Project Number: MLC-MQP-BM00

BETTER-BEHAVED MULTIMEDIA NETWORKING

A Major Qualifying Project Report:

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

By

Jason M. Ingalsbe

Keith R. Barber

Joel M. Thibault

Date: March 1, 2001

Approved:

1. Multimedia
2. Protocol
3. Internet

Professor Mark L. Claypool, Major Advisor

Abstract

The Internet was not designed with multimedia in mind. TCP is not well suited for multimedia and UDP is un-responsive in the face of congestion. MM-Flow, a rate-based protocol that responds to congestion, has not been thoroughly tested. We improve MM-Flow, perform an extensive analysis, and explore what it means to be a TCP-Friendly protocol. We find the new MM-Flow performs better over a wide range of network conditions.

Acknowledgements

We would like to thank Prof. Mark Claypool and Jae Chung for all their time and effort put in to helping us. Without their support and expertise this project never would have materialized.

Table of Contents

List of Figures

1 Introduction

The Internet is quickly becoming a way of life. Originally designed for text-based traffic, the Internet is increasingly serving as a medium for multimedia applications streaming video and audio, creating vast opportunities for communication and exchange of information. Radio and television broadcast, video-conferencing, and virtual classrooms are just a few of the benefits of multimedia over the Internet. Unfortunately, the underlying network structure allowing for these applications is faced with some inherent problems. Broadband technology is becoming increasingly available to consumers, but overall demand on the Internet is growing faster than the network can support, which leads to congestion and poor performance.

Text-based traffic, such as e-mail and HTML web pages, uses a protocol known as TCP, which recognizes congestion and reduces its sending rate appropriately. Multimedia traffic, on the other hand, has different performance requirements that make TCP, for reasons that will be discussed, a poor choice for multimedia. Instead, multimedia typically uses a protocol known as UDP. Unfortunately, UDP ignores congestion and has the potential for receiving more than its fair share of bandwidth while TCP is prevented from receiving its fair share.

An area that needs considerable research is the issue of congestion control. Congestion is typically measured in the form of packet loss. When a packet travels from the sender to the receiver it must go through a number of routers. The job of the router is to send packets along one of its outgoing lines such that it is directed toward the receiver. Routing tables are used to determine the best path. The problem is that routers have a set queue size, which means they can only hold a certain number of packets at any point in time, and the outgoing lines have a limited bandwidth. Congestion occurs at the router when the rate of incoming packets is faster than the rate of outgoing packets and the queue fills up. Once the queue is full the router can no longer handle the incoming packets, which are then dropped.

If the end hosts always sent packets at the fastest rate possible, routers would constantly be overloaded, packets would continually be dropped, and nothing would get done. TCP recognizes congestion in the form of packet loss and reduces its sending rate through a process known as Additive Increase Multiplicative Decrease (AIMD), which means that the sending rate is cut in half when a packet loss is detected and then slowly climbs again. This has proven extremely effective for text-based traffic. As stated, UDP ignores congestion and continues sending at its specified rate. The end result is that competing TCP and UDP flows will eventually cause congestion. TCP will reduce its sending rate, allowing UDP to take all of the available bandwidth. This situation is described as "starving" the TCP flow. While this situation may be desirable for the multimedia user, it is considered unacceptable because much of the traffic on the Internet travels across TCP.

If UDP causes so many problems with congestion, then one may wonder why multimedia does not use TCP. First, multimedia applications do not need to be "reliable", meaning that they can tolerate some data loss. This is due to fact that human beings can tolerate some loss without becoming annoyed. TCP is a reliable flow, meaning it guarantees that all packets are delivered through retransmission. This is unnecessary in a multimedia application, and therefore would be wasting valuable bandwidth. Furthermore, multimedia is extremely sensitive to jitter, or variation in inter-packet arrival time. In other words, if frames do not arrive at a consistent rate the user will notice choppiness, which is then perceived as poor quality. TCP only makes this situation worse through retransmission. If a packet is retransmitted it arrives at the receiver considerably later than when it is needed, thus contributing to jitter. Similarly, TCP's aggressive approach to AIMD causes significant fluctuations in transmission rate, which also leads to jitter. As a result, UDP is simply a better choice for multimedia.

If TCP is too responsive and UDP is not responsive enough, a possible solution would be to compromise in the form of a "TCP-friendly" protocol, meaning that it will not starve the TCP flow. While the notion of TCP-friendliness is easy to grasp, the difficulty lies in measuring it and determining whether a particular protocol is really TCP-friendly. In later sections of this paper, we will examine the nature of TCP-friendliness, but first we must discuss some existing protocols that claim to be TCP-friendly.

The creators of TCP-Friendly Rate Control (TFRC) [FHPW2000] introduced one approach to bridging the gap between TCP and UDP. The idea behind this protocol is to react to congestion but not as quickly and as drastically as TCP, thereby providing a smoother sending rate. For example, instead of tracking packet loss TFRC tracks "loss events," meaning that multiple consecutive packet drops are considered as a single packet. The receiver calculates the loss event rate using the "average loss interval" method to compute a weighted average of the loss rate over the last n loss intervals, with equal weights on each of the most recent $n/2$ intervals. The receiver then reports this information back to the sender via an ACK (acknowledge) packet at least once per round-trip time, assuming it has received packets within that interval. The sender uses this loss event rate to determine the sending rate. If the sender does not receive any ACKS within several round trip times it assumes congestion and reduces the sending rate.

The creators of TCP Emulation At Receivers (TEAR) [OY2000], a rival to TFRC, have proposed another solution. TEAR has the same goal as TFRC—respond to congestion while providing a smoother sending rate—but uses a slightly different approach. TEAR determines that the sender's role is simply to send a packet. Therefore, all calculations, including loss rate and sending rate, are calculated at the receiver. Whenever the sending rate should be changed the receiver sends an ACK packet back to the sender with the new sending rate. Since the receiver only sends packets back to the sender when it requests to either speed up or slow down the sending rate, it lessens the amount of data being sent back to the sender, thus using less bandwidth. The creators of this protocol argue that they have an advantage over TFRC in a multicast environment, in that the sender will not be constantly bombarded with ACKS from multiple receivers, but rather only when a receiver indicates a need to change the sending rate. In addition, the computational burden of rate calculation is spread among the receivers instead of concentrated at the sender.

A third approach is known as MM-Flow and suggests that TCP-friendly applications can be built on top of UDP [CC2000]. There are a few major differences between MM-Flow and TFRC or TEAR that are worth mentioning. First, since congestion control is found in the application layer, congestion is determined by *frame* loss rather than *packet* loss. Second, the receiver determines the sending rate in the form of a *scale value* and ACKs this value back to the sender. The scale value is important because, unlike TFRC and TEAR, MM-Flow was designed to support different types of multimedia applications. The scale value provides a generic reference that can then be mapped to the desired encoding scheme. For example, TFRC and TEAR seem to assume the application layer is sending with a fixed frame size and variable rate. MM-Flow, on the other hand, has been designed with two different applications in mind—MM-App, which sends frames of fixed size at variable rates, and MPEG-App, which follows the MPEG encoding standard of variable frame size with constant rate.

MM-Flow serves as the foundation of our project. While initial tests suggested that MM-Flow is more TCP-friendly than UDP, it had not undergone exhaustive testing. As described in the next section, we first re-engineered MM-Flow somewhat to separate the protocol decisions into an actual transport layer, which measures congestion at the packet level rather than the frame level. The application layer now sits directly on top of our transport layer, no longer needing UDP. We hypothesize that this transition yields an increase in performance. We also designed the application layer such that it is easier to add new types of encoding schemes. Next we thoroughly tested the MM-Flow protocol to look for improvements, such as including some of the strengths of TFRC and TEAR.

In this paper we examine some of the issues involved with multimedia applications over the Internet and some of the proposed solutions. Through inspection of existing solutions we develop and test an improved protocol for multimedia applications that is considered TCP-friendly while still providing acceptable multimedia quality to the user.

The chapters to follow go into depth regarding how we changed the MM-Flow protocol, including the specific modifications we made to both the application and transport layers. Upon making these changes, we continually tested MM-App against other protocols using simulations using a variety of network scenarios, as will be shown. These simulations provided us with the results that form the bulk of this paper, and will be discussed near the end. We discovered a number of new topics that need exploration; these ideas will be discussed in the future work chapter.

2 Approach

2.1 Re-Engineering of MM-Flow

Before analyzing and testing protocols, we first re-organized MM-Flow. Originally, the MM-Flow project integrated a networking protocol with a multimedia application and so had to be considered as a unit. We felt that breaking down MM-Flow into a transport layer and an application layer was beneficial, as it made each layer independent and MM-Flow became easier to compare with other protocols. After making these changes to MM-Flow, we ran tests on it and the older version of MM-Flow to make sure that the functionality had not been changed. Next, changes to how MM-Flow worked were considered.

The initial version of the MM-Flow system contained most of the logic at the application layer, with the transport layer's function only to separate the frames into packets and send them across the network. In order to make the MM-Flow protocol more universal, it was necessary to move flow control and scale adjustment algorithms to the transport layer. Applications that make use of MM-Flow no longer have to re-implement this functionality. Instead, the application specifies a range and number of transmission scale values. The transport layer then constantly assesses network conditions to decide which scale is most appropriate. Periodically the application layer will query the transport layer to discover which scale it should use, and acts accordingly. One example application, MM-App, gets the current scale value after each time a frame is sent, and uses it to calculate when the next frame should be sent. Another application, MPEG-App, gets the scale value before a frame is sent, and uses it to determine how many frames it can send, according to the MPEG specification.

When the transport layer became responsible for determining the scale value, we needed a new system for adjusting that value. For the receiver to determine the proper scale value, as before, it would need to know which scale values were appropriate. The sender at the application layer uses the scale values, so rather than specifying a means for communicating scale values to the receiver, it was simpler to make scale adjustments occur at the sender. The receiver's role then became simply signaling, via ACK or NACK (negative acknowledge) packets, whether the sender should increase or decrease sending rate, respectively. Upon receiving an ACK/NACK message, the sender adjusts its scale according to AIMD.

The default number of rates was changed as well. The original protocol was fixed at five scale values, for rates ranging from 300 Kbps to 1.5 Mbps. We observed that the rate fluctuated a good deal and reasoned that increasing the number of rates would enable the protocol to respond more precisely; the "fair share" of bandwidth taken up by MM-Flow can be more closely approximated when there are more choices available for rates. The simulation designer may now set how many rates to use, and how much bandwidth to occupy at the highest rate.

In addition, we implemented two algorithms for determining what the current scale value should be at a given time. The first method returns the standard scale as determined by the AIMD configuration. Since one of the goals of MM-Flow is to have less variation than TCP, the protocol also calculates a weighted average of recent scale values by using a history window. The hope is that a single dropped packet (which normally causes the rate to be cut in half) will not cause so drastic a change in rate so quickly, while at the same time MM-Flow will still respond to more serious problems quickly. Simulation designers can choose whether to use the weighted scale or the standard AIMD-based scales through the MM-Flow interface.

In addition, we implemented two algorithms for determining what the current scale value should be at a given time. The first method returns the standard scale as determined by the AIMD configuration. Since one of the goals of MM-Flow is to have less variation than TCP, the protocol also calculates a weighted average of recent scale values. The hope is that a single dropped packet (which normally causes the rate to be cut in half) will not cause so drastic a change in rate, while at the same time MM-Flow will still respond to more serious problems quickly. Simulation designers can choose whether to use the weighted scale or the standard AIMD-based scales through the MM-Flow interface.

The weighted scale is implemented by means of an 8-slot history window. Each time the new scale is calculated by AIMD, its value is placed at the head of this window with the oldest value being removed, ensuring that the array always contains the 8 most recent scales. The weighted scale is then equal to the nearest integer to the result of this formula:

Weighted scale = (20% * most recent) + (15% * 2nd most recent) + (15% * 3rd most recent) + (15% * 4th most recent) + (15% * 5th most recent) + (10% * 6th most recent) + (10% * 7th most recent) + (10% * 8th most recent)

Equation 2-1: Weighted Scale Calculation

2.2 Exploring TCP-Friendliness

In addition to re-engineering MM-Flow, our project explored the nature of TCP-Friendliness. First, we tried to determine the meaning of a “TCP-friendly” protocol. This task proved more difficult than one would expect. Many protocols claim to be TCP-friendly, without supporting their claim with an explanation or measurements. For example, it might be acceptable to say a flow is TCP-friendly if it uses a fair share of the bandwidth, meaning the total bandwidth divided by the number of flows. Another solution might be to say a flow is TCP-friendly if it uses the same bandwidth as TCP under the same network conditions. Perhaps a flow is TCP-friendly if it at least responds to congestion, unlike UDP. Eventually, our search for an answer lead us to the equation

$$T \le \frac{1.5\sqrt{2/3} * B}{R * \sqrt{p}}$$

Equation 2-2: TCP-Friendly Transmission Rate

where T is the maximum sending rate of a conformant TCP flow in Bps, B is the packet size in bytes, R is the round trip time in seconds, and p is the per-flow packet loss rate [FF1999].

