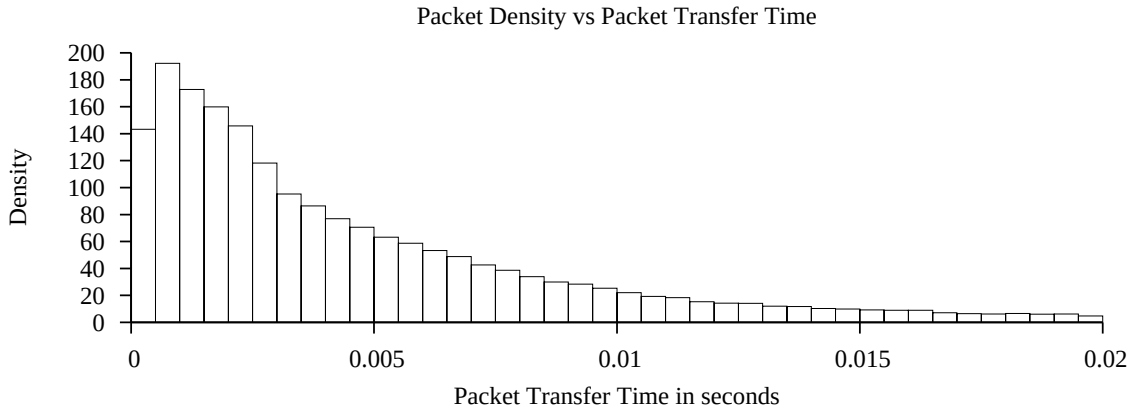


Figure 3.12: Time for packet delivery using shortest distance algorithm and fixed packet size

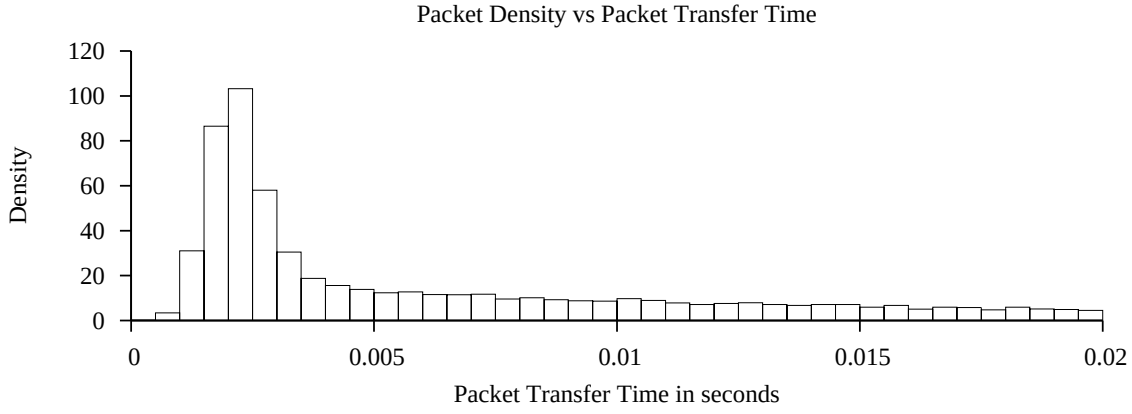


Figure 3.13: Time for packet delivery using line speed optimized algorithm and fixed packet size

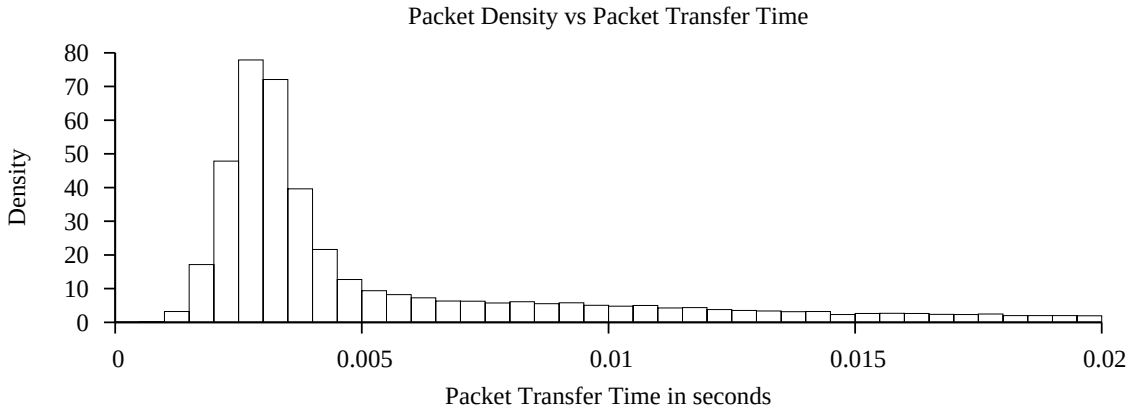


Figure 3.14: Time for packet delivery using congestion optimization algorithm and fixed packet size

Figure 3.12 has a large number of packets with completion times between 0 and .0025 seconds. The amount of packets arriving at times in excess of .0025 seconds steadily declines past that point. Under the shortest distance algorithm, the largest number of static packet sizes at a completion time was moved out farther compared to the other two algorithms. The speed and congestion optimization algorithms offer an evenly distributed completion time for packets. These figures show that the shortest distance algorithm is the most efficient when using a fixed, larger packet size.

Tests were also run with the same three algorithms on a network in which the packet size was based upon a distribution of the observed packet sizes shown in Figure 3.2.

Figures 3.15, 3.16, and 3.17 show the amount of hops each packet took to reach its destination; using the shortest path, quickest path, and congestion-sensitive quickest path algorithms.

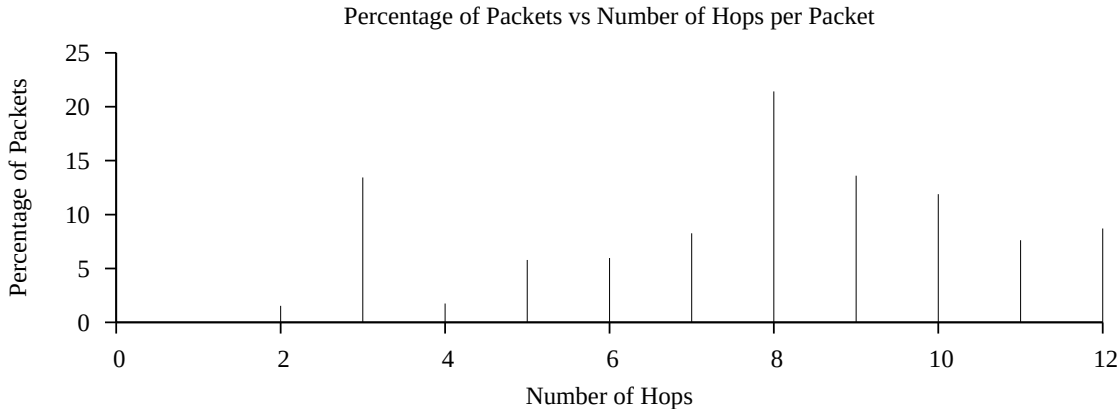


Figure 3.15: Packet hop percentages using shortest distance algorithm and varied packet size