This equation calculates the maximum arrival rate at the router for a TCP flow when given the packet size, round trip time, and drop rate. The authors claim that a flow is TCP-friendly if it arrives at a rate less than or equal to the value specified by the equation. We then used this equation to compute TCP-friendly measurements for each of our simulations. These results, as well as an in depth discussion of the formula, are presented in chapter 4, Results.

3 Evaluation Techniques

In order to accurately measure how a protocol acts within a networking environment we needed to use a simulator. The simulator we used is NS version 2.1b7. NS is an object-oriented, discrete event driven network simulator developed at UC Berkeley written in C++ and OTcl. NS is primarily useful for simulating local and wide area networks [perform]. It allows us to run our protocols against other protocols such as TCP, UDP, and TFRC, by writing OTcl scripts. By configuring the OTcl scripts we are able to test these protocols under different conditions that may provide insight into their behavior. With the simulator we are able to track the data of all running protocols. We then run our data collection scripts against the results provided by NS in order to obtain useful data that can be analyzed.

3.1 Simulation Scenarios

Our simulation scenarios include the standard bottleneck layout, standard delay layout, standard fragile flow layout, and the multi-protocol layout. In the first three, we used one protocol as our base for comparison with our protocols and others. This base protocol is TCP, as one of our main goals is to create a protocol that is TCP-friendly. Each layout can be used in numerous tests simply by changing values within the protocol or switching the roles of the protocols we are looking at.

The standard bottleneck layout, as shown in Figure 3-1, contains four nodes in a system. Nodes numbered zero and one correspond to the senders of the two protocols to be tested. Both of these are configured to send information across the link to node number three, the receiver in both protocols. Node two contains our router. Since both protocols must send via node two, a bottleneck is created there. Most of our measurements are taken from the packets traveling between node two to node three. The length of the line connecting two nodes represents the delay between them. As shown in the figure, all of the delays are equal in this scenario. This allows us to observe the interactions of the two protocols as they operate under the same conditions.

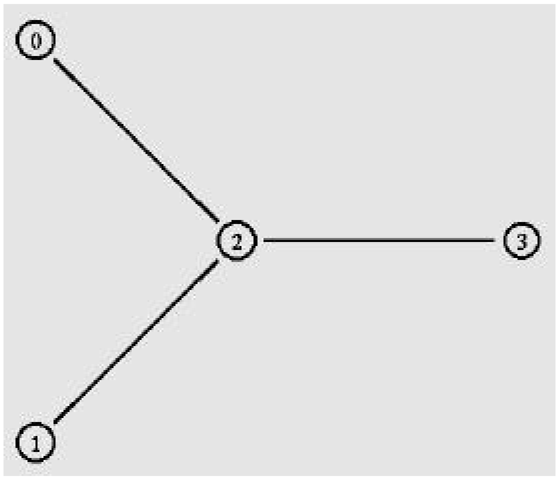

Figure 3-1: Standard Bottleneck Layout

The second major layout that we use is shown in Figure 3-2, the standard delay layout. This is similar to the bottleneck layout in that nodes zero and one still correspond to the senders of the two protocols and have the same delay as node two, as shown by the length of the lines connecting them in the diagram. The difference is that the bottleneck delay is greater. The purpose of such a layout is to determine how a protocol reacts to longer round trip times. This is worth looking at since the time required to receive an ACK or NACK is increased, which might decrease responsiveness under congestion.

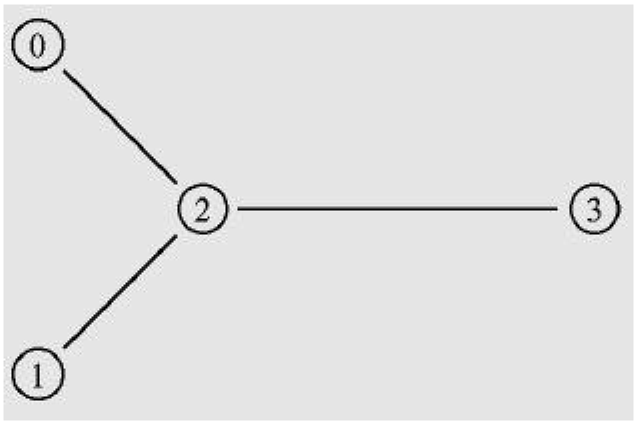

Figure 3-2: Standard Delay Layout

The third major layout that we use throughout our testing procedures is shown in Figure 3-3, the standard fragile layout. In this layout the two protocols are on nodes zero and one, as before, but they have different delays to the router. As one can see in the picture, the delay between node zero and two is significantly shorter than between node one and node two. This allows us to test how the protocols react when another protocol is at an advantage with regards to round trip time. The delay between node two and three is the same as in the standard bottleneck layout.

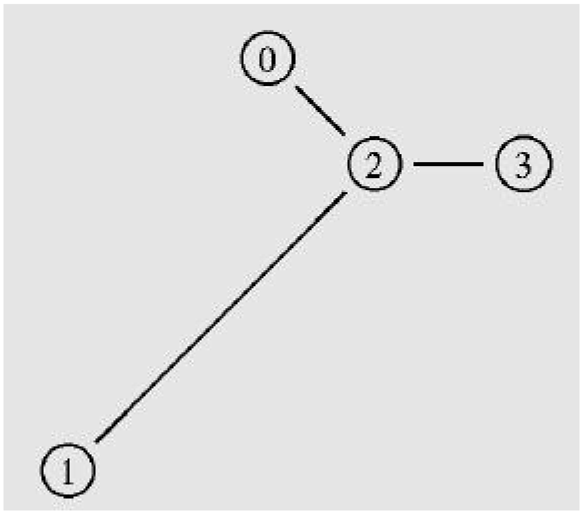

Figure 3-3: Standard Fragile Layout

The fourth scenario, the multi-protocol layout, is used to get a more "real-world" perspective. This layout, as shown in Figure 3-4, has a total of eight sending nodes, with eight receiving nodes. Nodes zero through seven have the sending protocols. Each sender node has only one receiver node. For example, node zero sends to node eight, node one sends to node nine, etc. The simulation has a total of four TCP flows, two MM-App-New flows, and two TFRC flows all running at once. Nodes zero through three run TCP, four and five run MM-App-New, and six and seven run TFRC.

This simulation provides us with a more realistic simulation than the rest because in the "real-world" there will not be just two flows running against each other over a network link, but rather there will be numerous flows all running at once going to different places. All the links in the simulation are 4 Mbps, and the delays for all links are 5ms, except for the bottleneck link between nodes 16 and 17, which is 20ms.

The picture of the layout for this simulation is somewhat distorted due to the fact that there is such a large number of nodes and that the delays are all small. Also, nodes 11 and 15 appear in the same place, between nodes 10 and 12, but are in fact two different nodes.

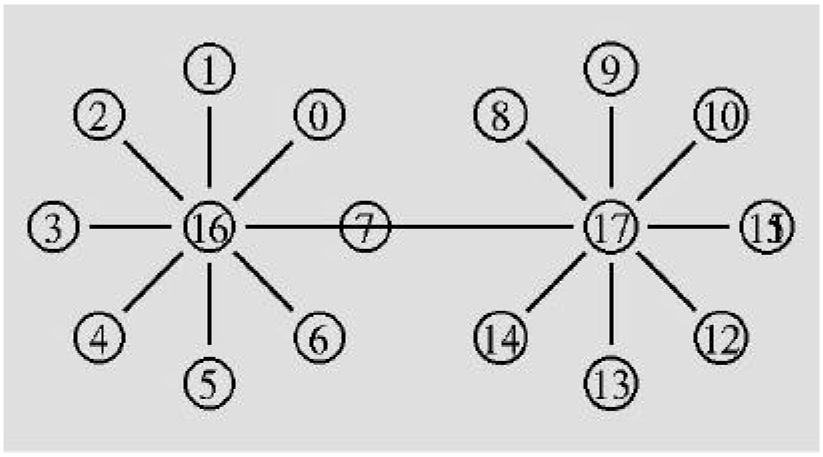

Figure 3-4: Multi-Protocol Layout

3.2 Data Collection Scripts

After creating simulation scenarios, the next step in simulating network traffic is obtaining and formatting the data that the simulation provides. The NS simulator monitors all events that occur throughout the simulation and produces a trace file. An event may be classified as an enqueue, dequeue, drop, or receive. The trace file contains one line per event and identifies the event type, time, from-node, to-node, packet type, packet ID, and many other useful values.

While the trace file contains most of the information one needs to know, a significant effort must be given to formatting the results in a useful manner. Many times people generate their own data collection scripts from scratch or borrow and modify someone else's scripts to fit their simulation. As a part of our project, we decided to make a sincere effort at creating a library of generic data extraction tools that are readily available and could be used on any simulation. The scripts we developed are written in C and include *get_thruput_data*, *get_delay_data*, and *get_tcpfriendly_data*.

Before discussing these new tools, we would like to mention that MM-Flow is capable of recording delay files by using the "record-mm-packet-arrival" variable. Similarly, our application layer protocols, MM-App and MPEG-App, are capable of recording scale values using the "record-mm-scale-values" variable. Scale values needed to be printed from our protocols directly; not all protocols have the notion of scale values, so creating a generic script is difficult. The delay values printed by MM-Flow, however, are purely a convenience. These values could have been obtained using the *get_delay_data* script.

3.2.1 get_thruput_data

The *get_thruput_data* script is based upon scripts collected from fellow NS users and is probably the most useful of our scripts. This script collects data about events along a given link in the simulated network. The input values used to run *get_thruput_data* include the following:

- The trace file containing the raw data created by the simulation.
- The "from" node of the link being monitored.
- The "to" node of the link being monitored.
- The maximum outgoing queue size for the link.
- The maximum bandwidth along the link.
- The id of the flows to be monitored (-1 for all).
- The measurement interval in seconds.
- An optional label to be added to the output file names.

The resulting output includes the following:

- A ".utl" file containing percent bandwidth utilization (based on receive events) for each flow and the total utilization.
- A ".drp" file containing the percent of bandwidth dropped for each flow and the total percent bandwidth dropped.
- A ".enq" file containing the percent bandwidth enqueued for each flow and the total percent bandwidth enqueued.
- A ".deq" file containing the percent bandwidth dequeued for each flow and the total percent bandwidth dequeued.
- A ".que" file containing the actual and average queue size for the outgoing queue on the link. Average queue size is a weighted average similar to that used by RED routers.

3.2.2 get_delay_data

The *get_delay_data* script calculates delay for each packet traveling along a given link in the simulated network for a given flow. The input values used to run *get_delay_data* include the following:

- The trace file containing the raw data created by the simulation.
- The name of the output ".dly" file to be created.
- The flow ID to monitor.

- The node where the sender is located.
- The node where the receiver is located.

The resulting output includes the following:

- A file containing the arrival time, packet ID, and delay in seconds.

3.2.3 get_tcpfriendly_data

The *get_tcpfriendly_data* script calculates the TCP-Friendly bandwidth, as determined by the formula presented in section 2.2, as well as the actual bandwidth for a given flow along a given link in the simulated network. The input values used to run *get_tcpfriendly_data* include the following:

- The name of the output ".tfd" file to be created.
- The trace file containing the raw data created by the simulation.
- The delay file containing packet delay values for the flow.
- The flow ID to monitor.
- The "from" node of the link being monitored.
- The "to" node of the link being monitored.
- The measurement interval in seconds.
- The maximum packet size in bytes.
- The maximum bandwidth along the link.

The resulting output includes the following:

- A file containing the TCP-Friendly bandwidth and actual bandwidth used by the flow.

4 Results

This chapter will discuss how the re-engineering of MM-Flow affected both MM-App and MPEG-App, with the main focus on the changes to MM-App. Then, TCP-Friendliness is defined and discussed in regards as to what we took its definition to mean. We then measured how TCP, MM-App-New, and TFRC performed according to the TCP-Friendliness measurements.

4.1 Effects of Re-Engineering MM-Flow

Re-engineering MM-Flow was a big undertaking. Changing the way the protocol layer worked creates significant changes in the application layer. The specific changes that were made, as discussed earlier, had some interesting effects. First we wanted to examine how the performance had changed between the old and new versions of MM-App. Once discovering how our changes to the protocol layer affected MM-App, we took an in depth look as to how different factors regarding the protocol would affect it's performance. Along with looking at how MM-App was affecting by the protocol layer changes, we felt it necessary to examine how the new protocol layer affected MPEG-App's performance.

4.1.1 MM-App-Old vs. MM-App-New

We first wish to compare the original MM-App (from this point forward called MM-App-Old) to the new MM-App (referred to as MM-App-New). To do so, we ran each of these protocols against TCP using our standard bottleneck layout. The bandwidth of the bottleneck link was 2 Mbps and MM-App-Old used the default values of 1.5 Mbps and its highest scale value (scale 4).

Figure 4-1 shows percent utilization of the bandwidth for MM-App-Old and TCP. As indicated, MM-App-Old totally dominated the system by occupying approximately two-thirds of the available bandwidth on average and 75% of the bandwidth at most (which makes sense since 1.5 Mbps is 75% of 2 Mbps). In MM-App-New there is relatively equal sharing of the bandwidth, as shown in Figure 4-2. MM-App-New received about 52% of the bandwidth and TCP received about 45%. This is a tremendous improvement in TCP-Friendliness and fairness over MM-App-Old. One thing that is apparent in Figure 4-2 is that utilization is not very smooth, possibly leading to poor perceptual quality. We will discuss this issue later, but for the time being we will concentrate on smoothness rather than fairness.