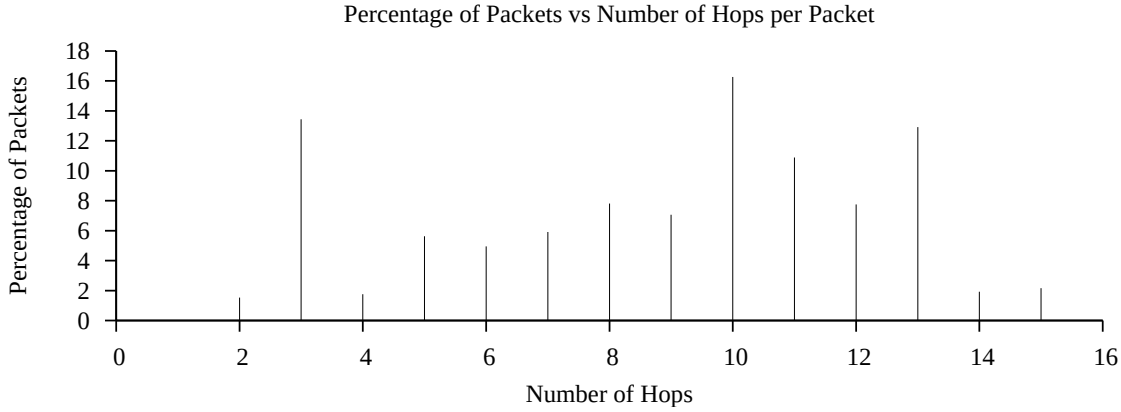


Figure 3.16: Packet hop percentages using line speed optimized and varied packet size

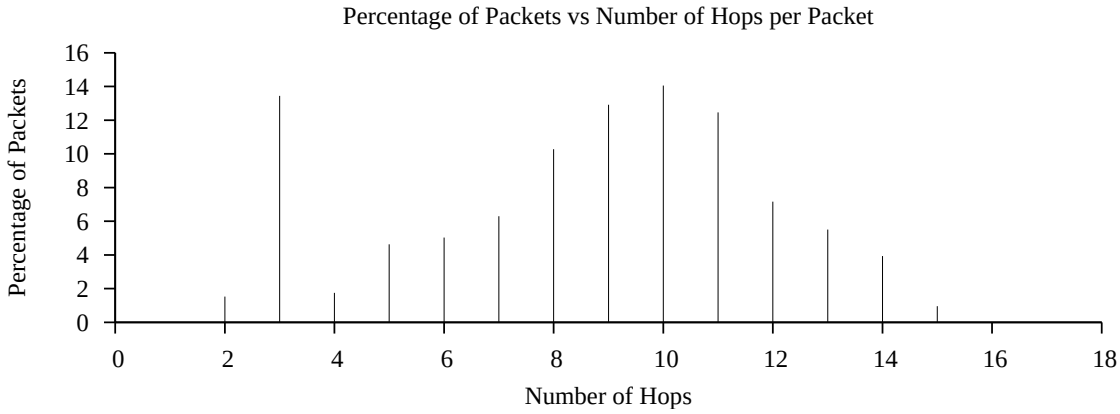


Figure 3.17: Packet hop percentages using congestion optimization algorithm and varied packet size

A surge of routes utilizing a large number of hops is shown in Figure 3.17. Since the algorithm optimizes its route based upon congestion on the network as it reaches a congested area, the path is routed around it utilizing a higher number of hops.

A large spike in the number of hops for a packets completion is shown around ten hops in all hop charts shown. Next to that, all charts of hops show a low area around two and four hops. These same general traits also occurred when using the fixed packet sizes. The routes generated in each simulation are the same between both fixed and variable packet sizes under the shortest distance and line speed algorithm. Using the congestion algorithm the routes will differ under a heavy load. By comparing the amount of hops from both the fixed and variable packet size experiments, we can see that using the congestion optimization algorithm yielded on average the same amount of hops.

Figures 3.18, 3.19, and 3.20 show the time it took for a packets to complete its travel across the network using the shortest path, quickest path, and congestion sensitive quickest path algorithms using a distributed packet size.

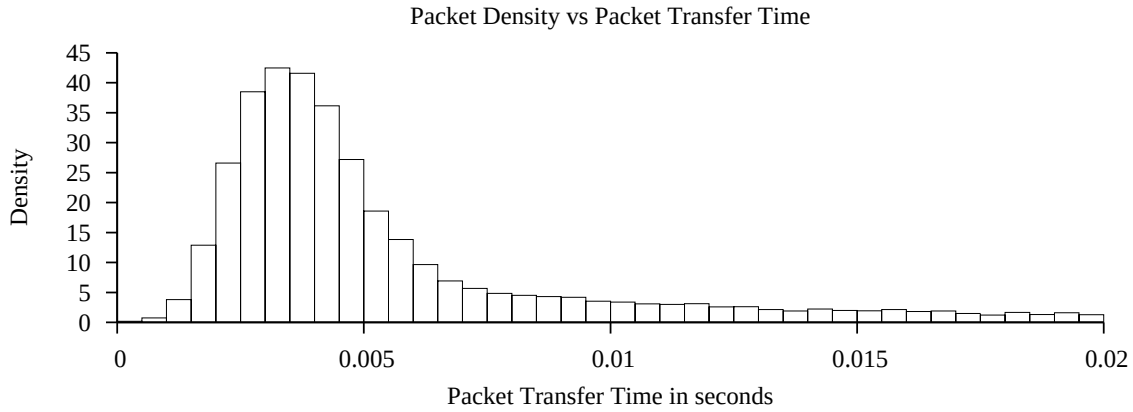


Figure 3.18: Time for packet delivery using shortest distance algorithm and varied packet size

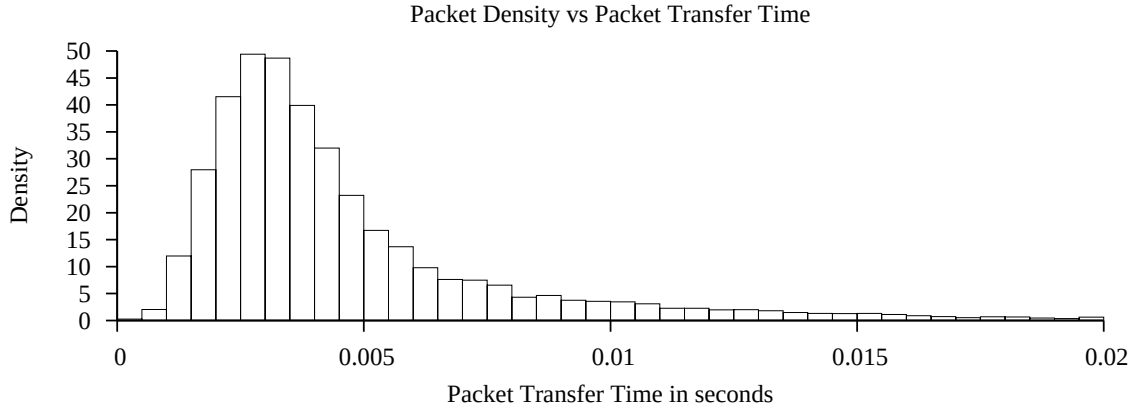


Figure 3.19: Time for packet delivery using line speed optimized and varied packet size

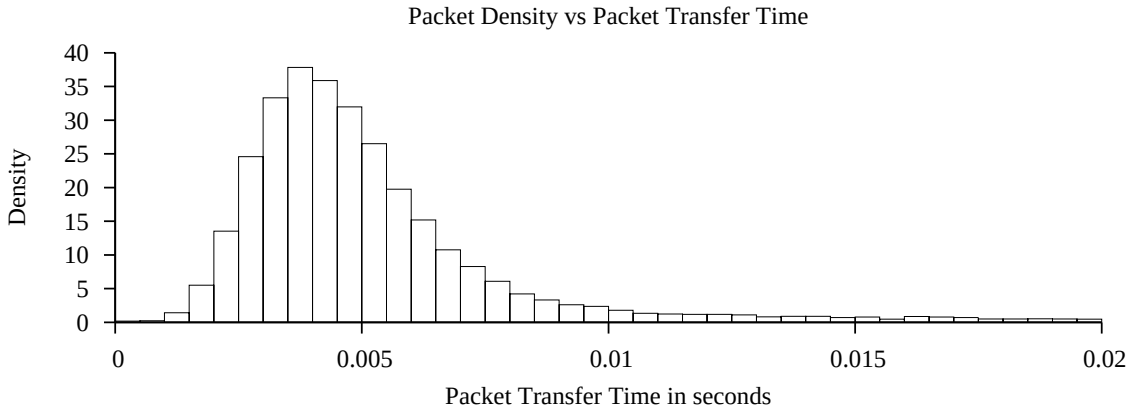


Figure 3.20: Time for packet delivery using congestion optimization algorithm and varied packet size

Figure 3.18 shows a peak just before .005 seconds of transfer time and shortly after the density of packets with higher traffic times trails off. This is different from the same experiment run with a fixed packet size. Under the fixed packet size this algorithm seemed to clearly excel. With the variable packet size this algorithm is now more on par with the line speed and congestion optimization algorithms.

The line speed sensitive algorithm has the majority of its packets with completion times of less then .005 seconds when using the distribution of packet sizes.

An interesting point to notice is that in Figure 3.20 the density of the curve's tail as the packet transfer time exceeds .001 it begins fades quickly. In comparison, this indicates that the algorithm has less packets completing with long time.

3.5 Discussion

From the simulations that were run, several conclusions can be drawn from the data. An interesting point to note is that in all cases simulations had a peak in transfer completion time occurring before .005 seconds. The shape of the curve that surrounds that peak was also similar. After .005 seconds the densities of packets completed at those specified times begins to fall. This is significant, because it shows that a large amount of the packets are being completed in a short time, and that the more dense the tail of the

curve the more packets are taking longer. A small tail with large peak early designates the most efficient algorithm.

By looking at Figures 3.18, 3.19, and 3.20 the general shape of completion times is similar. The line speed optimizing algorithm's completion times shown in Figure 3.19 start to become dense and at a faster rate then those from the shortest path algorithm. Completion times using the shortest distance algorithm are shown in Figure 3.18. The shortest path algorithm was the most efficient when using a fixed large packet size on the network. This algorithm causes the packets to leave the network in the fewest amount of hops reducing traffic and resulting in lower packet completion times.

When comparing completion time between the line-speed algorithm and the congestion-optimization algorithm completion times, the data shows that more packets finished sooner when routing based upon line speed then when routing based upon congestion of the lines. This is shown by Figure 3.19 and 3.20. Figures 3.16 and 3.17 show an explanation for why more packets finish sooner under the line speed algorithm than with the congestion algorithm. The congestion-optimization algorithm also shows an increased number of hops versus the line speed algorithm. This is due to the fact that under the congestion algorithm the routes are optimized based upon traffic at the time of a packet's creation. The route originally calculated on a network with high traffic congestion levels is not necessarily the fastest. After the first hop of the data on the pre-chosen route, the routers that it chose to navigate due to to congestion, may now be clear and the routers on the current path may be congested. Thus a large portion of packets using this algorithm take longer to reach their destination than with the line speed algorithm.

Due to the design of the tested algorithms, the line speed algorithm and the congestion optimization algorithm had a higher hop count then when using algorithm one on the same network. This is a result of the layout of the network, the amount of interconnections and the speeds of the connection on the network. A comparison of the hop counts is shown in Figures 3.15, 3.16, and 3.17. Since both these algorithms optimize their paths, they have a high probability of being larger than the same route chosen by shortest path algorithm. For every one 10Mb/s connection found in our network, the line speed and traffic optimization algorithms have a choice of using up to ten 100Mb/s connections to replace one slower connection, yet still remaining equally efficient.

When using the fixed large-packet size, the efficiency of the congestion algorithm was much worse then expected. Figure 3.14 shows a higher density at a higher time than that of the data from the least distance algorithm,

Notable Percentages of Hops using a variable packet size

Number Hops	Shortest Path	Quickest Path	Least Congested Path
3	13.44	13.44	13.44
8	21.42	7.81	10.28
9	13.60	7.06	12.92
10	11.89	16.27	14.06

Table 3.1: Notable Percentages of Hops Comparison between algorithms with variable packet size

shown in Figure 3.12. The reason is that the congestion-algorithm's path is optimized upon creation of a packet. However, after a newly scheduled packet makes a movement between two routers, the network is in an entirely different state of congestion. The previously optimized path may now be inefficient. When large packets enter the network at a high rate, the routers along the network quickly become backed up. With all routers backed up, the most efficient path at any given time is really inefficient. This leads to the longer packet completion times shown in Figure 3.14.