Another issue we had hoped to resolve with our re-engineering effort was the coarseness of MM-App-Old's scale values. Since MM-App-Old had only five scale values, the difference in sending rate between scale values is very large. As shown in Figure 4-3, the scale values typically cut in half from 4 to 2, but reach maximum transmission rate again rather quickly, thus never giving TCP a fair chance at obtaining bandwidth. MM-App-New has dynamic scale creation, with a default of 50 scales but the ability to be configured by the Tcl script to allow for as many scales as desired. Figure 4-4 shows how the 50 scales are less coarse and allow for a smooth transition between sending rates. Also, this figure shows how as the scale values get higher, the amount of time before the next increase is longer. This tells us that the delay in the system is getting longer and as a result we receive less ACKS indicating to increase the scale values. The increased number of scale values does not lead to smooth transmission, due to the fact that MM-Flow is constantly probing for bandwidth, but does provide a wider range of transmission rates to explore.

It is also worth pointing out that utilization was not 100% all the time with MM-App-New. While full utilization is optimal, the occasional drop in utilization is a direct result of the queue being allowed to drain, as shown in Figure 4-5. The queue drains when MM-App-New encounters a drop and backs off on its sending rate in order to be fair to competing flows, namely TCP. Thus, we do not consider this to be of major concern.

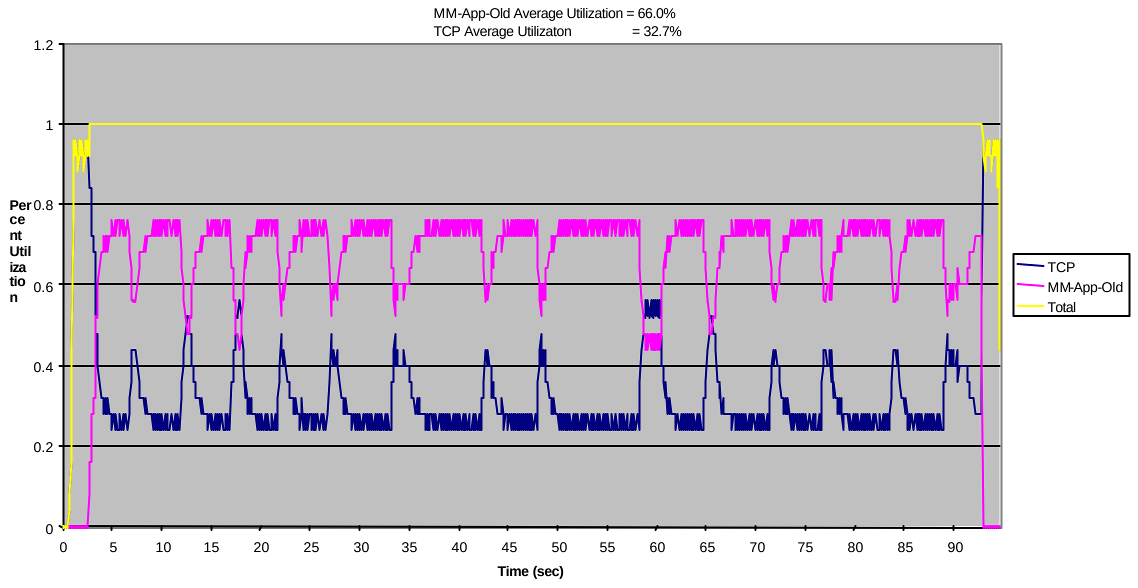

Figure 4-1: Percent Utilization with MM-App-Old vs. TCP

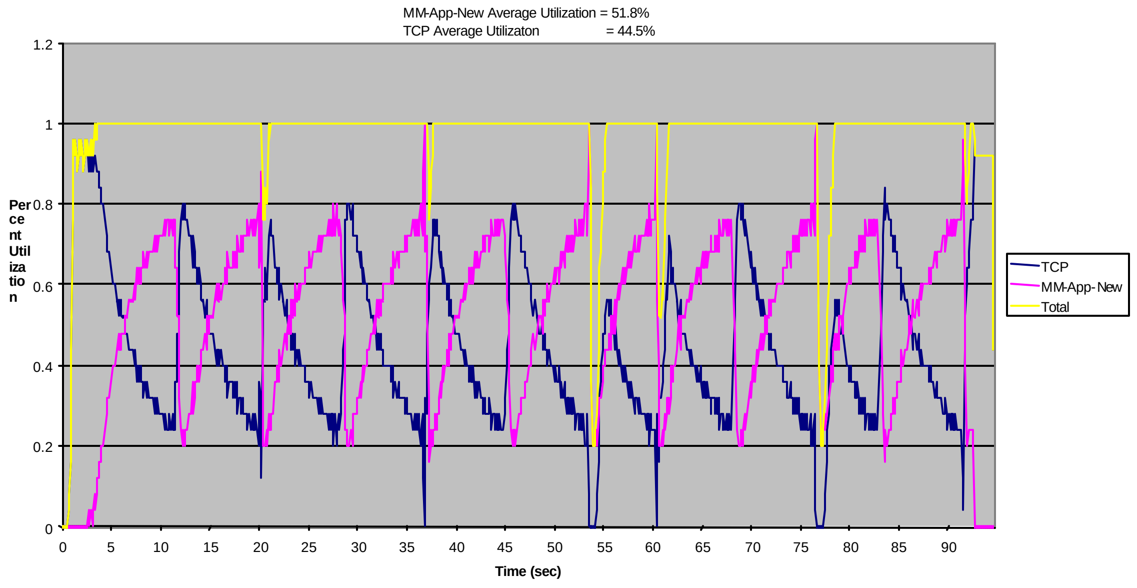

Figure 4-2: Percent Utilization with MM-App-New (Un-weighted) vs. TCP

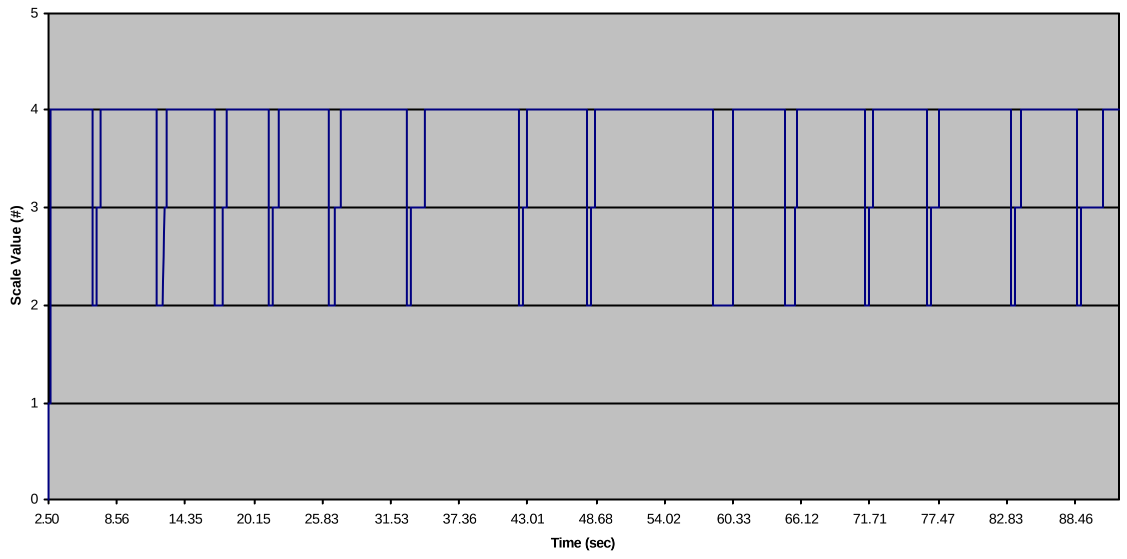

Figure 4-3: Scale Values with MM-App-Old vs. TCP

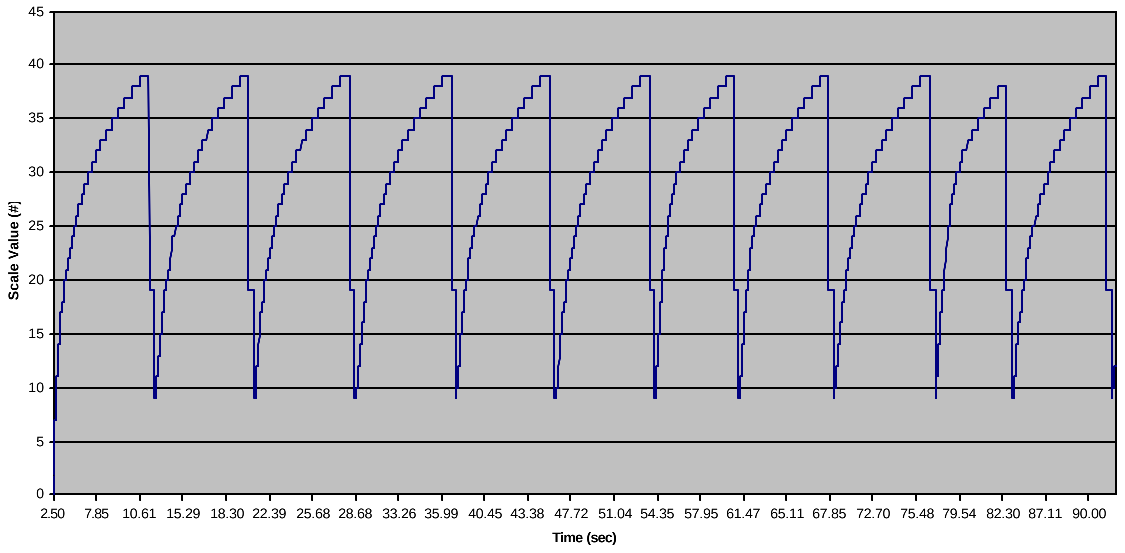

Figure 4-4: Scale Values with MM-App-New (Un-weighted) vs. TCP

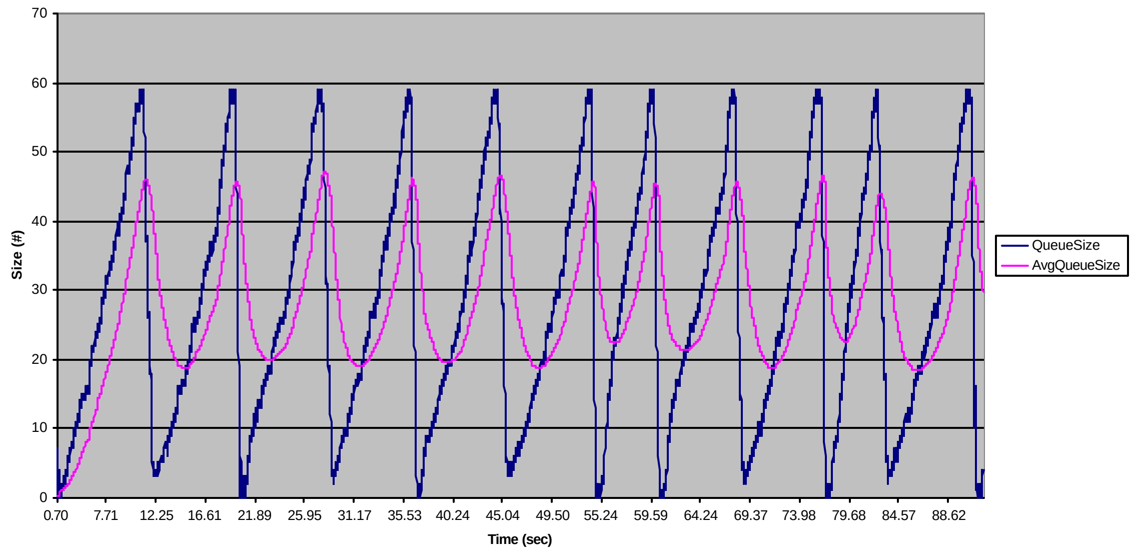

Figure 4-5: Queue Size with MM-App-New (Un-weighted) vs. TCP

4.1.2 Further Evaluation of MM-App-New

In order to gain a better understanding of how MM-App-New performs, we decided to look at a series of simulations with varying conditions. The first thing we wanted to look at was the change in packet and frame size. This is significant since the new protocol decisions are at the transport layer, whereas the old were in the application layer. Second, we wanted to examine how the number of scale values affects the system. Third, we decided to evaluate the responsiveness of MM-Flow when it is affected by different delays in the bottleneck link or when it, or its competing flow, is fragile. Finally, we examined the use of weighted scale values versus un-weighted values.

4.1.2.1 Effect of Packet Size

Unless indicated, both the frame and packet size in our simulations are 1000 bytes, which is typical in the real world [FF1999]. Thus, utilization and scale values for MM-App-New under these conditions is shown in the previous Figures 4-2 and 4-4 respectively.