With a fixed, large-packet size, the shortest path algorithm completion times (shown in Figure 3.12), were lower compared to the other two algorithms. The benefit came from the ability of the shortest path algorithm to remove packets from the network in the least amount of hops. We demonstrated that the congestion algorithm is generally not effective compared to the line-speed algorithms. Both algorithms, however, create longer paths for a packet to take on our network. With a large packet size, the network becomes saturated easily. To reduce the congestion on the network, the most efficient way it to move large packets to their destinations without having the need to clog more router queue's any more then absolutely necessary.

The significant number of hops that packets took during each of the three variable packet size simulations is displayed in Table 3.1. In each simulation, because of the specific topology used, the layout of the network and the percentages of interconnections we saw two peaks. The first was when packets were traveling on their local network to a destination on the same network. The second spike came when a routing algorithm had to direct a packet across the ring. The large number of packets in these two specific amounts of hops is due to the equal distribution of destinations that packets must traverse to. These same results occur when looking at the amount of hops used in a fixed packet size simulation. Table 3.2 shows the significant percentages of hops that packets took in a simulation which used fixed

Notable Percentages of Hops using a fixed packet size

Number Hops	Shortest Path	Quickest Path	Least Congested Path
3	13.44	13.44	13.44
9	13.60	7.06	12.99
10	11.89	16.27	13.81

Table 3.2: Notable Percentage of Hops Comparison between three algorithms with fixed packet size

packet sizes.

When a network has a low percentage of interconnections between levels, a decrease in packets making their journey in 2-3 hops should be seen. It would also lead to an increase in packets taking four hops to traverse to other computers on the same local area network portion of the overall network. By studying the topology, we can show the least number of hops on the entire network and an average on what the least number of hops would be for packets to travel on a pyramid-shaped portion of the network.

3.6 Conclusions

The ring, while the most unique part of our network, causes the most problems in routing efficiency. The ring acts as a bottleneck for all packets that need to travel to other networks on the ring. If traffic builds up at the routers on the ring, it can not be effectively routed around. The ring only has two possible paths to travel around. Because of this feature in our specific topology, none of the algorithms can truly excel at routing.

Routing algorithms have the chance to optimize on each smaller network off the ring. Once a packet enters the ring portion of the network only two distinct paths around the ring exist. The solution to this problem is to ensure that the ring connection speeds are sufficiently high enough so that the local networks could not put packets on the ring at a fast enough rate to cause congestion. As more smaller networks are connected to the ring the ring speed must be raised. Redundant ring connections may also alleviate some of the problems caused by this specific topology feature.

At first glance, we thought the congestion optimization algorithm would be much more efficient then the other algorithms we tested. However, in the congestion sensitive algorithm, traffic on the network is dynamic and the route, after it is generated, is not. With a longer and static route, as compared to the other algorithms, the congestion sensitive algorithm actually

performs worse under our tested traffic intensities.

Routing algorithms are effective in optimizing the way in which data traverses a network. It must be stated that this study in no way concludes that a perfect or a "best" algorithm exists. It simply shows that on this specific topology, with these specific test conditions, the line speed based optimization algorithm and shortest distance algorithms are the most effective for variable and fixed packet sizes. The central ring constantly presents a unique challenge in effectively routing on networks with the studied topological shape.

Chapter 4

Future Work

While performing the research many questions arose that because of time restraints we were not able to fully explore. These topics would provide meaning full extension to the data results and allow us to better understand the effects of routing algorithms and this specific network topology.

Because of the time it took to implement and verify the simulation we were not able to gather all the data we would have liked. Using the simulation software as it was configured for the above experiments we would like to simulate the network under varying traffic intensity levels. This would allow us to gain a better understanding of the performance of each algorithm on this specific topology under an assortment of network conditions.

The center ring of the network is one of the more interesting features in the topology researched. By increasing connection speed between routers on this ring, a percentage of the packets traveling on the network would see a gain in speed when traveling across the ring. Research could be done as to what levels of traffic on the network would the enhanced ring speeds have an effect on the completion time of packets. With a fixed amount of smaller networks off the central ring, only so much traffic can be produced by the smaller network. Additional research could be done to determine at what speed of the ring would make it impossible for it to ever become fully congested.

By choosing a specific algorithm to use just on the ring, we could also effect the completion times of the packets. In our simulations, the routing algorithm on the ring was the same as used on networks connected to it. If improvement could be shown, which traffic levels on the network would this change affect in a positive manner?

Algorithms on the rest of the network topology can also be varied. Our experiment surrounded three simple routing algorithms. The third of which

optimized routes not only on length, but also speed, and congestion level at routers along its path. It was effective for networks that are under moderate load. With a very high capacity on a network, this algorithm may become less successful in optimizing the routes of packets. Running simulations with this algorithm would allow us to determine if a cutoff of effectiveness exists. We started to see this with during our simulations when using a large fixed data packet size. Would this same problem occur on a network with heavier traffic using the distributed packet sizes?

Testing of additional network routing algorithms on the network would allow us to see if a better solution exists for improving the effectiveness of a specific topology. By modeling an algorithm that at each hop the route was recalculated based on global conditions, we may be able to find an optimal routing algorithm. While we feel this would offer the best possible route for a packet to travel at any given point in time, this type of algorithm is not feasible to implement in present day routing hardware but may reveal other interesting properties of the ring's effect on our topology.

The contributions of the current research gives new insight into the effectiveness of the ring architecture for interconnecting smaller local area networks. Our research has shown that algorithm choice can have a large effect on the effectiveness of a network and that no one algorithm is the perfect solution in all cases. Our algorithms each performed effectively under certain conditions. When designing a network based upon a topology such as the one used in PPL's network, routing algorithm's should be studied to maximize efficiency of the network.

Chapter 5

Appendix

5.1 Network Generator Source

```
/**
 * Network File Generation and parsing software
 * Written by: Tyler Worman
 * Email: worman@cs.moravian.edu
 */

#include "fileClass.cc"
#include "rvgs.h"
#include <iostream>
#include <fstream>
#include <string>
#include <vector>

int nodeCounter = 0;

void createNode(int treeID, int level, int depth, router* parent,
network* myNet, int widthLow, int widthHigh, int probSpeed);
void makeConnection(router* router1, router* router2, int probSpeed);

using namespace std;

int main(int argc, char **argv) {
  //argv[1] = filename
  //argv[2] = circle of tree's
  //argv[3] = width lower
  //argv[4] = width upper
  //argv[5] = depth lower
  //argv[6] = depth upper
  //argv[7] = inter connection tolerance in percent style aka 70
  //argv[8] = Probability of 10/100 connections in percent style aka 70
```

```
  network *myNet = new network();
  int nodeID = 0; //Node counter when creating files.
  for (int i =0; i < atoi(argv[2]); i++) {
    //Create a root node
    router* newRouter = new router();
    newRouter->idNum = nodeCounter;
    newRouter->treeID = i;
    newRouter->levelID = 0;
    nodeCounter++;
    // Place node on the stack.
    myNet->nodes.push_back(newRouter);
    myNet->circle.push_back(newRouter);
    //Generate a target depth built tree recursively.
    int tempWidth = Equilikely(atoi(argv[3]),atoi(argv[4]));
    for (int j =0; j<tempWidth; j++) {
      int tempDepth = Equilikely(atoi(argv[5]),atoi(argv[6]));
      createNode(i, 1, tempDepth, newRouter, myNet,
 atoi(argv[3]),atoi(argv[4]),atoi(argv[8]));
    }
    cout << "Connecting the Loop" << endl;
    //Restart loop and generate another connecting it
    //to the last node in the circle.
    if (i != 0) {
      makeConnection(myNet->circle[i], myNet->circle[i-1], atoi(argv[8]));
     }
    //This is the last node in the tree... connect it back to the
    //first and complete the ring
    if (myNet->circle.size() - 1  == (atoi(argv[2]) - 1) && i !=0) {
      makeConnection(myNet->circle[i], myNet->circle[0], atoi(argv[8]));
    }
    cout << "Connecting to others in tree" << endl;
    //connect members to each other within this tree
    vector<router*>::iterator it;
    for (it = myNet->nodes.begin(); it != myNet->nodes.end(); ++it) {
      if ((*it)->treeID == i) {
vector<router*>::iterator it2;
for (it2 = myNet->nodes.begin(); it2 != myNet->nodes.end(); ++it2) {
  //check level
  if (((*it2)->treeID == i) && ((*it)->levelID == (*it2)->levelID)) {
    double probabil = Equilikely(0,100)/100.0;
    if(probabil >= atoi(argv[7])/100) {
      //check connections
      vector<connection*>::iterator it3;
      bool isFound = false;
      for (it3 = (*it)->routes.begin();
   ((it3 != (*it)->routes.end()) && (isFound != true));
   ++it3) {
if ((*it3)->toNode == (*it2)) {
```

```
  //a route already exists lets exit this loop
  isFound = true;
}
      }
      if (isFound != true) {
//Add a route.
bool isEdge = false;
vector<router*>::iterator itEdges;

for (itEdges = myNet->edges.begin();
     itEdges != myNet->edges.end();
     ++itEdges) {
  if(((*itEdges)->idNum == (*it2)->idNum) ||
     ((*itEdges)->idNum == (*it)->idNum)) {
    isEdge = true;
  }
}
if(isEdge == false) {
  makeConnection((*it), (*it2), atoi(argv[8]));
}
      } else {
//do nothing a route already exists.
      }
    } //probabillity
  } //Check for tree level
} //for to loop through the second set of nodes
      } //loop through all nodes
    }
  }
  //Lets print out the results to our file
  ofstream fileHandle(argv[1], std::ios::out);
  vector<router*>::iterator it4;
  vector<router*>::iterator it6;
  fileHandle << myNet->nodes.size() << " ";
  fileHandle << myNet->edges.size();
  vector<router*>::iterator itTemp;
  for (itTemp = myNet->circle.begin();
       itTemp != myNet->circle.end();
       ++itTemp) {
    //Write edges to file
    cout << " "  <<  (*itTemp)->idNum;
  }
  cout << endl;

  for (it6 = myNet->edges.begin(); it6 != myNet->edges.end(); ++it6) {
    //Write edges to file
    fileHandle << " "  <<  (*it6)->idNum;
  }
```

```
  fileHandle << endl;
  for (it4 = myNet->nodes.begin(); it4 != myNet->nodes.end(); ++it4) {
    vector<connection*>::iterator it5;
    fileHandle << (*it4)->idNum
       << " "
       << (*it4)->treeID
       << " "
       << (*it4)->levelID
       << " "
       << (*it4)->routes.size()
       << " ";
    for (it5 = (*it4)->routes.begin();
 (it5 != (*it4)->routes.end());
 ++it5) {
      fileHandle << ((*it5)->toNode)->idNum
 << ":"
 << (*it5)->speed
 << " ";
    }
    fileHandle << endl;
  }
  cout << "Network Created\n";
  exit(0);
}


void makeConnection(router* router1, router* router2, int probSpeed) {
  if (router1->idNum == router2->idNum) {
    return;
  }
  double probabil = Equilikely(0,10)/10.0;
  connection* myConnection = new connection;
  myConnection->toNode = router1;
  connection* myConnection2 = new connection;
  myConnection2->toNode = router2;
  if(probabil >= (probSpeed/100.0)) {
    //100
    myConnection->speed = 100;
    myConnection2->speed = 100;
  } else {
    //10
    myConnection->speed = 10;
    myConnection2->speed = 10;
  }
  //push connections to oppostire routers
  router1->routes.push_back(myConnection2);
  router2->routes.push_back(myConnection);
}
```

```
void createNode(int treeID, int level, int depth, router* parent,
network* myNet, int widthLow,
int widthHigh, int probSpeed) {
  int tempwidth = Equilikely(widthLow, widthHigh);
  //depth
  for (int i = 0; i < tempwidth; i++) {
    router *newChild = new router();
    newChild->treeID = treeID;
    newChild->idNum = nodeCounter;
    nodeCounter++;
    newChild->levelID = depth;
    //create parent to child connection
    makeConnection(parent, newChild, probSpeed);
    myNet->nodes.push_back(newChild);
    if (depth > 1) {
      createNode(treeID, level+1, depth -1, newChild, myNet,
 widthLow, widthHigh, probSpeed);
    } else {
      myNet->edges.push_back(newChild);
    }
  }

}
```