To examine new conditions, we first decided to triple the size of both the frame and packet to 3000 bytes. This applies to both MM-App-New and TCP. As shown in Figure 4-6, there is less variation in utilization and MM-App-New occupies a much lower percent of the utilization. We concluded that both observations are the result of an increase in RTT, thus leading to a less aggressive MM-App-New. The reason that RTT is longer is that it physically takes longer to send the packets because the queue size is packet based but drain rate is byte based. This increase in RTT can be observed by comparing the time between scale value increases in Figure 4-4 and Figure 4-7.

We also chose to examine what happens when the frame size is larger than the packet size, such that frames need to be broken up into multiple packets. For MM-App-New, we set the frame size to 3000 bytes and the packet size to 1000 bytes. For TCP, the packet size was 1000 bytes as well. Upon observation of Figure 4-8 versus Figure 4-2, we found that there was little difference in the general appearance of the graph and a negligible difference in the average utilizations. Thus, splitting frames apart into multiple packets has little effect on the performance of the system.

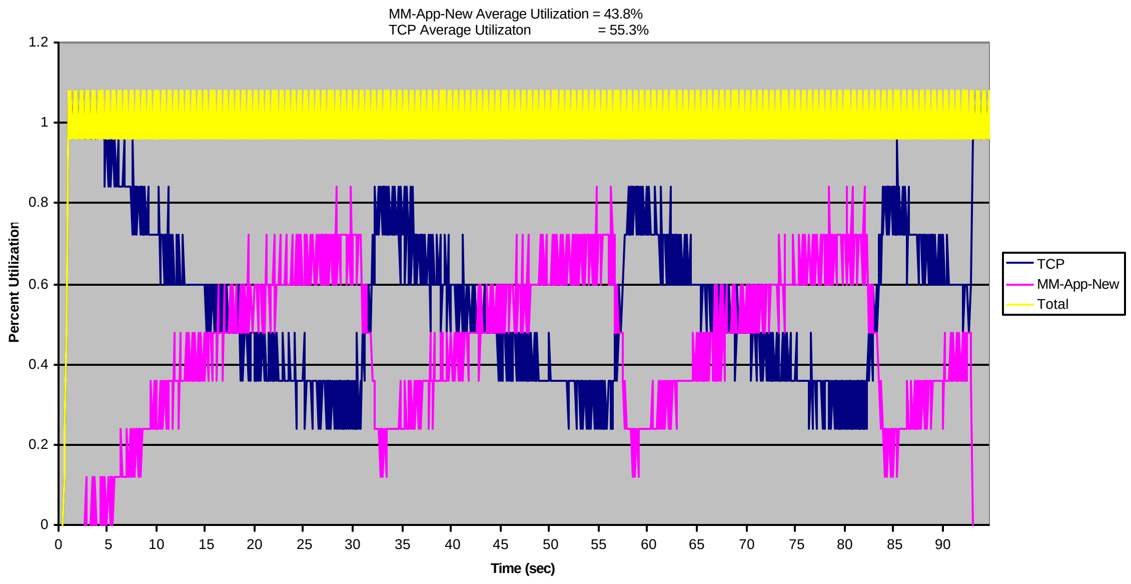

Figure 4-6: Percent Utilization with MM-App-New vs. TCP where Frame Size = 3KB and Packet Size = 3KB

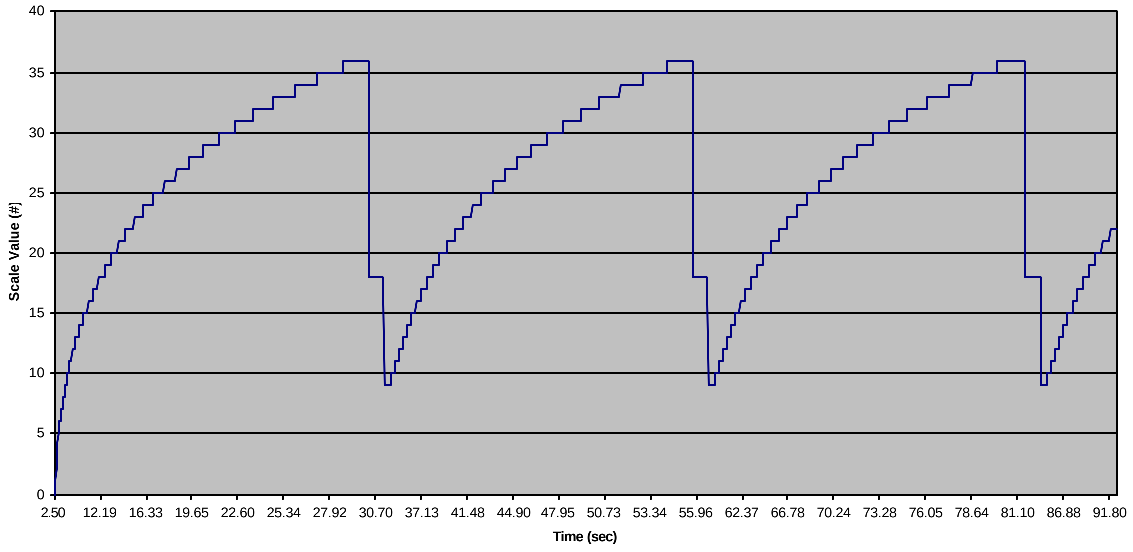

Figure 4-7: Scale Values with MM-App-New vs. TCP where Frame Size = 3KB and Packet Size = 3KB

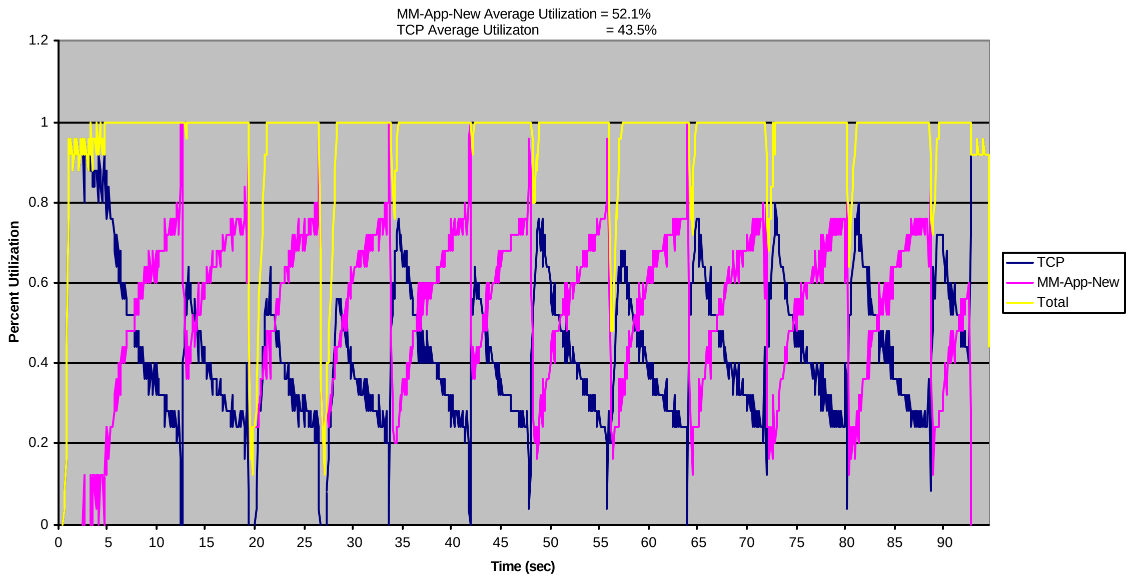

Figure 4-8: Percent Utilization with MM-App-New vs. TCP where Frame Size = 3KB and Packet size = 1KB

4.1.2.2 Effect of Number of Scale Values

Our original assumption was that varying the number of scales would have an effect on performance due to the varying degree of granularity. In order to examine this further, we conducted tests using 25 (Figure 4-9), 50 (Figure 4-10), 150 (Figure 4-11), and 250 (Figure 4-12) scale values for MM-App-New. (Note that 50 scales is the default value used in most of our scripts and that Figure 4-10 is identical to Figure 4-2 presented earlier.)

As shown in the figures, moving from 25 towards 250 scale values results in less variation in utilization. This behavior has to do with the fact that we are increasing by one scale value per ACK and we ACK/NACK once per RTT, so it simply takes a longer time to move across the entire range of scales. For example, with 25 scale values it would take 25 RTTs (assuming no NACKs) to reach 100% utilization. With 50 scale values it would take 50 RTTs, and so on. Figure 4-14 visually illustrates this point.

We also observed that as we increased the number of scale values the average utilization for MM-App-New decreased and the average utilization for TCP increased. This behavior is illustrated in Figure 4-14. As shown, the lines intersect, meaning MM-App-New and TCP each receive 50% utilization on average, when there were 150 scales. This suggests that a possible optimal configuration would have 150 scales, rather than the 50 scales that we used. We will revisit this point in the "Future Work" section of this paper.

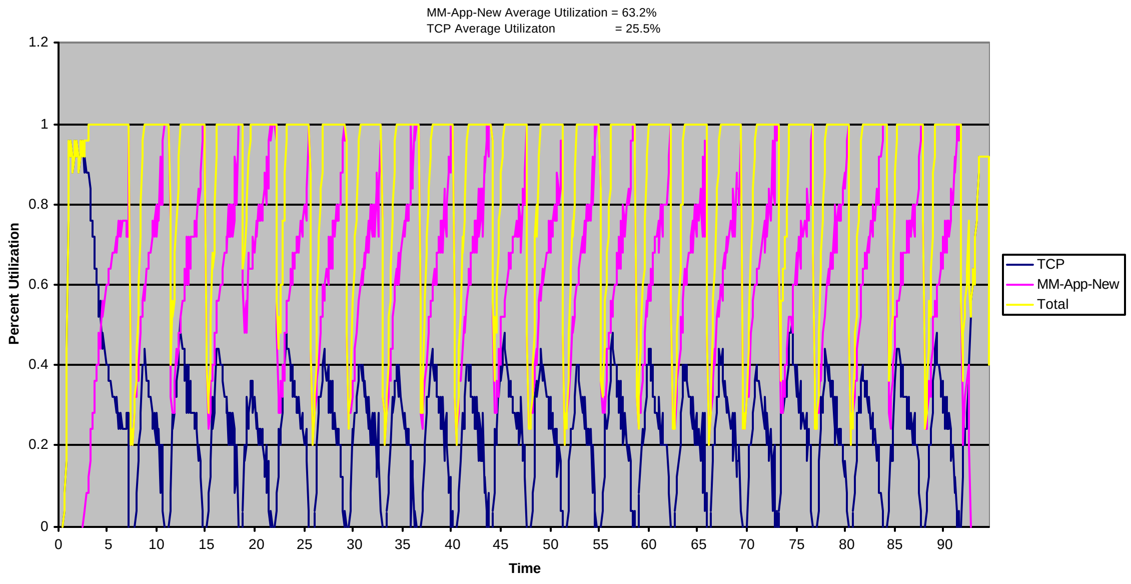

Figure 4-9: Percent Utilization with MM-App-New vs. TCP and 25 Scale Values

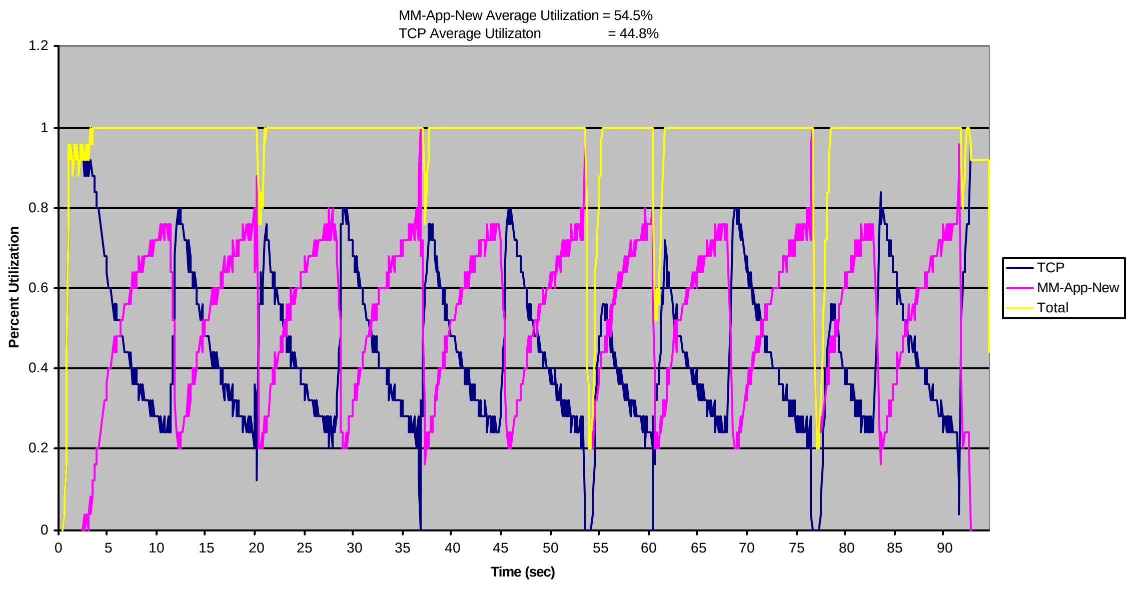

Figure 4-10: Percent Utilization with MM-App-New vs. TCP with 50 Scale Values

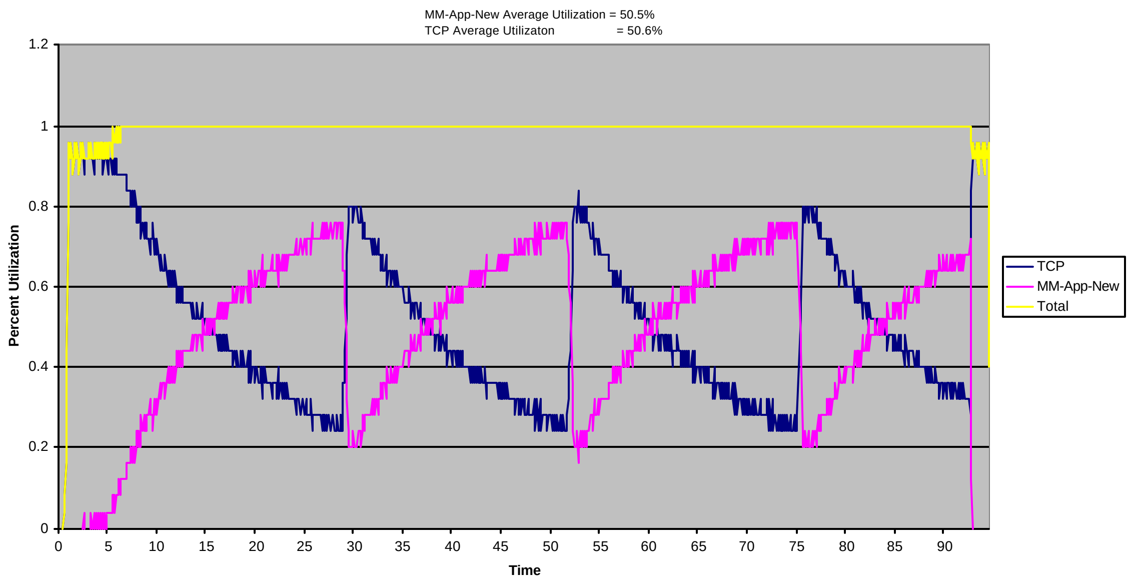

Figure 4-11: Percent Utilization with MM-App-New vs. TCP with 150 Scale Values

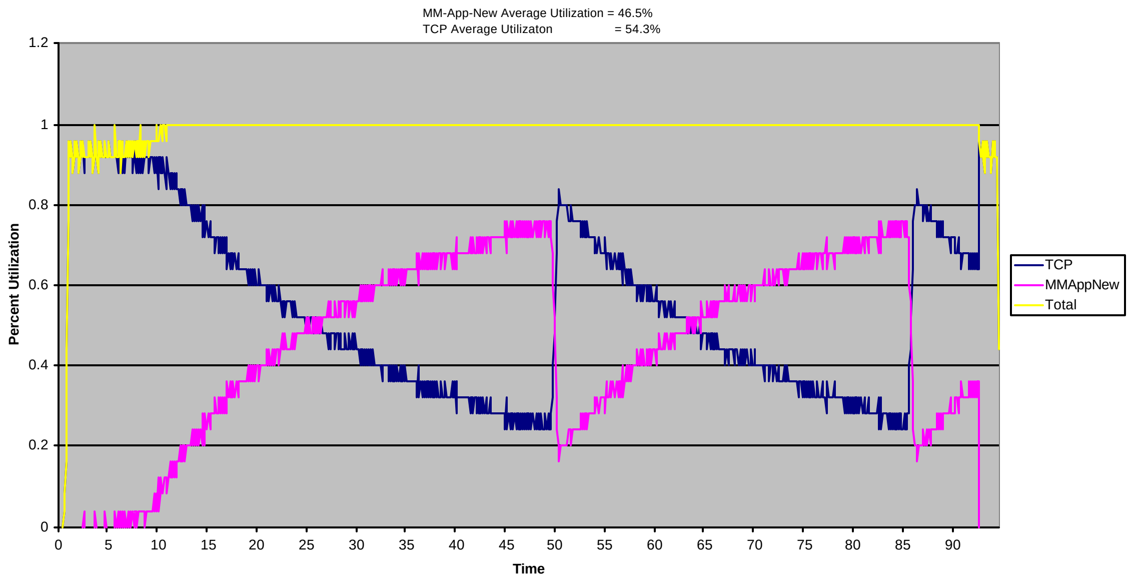

Figure 4-12: Percent Utilization with MM-App-New vs. TCP with 250 Scale Values

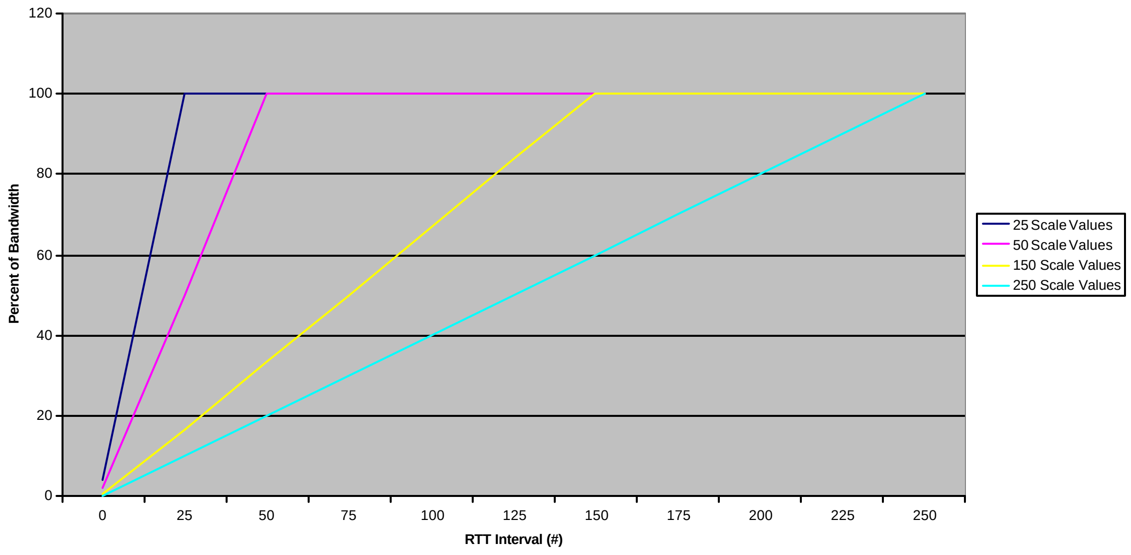

Figure 4-13: Aggressiveness of Reaching Maximum Bandwidth Using Different Numbers of Scale Values

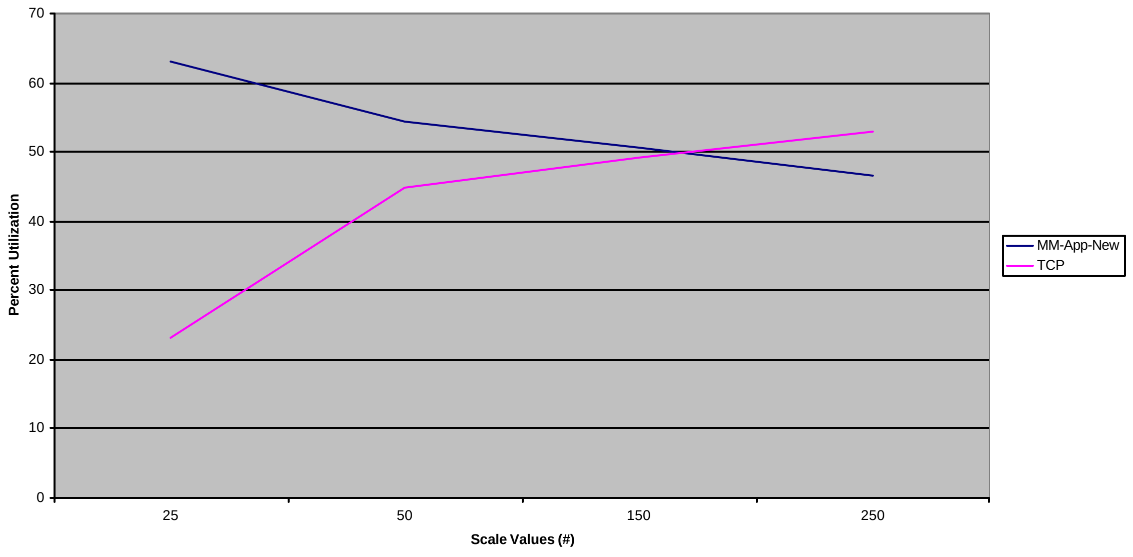

Figure 4-14: Average Percent Utilization vs. Number of MM-App-New Scale Values Used

4.1.2.3 Effect of Delay

Due to the fact that MM-App-New is a rate-based protocol, we had reason to believe that MM-App-New would take up more bandwidth when there is longer delay. This behavior is typical for situations in which rate-based protocols like MM-Flow are up against window-based protocols like TCP. A window-based protocol sends a series of packets and waits until it receives an ACK for them before sending again. As the delay increases, it has to wait longer for the ACK and effectively loses some of the bandwidth it could have used if it had received an ACK faster. A rate-based protocol, however, keeps sending data at a given rate regardless of how long it takes to receive an ACK. Thus, a rate-based protocol is able to steal some of the bandwidth not used by an idle window-based protocol.

As shown in Figure 4-15, the overall behavior of the system with longer delay is roughly the same as our original results in Figure 4-2, but MM-App-New does in fact consume more bandwidth, 55%, while TCP is only able to grab 33%. It is also important to notice that overall utilization in poorer in the system with longer delay. There are many more instances in which utilization falls below 100%, which can be explained by examining the queue. By comparing Figure 4-16 with Figure 4-4, one will notice that the queue size reaches zero more often when there is a longer delay. This is caused by the fact that it takes longer for ACK/NACK packets to be sent back to the sender. MM-App-New and TCP are, therefore, slow to respond to the congestion. By the time they respond, the congestion may be over and the queue is allowed to drain, which causes total utilization to fall.

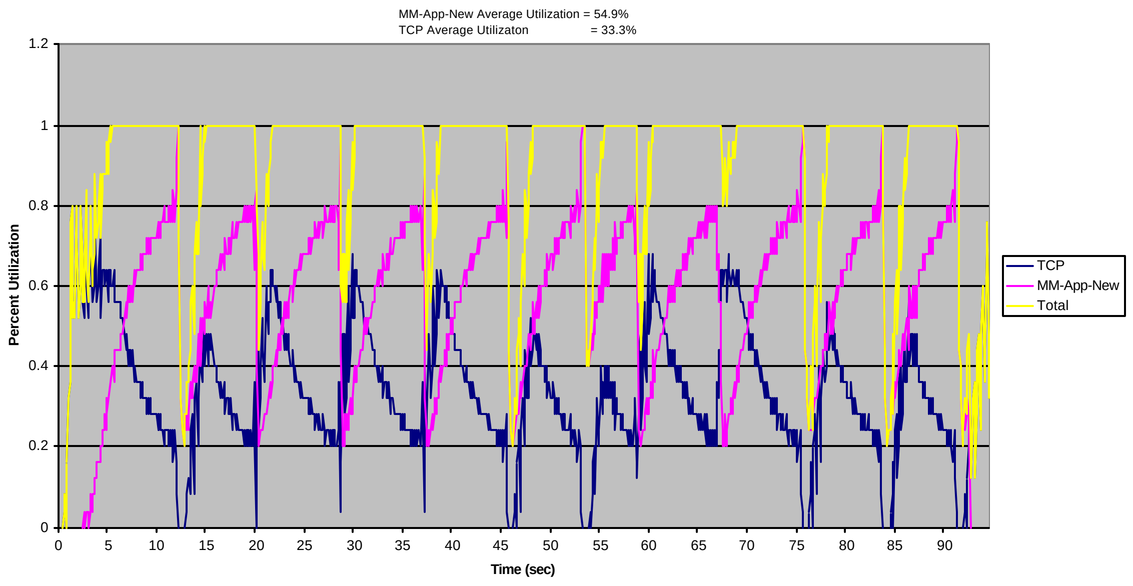

Figure 4-15: Percent Utilization with MM-App-New vs. TCP with Longer Delay (40ms)

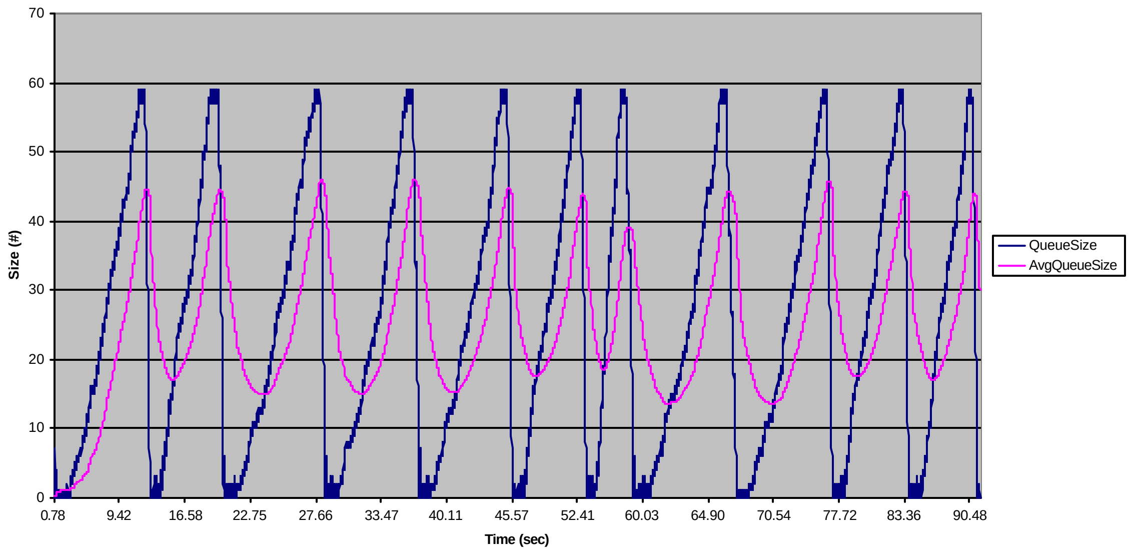

Figure 4-16: Queue Size with MM-App-New vs. TCP with Longer Delay (40ms)

4.1.2.4 Effect of Fragile Flows

Now that we've seen how MM-App-New fairs against TCP when they both have a long delay over the bottleneck link, it is important to see how they will act when each is fragile. The fragile flows in the following simulation are five times further away from the router than the competing flow, meaning 100ms rather than 20ms.