5.2 Network Data Type Source

```
/**
 * Network Simulation Data types and assistant classes
 * By: Tyler Worman
 * Email: worman@cs.moravian.edu
 */

#include <vector>
#include <queue>
#include <limits.h>
#include <iostream>
using namespace std;


//Section for Creation of network.
class router;

class connection {
public:
```

```
  router* toNode;
  int speed;
};

class router {
public:
  vector<connection*> routes;
  int idNum;
  int treeID;
  int levelID;
};

class network {
public:
  vector<router*> circle;
  vector<router*> nodes;
  vector<router*> edges;
};


//This is for sim.
class packet {
 public:
  int startNode;
  int endNode;
  deque<int> routeList;
  int currentNode;
  int hops;
  int idNum;
  double timeSoFar; //this stores time the packet was created
  double nodeArrivalTime; //this stores the time that the
  //packet arrived at current node..
  int packetSize;  //in bytes
};


class nextNode {
 public:
  double queuedAt;
  double serviceTime;
  packet* eventPacket;
};


class serverNode {
 public:
  deque<packet *> waitList;
  float waitTime;
```

```
  int numProcessed;
};

class routeFinder {
public:
  //set the local graph so paths can be found on it..
  void init(vector<router*> thegraph) {
    graph = thegraph;
  }

  deque<int> GetRouteList(int start, int end, int alg,
  deque<serverNode*>* routerList) {
    deque<int> tempRoute;
    float dist[graph.size()];
    int prev[graph.size()];
    int i;
    deque< int > graphUnfinished;
    deque< int > graphFinished;
    /*Init both arrays*/
    for (i = 0; i < graph.size(); i++) {
      dist[i] = INT_MAX;
      prev[i] = -1;
      graphUnfinished.push_back(i);
    }
    //This sets the starting node's idNum to 0
    dist[start] = 0;
    int v = start; //current vertex
    deque< int >::iterator it2;
    while (graphUnfinished.size() != 0) {
      deque< int >::iterator itTemp; //tempt it;
      for (it2 = graphUnfinished.begin();
   it2 != graphUnfinished.end();
   it2++) {
//on first time through make v the first
//then loop to find the shortest.
if((dist[(graph[(*it2)])->idNum] < dist[v]) ||
   it2 == graphUnfinished.begin()) {
  v = graph[(*it2)]->idNum;
  itTemp = it2;
}
      }
      //Add V to processed list.
      graphFinished.push_back(v);
      //For each edge in v, relax them...
      vector< connection* >::iterator it3;
      for(it3 = graph[v]->routes.begin();
  it3 != graph[v]->routes.end();
  it3++) {
```

```
//this if determins the weights... for each alg
int v2 = ((*it3)->toNode)->idNum;
if(alg == 2) {
  //If statement for speed sensative alg
  vector<connection*>::iterator connIt;
  int speed = 0;
  for (connIt = graph[(*itTemp)]->routes.begin();
       connIt != graph[(*itTemp)]->routes.end();
       connIt++) {
    if((*connIt)->toNode->idNum == v2) {
      speed = (*connIt)->speed;
    }
  }
  if (speed == 0) {
    cout << "UHOH!" << endl;
  }
  if ((dist[v] + (1/speed)) < dist[v2]) {
    dist[v2] = dist[v] + (1/speed);
    prev[v2] = v;
  }
} else if(alg == 3) {
  //If statement for congestion optimization alg
  vector<connection*>::iterator connIt;
  deque<packet *>::iterator sizeIt;
  vector<connection*>::iterator speedIt;
  int sizePackets =0;
  int avgLineSpeed =0;
  int speed =0;
  for (sizeIt = (*routerList)[v2]->waitList.begin();
       sizeIt != (*routerList)[v2]->waitList.end();
       sizeIt++) {
    sizePackets += (*sizeIt)->packetSize;
  }


  for (connIt = graph[(*itTemp)]->routes.begin();
       connIt != graph[(*itTemp)]->routes.end();
       connIt++) {
    if((*connIt)->toNode->idNum == v2) {
      for(speedIt = (*connIt)->toNode->routes.begin();
  speedIt != (*connIt)->toNode->routes.end();
  speedIt++) {
avgLineSpeed += (*speedIt)->speed;
      }
      //assumes every node has 1 route..
      avgLineSpeed /= (*connIt)->toNode->routes.size();
      speed = (*connIt)->speed;
    }
```

```
  }

  if (speed == 0) {
    cout << "UHOH!" << endl;
  }
  if ((dist[v] + (1/speed) + (sizePackets/avgLineSpeed)) < dist[v2]) {
    dist[v2] = dist[v] + ( 1/speed )+ (sizePackets/avgLineSpeed);
    prev[v2] = v;
  }

} else if(alg == 1) {
  //If statement for shortest path algorithm
  if(dist[v] + 1 < dist[v2]) {
    dist[v2] = dist[v] + 1;
    prev[v2] = v;
  }
} else {
  cout << "something didn't go right" << endl;
}
      }
      it2 = graphUnfinished.erase(itTemp);
    }
    //shortest path in table. Load it to a route list
    tempRoute.push_front(end);
    int myVal = end;
    while(myVal != start) {
      tempRoute.push_front(prev[myVal]);
      myVal = prev[myVal];
    }
    tempRoute.pop_front();
    return tempRoute;
  }



private:
  deque< deque<int> > cachedRoutes;
  vector<router*> graph;
};
```