For the first simulation TCP is the fragile flow. As shown in Figure 4-17, MM-App-New dominates the system while TCP averages only 10.5% utilization. While this situation might be undesirable for the TCP user, it is typical of systems involving fragile flows. In the second simulation we examined what happens when MM-App-New is fragile. The results, shown in Figure 4-18, indicate that MM-App-New and TCP share the overall bandwidth relatively fairly. We believe this occurs due to the fact that in all of our simulations TCP is really just filling in the gaps left behind by MM-App-New. Thus, when MM-App-New is fragile he is still able to claim half of the bandwidth. Luckily in most situations MM-App-New is responsive enough to allow TCP to gain a fair share as well.

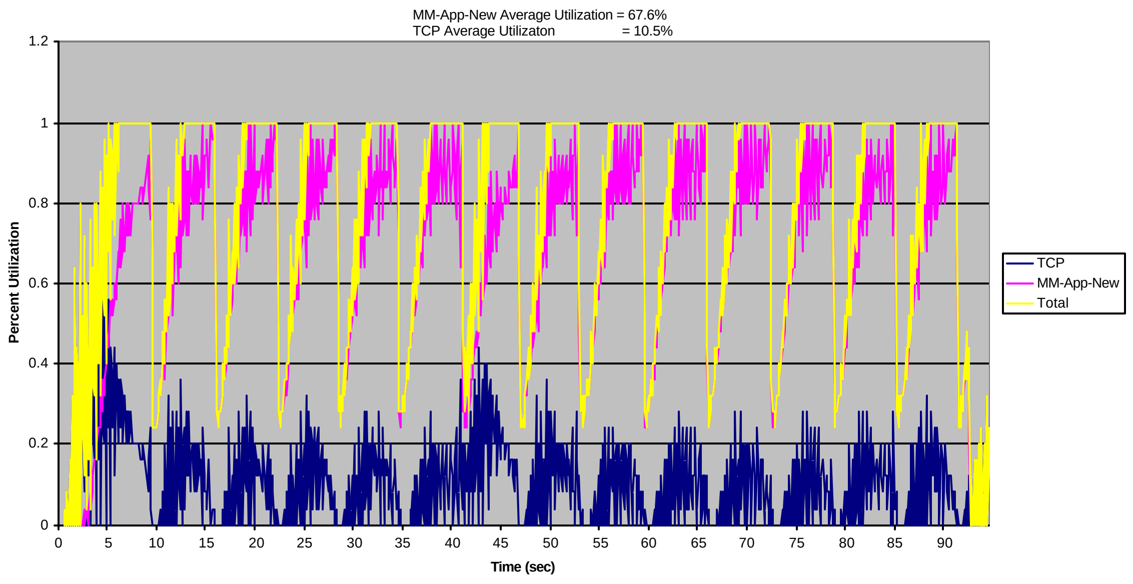

Figure 4-17: Percent Utilization with MM-App-New vs. TCP when TCP is Fragile

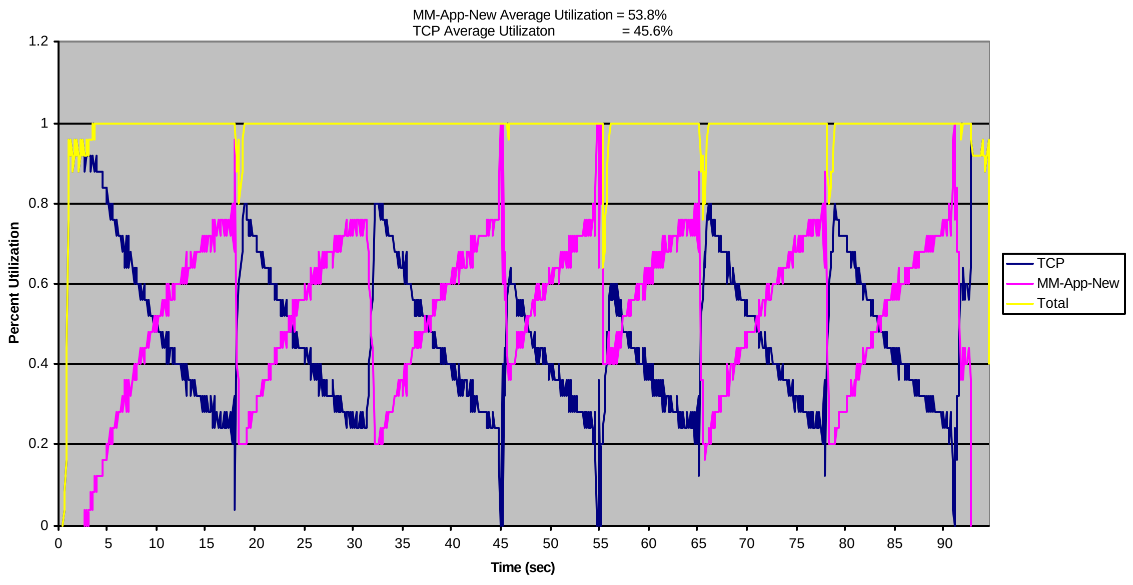

Figure 4-18: Percent Utilization with MM-App-New vs. TCP when MM-App-New is Fragile

4.1.2.5 Effect of Weighted Scale Values

While MM-App-New has given us some improvements over MM-App-Old, it is apparent that the protocol does not quite provide smooth transmission rates. We have already shown that increasing the number of scale values leads to less variation, but it is also common for protocols to use weighted averages in order to smooth sending rates. This includes MM-Flow, and therefore we must examine what effect weighted scale values have upon the performance of MM-App-New.

Figure 4-19 shows the results of when we ran MM-App-New with weighted scale values against TCP. The figure shows that there was approximately 10 cycles in utilization, whereas our original data (shown in Figure 4-2) contained 11 cycles. Therefore, applying weighted scale values did not produce dramatic differences but still reduced the amount of variation in the system, as expected. It is also important to point out, however, that MM-App-New takes a little more bandwidth than it used to. This was expected as well, since weighted scale values mean the protocol is less responsive to congestion.

Since weighted values produced subtle results, we will continue using un-weighted values in each simulation throughout this paper. We will also revisit scale values, weighting schemes, and efforts at achieving smoother data in the “Future Work” section of this paper.

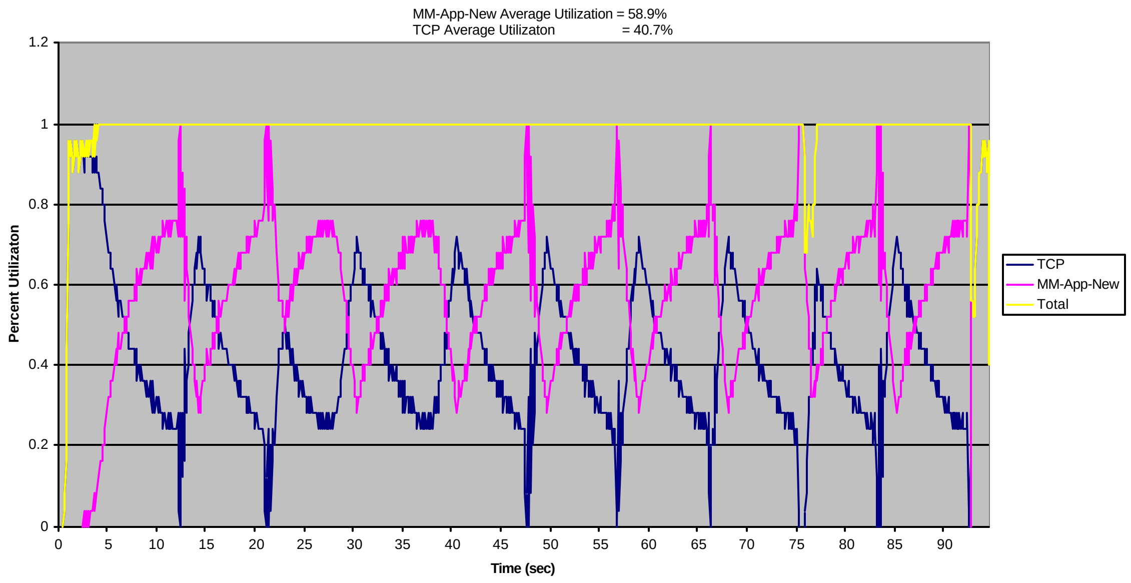

Figure 4-19: Percent Utilization with MM-App-New vs. TCP with Weighted Scale Values

4.1.3 MPEG-App-Old vs. MPEG-App-New

When MM-App-Old was constructed, another application was created called MPEG-App. MPEG-App works by breaking up and sending specific parts of the MPEG media file. Each mpeg file can be broken down into three different types of frames, all of different sizes and known as I, B, and P frames. MPEG-App has five scale values. Each scale value corresponds to sending various combinations of I, B, and P frames such that the relationship between transmission rate and scale value increases linearly.

Since we have shown that more scale values typically leads to better performance, it would have been desirable to create more scale values for MPEG-App. This would have taken a considerable amount of time relative to the length of our project, so we decided to focus our project on just the MM-Flow transport layer and manipulating MM-App. Therefore, the only difference between MPEG-App-Old and MPEG-App-New is that the new version is designed to run on top of MM-Flow, just like MM-App-New.

Though we didn't change the application layer functionality, we felt it was still necessary to examine how MPEG-App-Old performs and compare it to MPEG-App-New now that it runs on top of MM-Flow. We only performed a simulation for the standard bottleneck layout with MPEG-App running against TCP.

Figure 4-20 shows the percent utilization for MPEG-App-Old vs. TCP. As shown, MPEG-App-Old uses about 58% of the bandwidth and TCP, for some reason, is restricted to only 32%. Also, utilization is very erratic which will result in poor quality. Figure 4-21 shows the results of MPEG-App-New versus TCP and shows that the new version totally overpowers TCP. This behavior is expected since five scale values is too coarse a granularity to be effective, so the protocol aggressively seeks bandwidth and has a hard time reacting to congestion.