5.3 Network Simulation Software Source

```
/**
 * Network Packet Simulation
 * By: Tyler Worman
 * Email: worman@cs.moravian.edu
 */
```

```
#include <stdio.h>
#include <math.h>
#include "rngs.h"           /* the multi-stream generator */
#include "rvgs.h"
#include <queue>
#include <vector>
#include <iostream>
#include <fstream>
#include <string>
#include <iomanip>
#include "fileClass.cc"

#define START         0.0       /* initial time                    */
#define STOP       30.0         /* terminal (close the door) time */
#define INFIN   (100.0 * STOP)  /* must be much larger than STOP  */

using namespace std;

//Globals
vector<router*> pathGraph;
deque<serverNode*> routerList;
vector<int> edgeList;

// End Globals

/**
 * This function is used to compare nodes for stl sort
 */
bool cmpNodes(const nextNode *a, const nextNode *b) {
  return a->queuedAt < b->queuedAt;
}


/* ------------------------------
 * return the smaller of a, b
 * ------------------------------
 */
double Min(double a, double c) {
  if (a < c) return (a);
  else return (c);
}

/* ---------------------------------------------
 * generate the next arrival time, with rate 1/2
 * ---------------------------------------------
 */
double GetArrival() {
```

```
  static double arrival = START;
  SelectStream(0);
  arrival += Exponential(.00035);
  return (arrival);
}


/* -------------------------
 * Gets packet size in bytes
 */
int GetPacketSize(int mode) {
  if (mode == 2) {
    //This is for constant size discussed in paper
    return 583;
  } else {
    //This is for distributed size.
    double chance = Equilikely(0, 100);
    if (chance < 51.3) {
      return 50;
    } else if (chance < 82.6) {
      return 1500;
    } else {
      return (int)Equilikely(50,1500);
    }
  }
}


/* ----------------------------------------
 * Get's an edge from the list
 * ----------------------------------------
 */
int GetEdge(int edges) {
  if (edges == 0) {
    cout << "NO EDGES!" << endl;
  }
  return (int)(Equilikely(0, edges -1));
}

/* ----------------------------------------
 * generate the next service time with rate 2/3
 * ----------------------------------------
 */
double GetService(int router1, int nextNodeNum, int packetSize) {
  // routerList[router1].waitTime
  // pathGraph[router1]

  //return (double)((packetSize) / (10 *1048576 ));
```

```
  vector<connection*>::iterator it;
  for (it = pathGraph[router1]->routes.begin();
       it != pathGraph[router1]->routes.end();
       ++it) {
    if((*it)->toNode->idNum == nextNodeNum) {
      return ((double)(packetSize) / (((*it)->speed) * 1048576));
      break;
    }
  }
  cout << "ERROR" <<  router1 << " " << nextNodeNum <<  endl;
  return 1.0;
}


int main(int argc, char **argv) {
  //ARGS
  //ARGV1 = Filename
  //ARGV2 = Packet Output Filename
  //ARGV3 = Which alg 1,2,3
  //ARGV4 = Locked, Distributed packets
  //Load file into my list of nodes...
  ofstream fileHandle(argv[2], std::ios::out); //open the output file.
  ifstream myfile (argv[1]);
  if (myfile.is_open()) {
    int edges = 0;
    int nodeNum = 0;
    myfile >> nodeNum;
    myfile >> edges;
    for (int i = 0; i < edges; i++) {
      int temp;
      myfile >> temp;
      edgeList.push_back(temp);
    }
    //create a bunch of empty nodes...
    for (int i = 0; i < nodeNum; i++) {
      router* myNode = new router();
      myNode->idNum = i;
      pathGraph.push_back(myNode);
    }
    for (int i = 0; i < nodeNum; i++) {
      serverNode* myNode = new serverNode();
      routerList.push_back(myNode);
    }
    //read in nodes and their connections then link them up....
    for (int i = 0; i < nodeNum; i++ ) {
        int connNums = 0;
int tempNum = 0;
myfile >> tempNum;
```

```
if(pathGraph[i]->idNum == tempNum) {
  myfile >> pathGraph[i]->treeID;
  myfile >> pathGraph[i]->levelID;
  myfile >> connNums;
  for (int j =0; j < connNums; j++) {
    char temp;
    connection* newConn = new connection();
    int tempConnNode;
    myfile >> tempConnNode;
    newConn->toNode = pathGraph[tempConnNode];
    myfile >> temp;
    myfile >> newConn->speed;
    pathGraph[i]->routes.push_back(newConn);
  }

} else {
  cout << "HORRIBLE DATA ERROR!";
}
    }
    myfile.close();
  }
  //done reading data



  //Setup The Data Statistics Parts
  struct {
    double arrival;                /* next arrival time */
    double current;                /* current time      */
    double next;                   /* next event time     */
    double completion;
    double last;
  } t;
  struct {
    double node;                /* time integrated number in the node  */
    double queue;               /* time integrated number in the queue */
    double service;             /* time integrated number in service   */
  } area      = {0.0, 0.0, 0.0};

  long index  = 0;              /* used to count departed jobs         */
  long number = 0;              /* number in the node                  */
  long iterations = 0;

  //Setup queue
  deque<nextNode*>  myQueue;
  nextNode* node = new nextNode;
  PlantSeeds(123456789);
```

```
  //Setup Router Network
  routeFinder getRoutes;
  getRoutes.init(pathGraph);
  //Setup Timer
  t.current    = START;           /* set the clock                          */
  t.arrival    = GetArrival();    /* schedule the first arrival             */
  t.completion = INFIN;        /* the first event can't be a completion */
  while ((t.arrival < STOP) || (number > 0)) {
    t.next          = Min(t.arrival, t.completion);


    //find ammount in each node...
    int numNodes =0;
    int nodesInUse = 0;
    for (int i; i < routerList.size(); i++) {
      numNodes += routerList[i]->waitList.size();
      if (routerList[i]->waitList.size() > 0) {
nodesInUse++;
      }
    }
    area.node    += (numNodes / routerList.size());
    area.queue   +=  myQueue.size();
    area.service += (nodesInUse); // routerList.size());
    t.current       = t.next;                    /* advance the clock */
    iterations++;

    // ARRRIVALS!
    if (t.current == t.arrival)  {
      index++;
      t.arrival     = GetArrival();
      //This is the arrival of the next event...
      //Create a packet
      packet* arrPacket = new packet();
      arrPacket->startNode   = edgeList[GetEdge(edgeList.size())];
      arrPacket->endNode     = edgeList[GetEdge(edgeList.size())];
      //incase they are somehow duplicates...
      while (arrPacket->endNode == arrPacket->startNode) {
      arrPacket->startNode   = edgeList[GetEdge(edgeList.size())];
      arrPacket->endNode     = edgeList[GetEdge(edgeList.size())];
      }
      arrPacket->routeList = getRoutes.GetRouteList(arrPacket->startNode,
    arrPacket->endNode,
    atoi(argv[3]),
    &routerList);
      SelectStream(1);
      arrPacket->packetSize  = (int)GetPacketSize(atoi(argv[4]));
      //Note packet size in bytes
      arrPacket->hops = 0;
```

```
      arrPacket->idNum = index;
      arrPacket->timeSoFar = t.current;
      arrPacket->currentNode = arrPacket->startNode;
      arrPacket->nodeArrivalTime = t.current; //store this for the 3rd alg
      //Place it on the Router Lists...
      routerList[arrPacket->currentNode]->waitList.push_back(arrPacket);
      if(routerList[arrPacket->currentNode]->waitList.size() == 1) {
//If size is = to 1 then we should go and schedule this
node = new nextNode();
node->eventPacket = arrPacket;
node->serviceTime = GetService(arrPacket->currentNode,
       arrPacket->routeList.front(),
       arrPacket->packetSize);
//cout << node->serviceTime << endl;
node->queuedAt = t.current;
myQueue.push_back(node);
number++;
      } else {
//Queue was larger then 1
//so it will be scheduled later for now it just sits.
      }
      //Sort by times.
      sort(myQueue.begin(), myQueue.end(), cmpNodes);
      t.completion = t.current + myQueue.front()->serviceTime;
      if (t.arrival > STOP)  {
        t.last      = t.current;
        t.arrival   = INFIN;
      }
    } else {                  /* process a completion */
      if (number > 0) {
int tempPacketLoc = myQueue.front()->eventPacket->currentNode;
if (myQueue.front()->eventPacket->routeList.front() ==
    myQueue.front()->eventPacket->endNode) {
  //Update data from the ending jump
  myQueue.front()->eventPacket->hops++;

  //Packet is done and just needs to vanish and load up the next..
  //outputdata
  fileHandle << myQueue.front()->eventPacket->hops
     << " "
     << setw(20)
     << fixed
     << (t.current - myQueue.front()->
 eventPacket->timeSoFar)
     << " "
     << myQueue.front()->eventPacket->startNode
     << " "
     << myQueue.front()->eventPacket->endNode
```

```
     << " "
     << myQueue.front()->eventPacket->packetSize
     << endl;
//Increase the ammount of packets this node has processed
  routerList[tempPacketLoc]->numProcessed++;
  routerList[tempPacketLoc]->waitTime += t.current -
    myQueue.front()->eventPacket->nodeArrivalTime;

  //Remove packet and update stats.
  routerList[tempPacketLoc]->waitList.pop_front();
  myQueue.pop_front();
  number--;

  if((routerList[tempPacketLoc]->waitList.size()) >= 1) {
    //Still events left in router
    //Figure out the service Time
    node = new nextNode();
    node->eventPacket = routerList[tempPacketLoc]->waitList[0];
    node->serviceTime =
      GetService(node->eventPacket->currentNode,
 node->eventPacket->routeList.front(),
 node->eventPacket->packetSize);
    node->queuedAt = t.current;
    myQueue.push_back(node);
    std::sort(myQueue.begin(), myQueue.end(),cmpNodes);
    number++;
  }
} else {
        //Packet Isn't done it must be moved and rescheduled.
  int goingTo = myQueue.front()->eventPacket->routeList.front();
  int currAt = myQueue.front()->eventPacket->currentNode;
  routerList[goingTo]->waitList.push_back(myQueue.front()->
  eventPacket);
  myQueue.front()->eventPacket->currentNode = goingTo;
  //Update routes to go left in list
  myQueue.front()->eventPacket->routeList.pop_front();
  myQueue.front()->eventPacket->hops++;

  //Increase the ammount of packets the current node has processed
  routerList[currAt]->numProcessed++;
  routerList[currAt]->waitTime += t.current -
    myQueue.front()->eventPacket->nodeArrivalTime;
  //this takes care of setting the packet arrival
  //time when it's queued on the new node.
  myQueue.front()->eventPacket->nodeArrivalTime = t.current;
  //Pop off old one...
  myQueue.pop_front();
  number--;
```

```
  routerList[currAt]->waitList.pop_front();

  //Reschedule event at old router if any exist.
  if(routerList[currAt]->waitList.size() >= 1) {
    //Still events left in router
    //Figure out the service Time
    node = new nextNode();
    node->eventPacket = routerList[currAt]->waitList[0];
    node->serviceTime =
      GetService(node->eventPacket->currentNode,
 node->eventPacket->routeList.front(),
 node->eventPacket->packetSize);
    node->queuedAt = t.current;
    myQueue.push_back(node);
    number++;
  }

  //Reschedule event at new router if it's first
  if(routerList[goingTo]->waitList.size() == 1) {
    //Still events left in router
    //Figure out the service Time
    node = new nextNode();
    node->eventPacket = routerList[goingTo]->waitList[0];
    node->serviceTime =
      GetService(node->eventPacket->currentNode,
 node->eventPacket->routeList.front(),
 node->eventPacket->packetSize);
    node->queuedAt = t.current;
    myQueue.push_back(node);
    number++; //Queue size is added one..
  }
  //Resort the Queue
  sort(myQueue.begin(), myQueue.end(), cmpNodes);
}
if (number >0) {
t.completion = t.current + myQueue.front()->serviceTime;
} else {
  t.completion = INFIN;
}
//cout << myQueue.size() << endl;
      } else {
        t.completion = INFIN;
      }
    }
  }
  printf("for %ld jobs\n", index);
  printf("Iteratiorns: %d", iterations);
  printf("   average interarrival time = %6.2f\n", t.last / iterations);
```

```
  printf("   average # in the nodes .. = %6.2f\n", area.node / iterations);
  printf("   average # in the queue .. = %6.2f\n", area.queue / iterations);
  printf("   utilization ............. = %6.8f\n", area.service);

  return (0);
}
```

Bibliography

[1] Cpsc 641 performance issues in high speed networks. http://pages.cpsc.ucalgary.ca/ carey/CPSC641/ass2.html.

[2] Ppl corporation. http://www.pplweb.com.

[3] Ishai Ben Aroya, Ilan Newman, and Assaf Schuster. Randomized single-target hot-potato routing. *Journal of Algorithms*, 23:101–120, 1997.

[4] Yossi Azar, Edith Cohen, Amos Fiat, Haim Kaplan, and Harald Racke. Optimal oblivious routing in polynomial time. *Journal of Computer and System Sciences*, 69:383–394, 2004.

[5] K. Chi, C. Yang, and X. Wang. Performance of coding based multicast. In *IEE Proceedings – Communications*, volume 153, pages 399–404, 2006.

[6] Douglas E. Comer. *Computer Networks and Internets with Internet Applications*. Pearson Prentice Hall, Upper Saddle River, New Jersey, 2004.

[7] Curt Franklin. How routers work. http://computer.howstuffworks.com/router.htm.

[8] M. R. HoseinyFarahbady and H. Sarbazi-Azad. The grid-pyramid: A generalized pyramid network. *Journals of Supercomputing*, 37:23–45, 2006.

[9] Lawrence M. Leemis and Stephen K. Park. *Discrete-Event Simulation: A First Course*. Pearson Prentice Hall, Upper Saddle River, New Jersey, 2006.

[10] Michael Mitzenmacher. Bounds on the greedy routing algorithm for array networks. *Journal of Computer and System Sciences*, 53:317–327, 1996.

[11] Oystein Ore. *Graphs and their Uses*. The Mathematical Association of America, New York, NY, 1990.

[12] Roozbeth Razavi. How routing algorithms work. http://computer.howstuffworks.com/routing-algorithm.htm.

[13] George Varghese. *Network Algorithms: An interdisciplinary approach to designing fast networked devices*. Morgan Kaufmann, San Francisco, CA, 2005.

[14] Xiaofan Yang, Graham M. Megson, and David J. Evans. An oblivious shortest-path routing algorithm for fully connected cubic networks. *Journals of Parallel and Distributed Computing*, 66:1294–1303, 2006.

[15] Chun yen Chou, D. J. Guan, and Kuei lin Wang. A dynamic fault-tolerant message routing algorithm for double-loop networks. *Information Processing Letters*, 70:259–264, 1999.

[16] B. Zhang and H.T. Mouftah. Fast bandwidth-constrained shortest path routing algorithm. In *IEE Proceedings – Communications*, volume 153, pages 671–674, 2006.