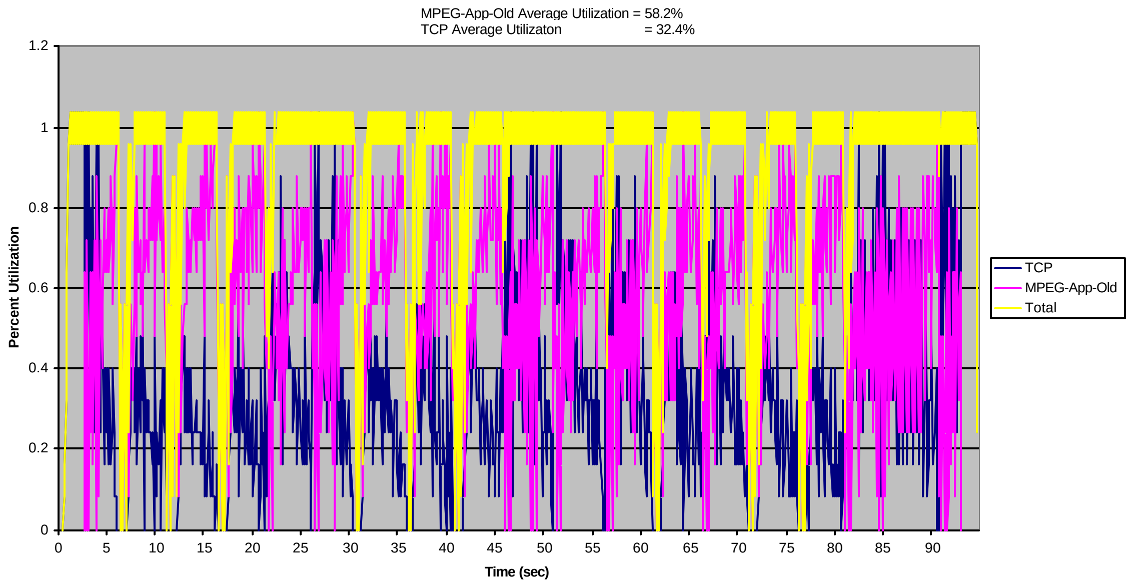

Figure 4-20: Percent Utilization with MPEG-App-Old vs. TCP

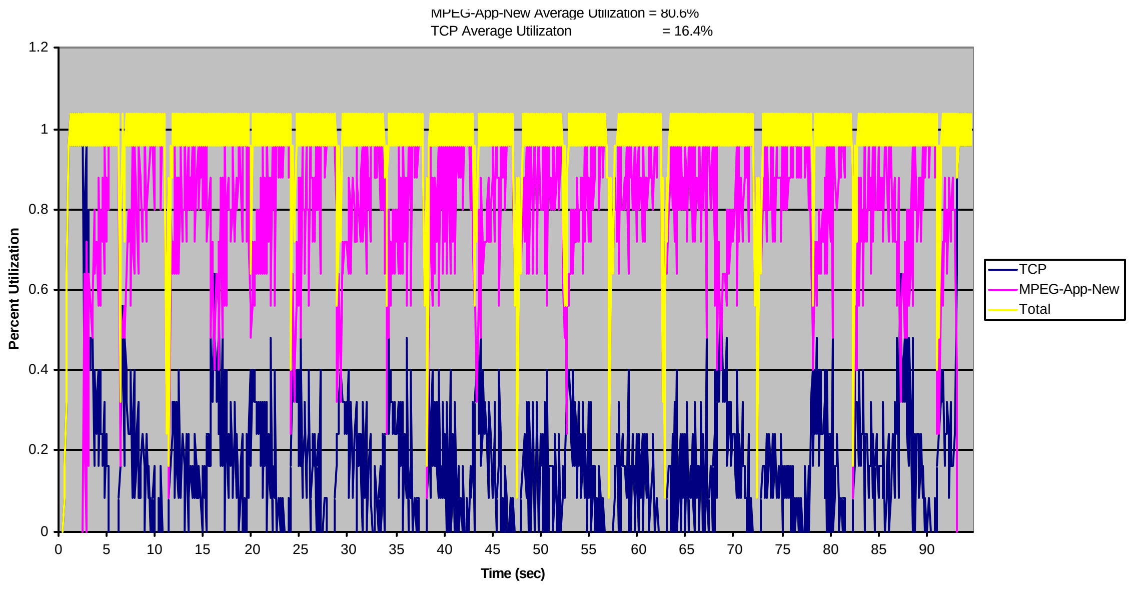

Figure 4-21: Percent Utilization with MPEG-App-New vs. TCP

4.2 *TCP-Friendliness*

While looking at graphs of utilization, scale value, and queue size allows us to understand the overall behavior of the system, it is also possible to obtain specific measurements concerning the TCP-friendliness of a flow. As stated previously, a TCP-friendly flow is considered to be one that transmits at a rate less than or equal to a TCP flow under the same conditions. The TCP-friendly bandwidth can be calculated by using Equation 2-2 (shown again below for reference).

$$T \le \frac{1.5\sqrt{2/3} * B}{R * \sqrt{p}}$$

Equation 2-1: TCP-Friendly Transmission Rate

Our data collection script *get_tcpfriendly_data* implements this formula and allows us to compare actual bandwidth used by a flow against its calculated TCP-friendly bandwidth. Before we examine the results obtained from this script, we need to gain an understanding of its implications and the circumstances in which this formula applies.

First, we'll go through a brief dimensional analysis. As the formula indicates, the TCP-friendly bandwidth T is dependent upon the packet size B, the round trip time R, and the packet drop rate p. As the transmitted packet size increases, so does the TCP-friendly bandwidth. Conversely, as the round trip time or packet drop rate increases the TCP-friendly bandwidth decreases.

It is also important to recognize the implications of the variables in this equation. Packet size and round trip time are almost always strictly positive, so they usually do not have any severe implications. Problems begin to arise when the packet drop rate is zero, which happens frequently in simple simulations. Since it is in the denominator, a packet drop of rate of zero causes the TCP-friendly bandwidth to approach infinity. When creating our *get_tcpfriendly_data* script we wanted to produce data that was accurate, but easy to graph, so we made the assumption that a packet drop rate of zero (or a round trip time of zero, just to be safe) will cause the value for TCP-friendly bandwidth to equal the maximum bandwidth of the link.

One way to solve the problem of a packet drop rate of zero and obtain more informative TCP-friendly data is to increase the measurement interval size. Figures 4-22, 4-23, and 4-24 show a TCP-Friendly graph with measurement intervals of 1 second, 3 seconds, and 5 seconds respectively. These figures illustrate that a small interval size will end up having few drops per interval, and therefore lots of places where the TCP-friendly bandwidth equals the maximum bandwidth of 4.0 Mbps. As the interval size increases the TCP-friendly bandwidth begins to take on useful values. The drawback of this process is that increasing measurement intervals also smoothes the actual bandwidth, possibly to the point that important data points are missing and its usefulness is lost. Therefore, the interval size one chooses to use will depend on the simulation and the degree of granularity desired.

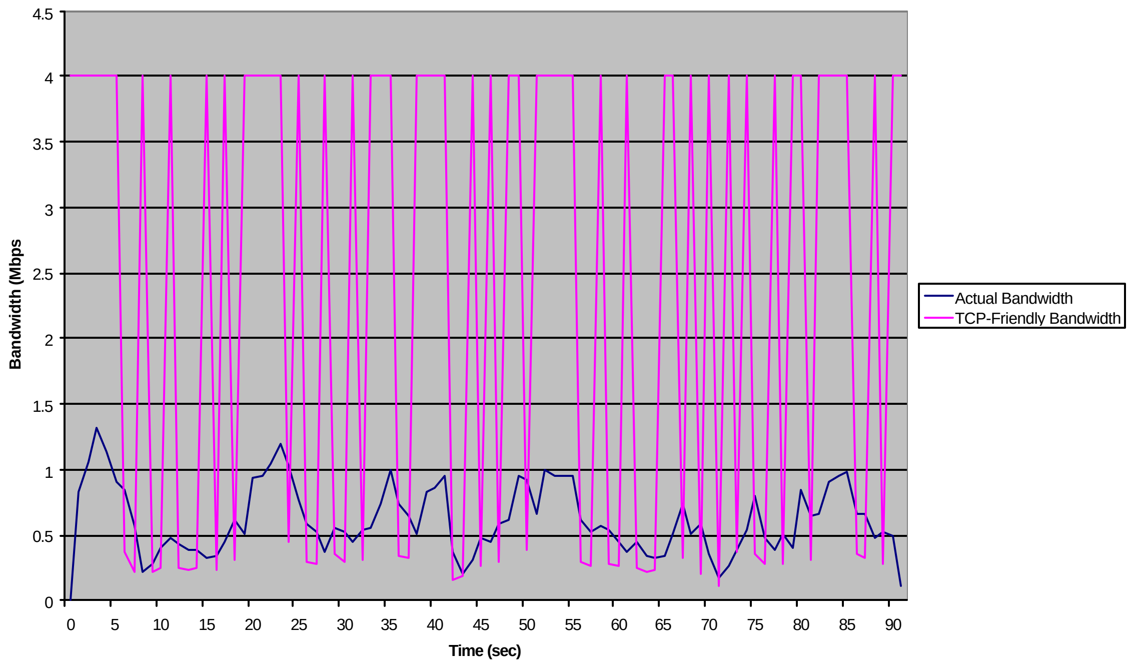

Figure 4-22: TCP-Friendly and Actual Bandwidth Measurements with a 1 Second Interval

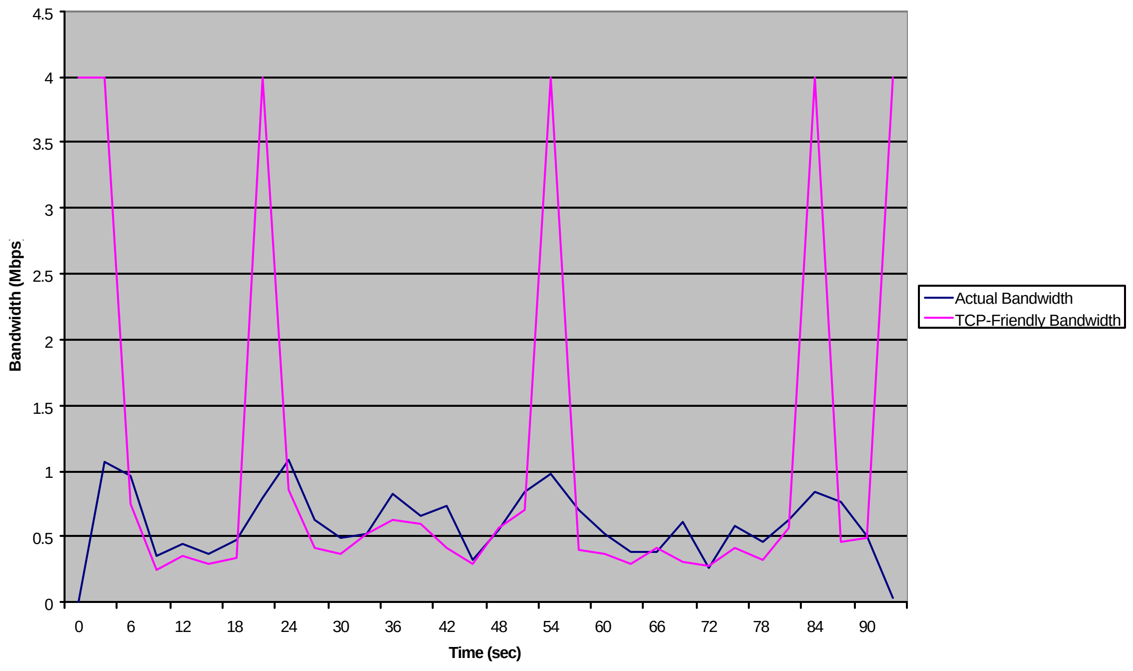

Figure 4-23: TCP-Friendly and Actual Bandwidth Measurements with a 3 Second Interval

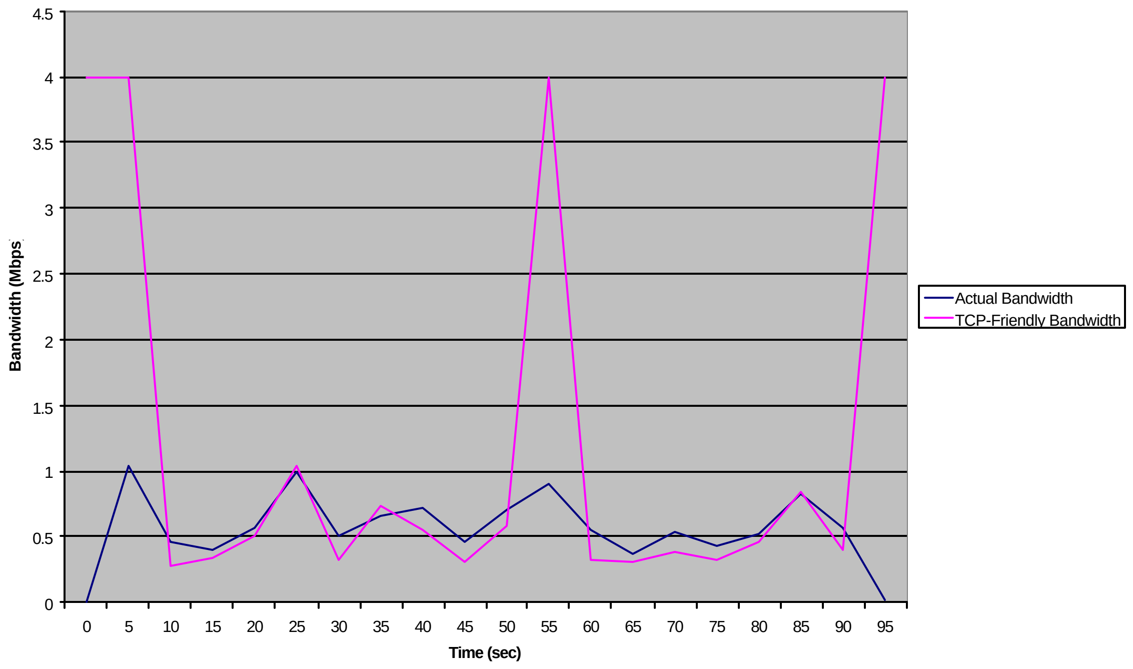

Figure 4-24: TCP-Friendly and Actual Bandwidth Measurements with a 5 Second Interval

4.3 TCP-Friendliness and Performance of MM-Flow vs. Other Protocols

Now that we have a better understanding of TCP-Friendliness and how it can be measured, we are interested to see how the different simulations hold up in regards to the TCP-Friendly equation. First, we decided to take a look at a TCP vs. TCP simulation so that we may see how two TCP flows re-act against each other. We will then take a look at MM-App-New versus TCP, TFRC vs. TCP, and MM-App-New vs. TCP. Finally, we will wrap up our evaluation with a multiple-protocol environment placing 4 TCP, 2 MM-App-New, and 2 TFRC in the system at the same time.

For all protocol environments except the multi-protocol we performed simulations using the standard bottleneck, standard fragile, and standard delay layouts. From these three tests, we want to examine how each competing flow responds. For each simulation we have provided a figure displaying the actual bandwidth, along with an overlay of the calculated TCP-friendly bandwidth. In most of our simulations we had a difficult time getting accurate TCP-friendly data due to extremely low drop rates. Therefore, we have added the concept of a “Fair Bandwidth” to aid our discussion. Fair bandwidth is calculated by dividing the total bandwidth of the bottleneck link by the total number of flows in the simulation.

4.3.1 TCP vs. TCP

Since we are examining the notion of TCP-friendliness, we felt it was necessary to see how two TCP flows reacted among themselves. This simulation is useful since we have yet to look at two TCP flows competing. Also, it may be interesting to see if even TCP is TCP-friendly according to the current definition.

4.3.1.1 Basic Simulation

This simulation was run with two TCP flows running against each other. With two identical TCP flows we felt it was only necessary to graph the data for one TCP flow and it’s TCP-Friendliness values. Figure 4-25 shows the results of this basic test. As can be seen from the figure, the value of TCP-Friendliness is constantly at the maximum bandwidth of 2 Mbps. This is due to the fact that there are no drops occurring within the simulation. Since TCP is always under the TCP-friendly line we must conclude that under standard bottleneck conditions it is TCP-friendly. Similarly, while TCP’s actual bandwidth may fluctuate it is relatively close to the Fair Bandwidth value, so we also conclude that it is fair.

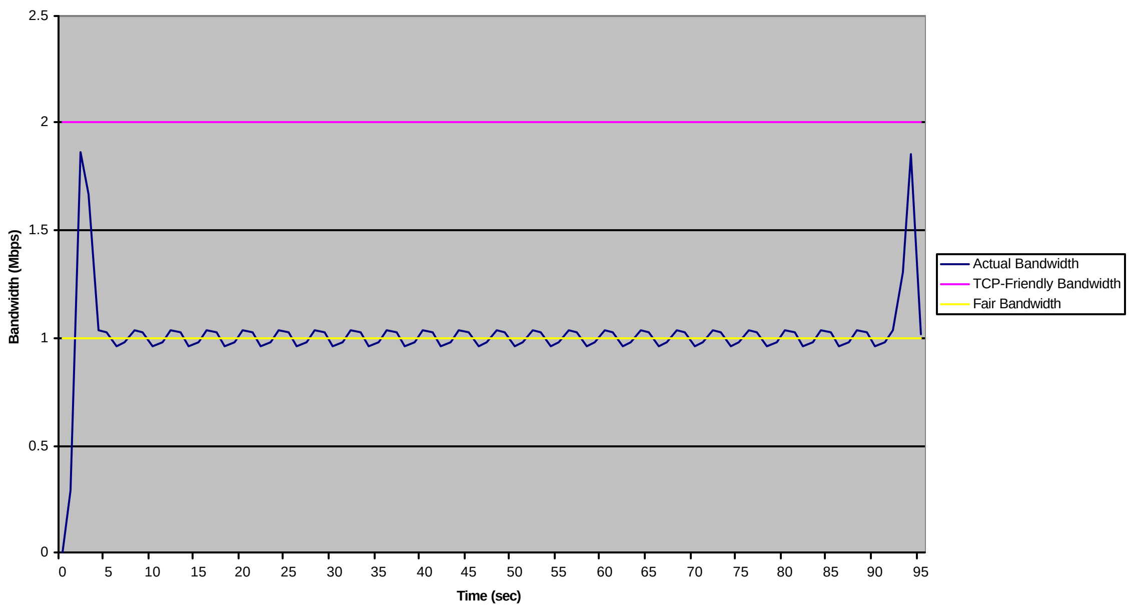

Figure 4-25: TCP Friendly & Fair Bandwidth Overlay for TCP in TCP vs. TCP

4.3.1.2 Effect of Fragile Flows

Even though two TCP flows running equal to each other may seem fine, we want to know how TCP reacts when one flow is far away from the destination while the other is close. We use the standard fragile layout for this simulation. Figure 4-26 shows the TCP-friendly and Fair overlays for the fragile TCP flow, while Figure 4-27 shows the data for the TCP flow that is close to the router. As shown, both flows fall under the TCP-friendly line so they are considered to be TCP-friendly. Fairness is another question entirely. The fragile flow is well below the Fair line, while the non-fragile flow is well above it. Thus, we must conclude that in a simulation with one fragile flow, the fragile will not receive its fair share. As in previous simulations, this outcome is expected since the fragile flow is at a disadvantage.

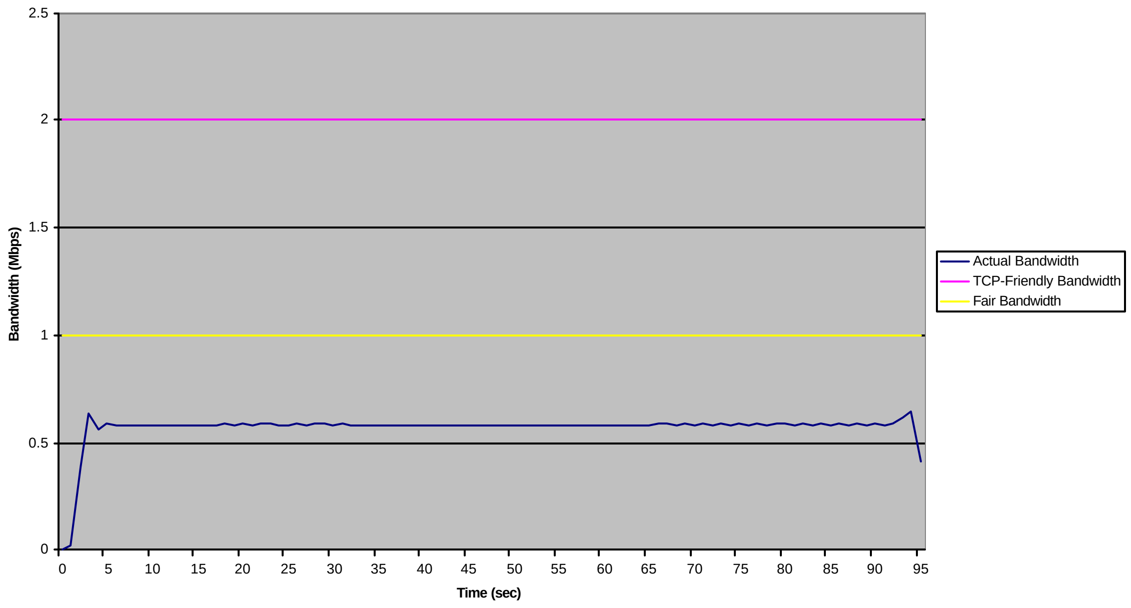

Figure 4-26: TCP Friendly & Fair Bandwidth Overlay for TCP1 in TCP vs. TCP where TCP1 is Fragile

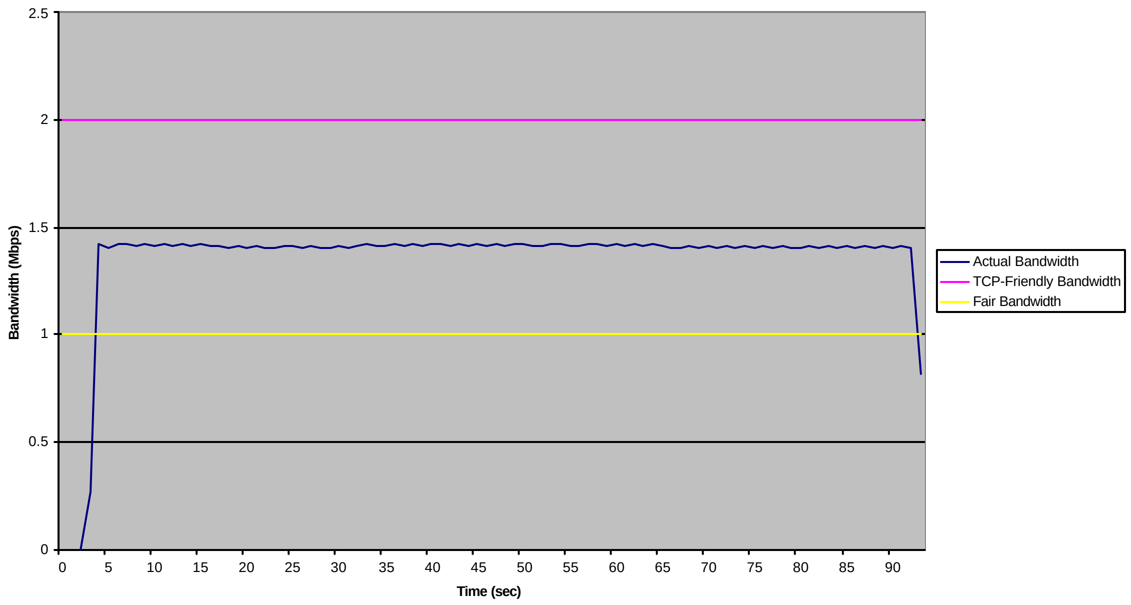

Figure 4-27: TCP-Friendly & Fair Bandwidth Overlay for TCP2 in TCP vs. TCP where TCP1 is Fragile

4.3.1.3 Effect of Delay

Measuring how the TCP flows re-act to having a long bottleneck delay is key, because in the real world, chances are this is likely to occur. Figure 4-28 shows us that the two TCP's are acting completely fair with one another. Each flow is using exactly half of the total bandwidth for the simulation. This is expected, because since both flows are experiencing the long delay, it will act similar to that in the basic simulation that we looked at above. Again we must conclude that both flows are TCP-friendly and fair.

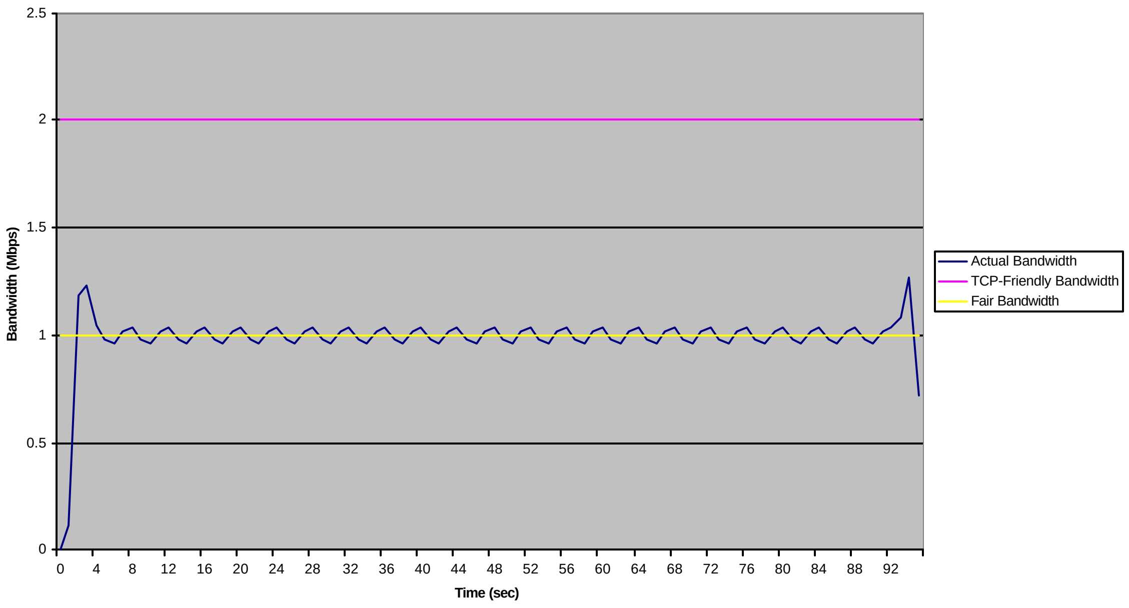

Figure 4-28: TCP Friendly & Fair Bandwidth Overlay for TCP in TCP vs. TCP with Longer Delay

4.3.2 MM-App-New vs. TCP

We have already examined utilization, scale values for MM-App-New versus TCP, but now we must take a look at TCP-Friendliness and Fairness in order to be able to compare it to the other protocols.

4.3.2.1 Basic Simulation

This simulation shows MM-App-New and TCP in a standard bottleneck layout. The measurements for TCP-Friendliness and Fair Bandwidth for TCP and MM-App-New are shown in Figures 4-29 and 4-30 respectively. In terms of fairness, these graphs show just about what we have already seen from the utilization. That is at times the TCP or

MM-App-New are above the fairness level, while other times they are below, but on average they are relatively fair with perhaps MM-App-New having a slight advantage over TCP. In terms of TCP-Friendly measurements, we are beginning to see an increase in the drop rate, but we would still need an interval of about ten seconds to start seeing this data form something concrete. A ten second interval was larger than we were willing to accept due to the fact that it is a long time in the world of networks. Using the results obtained from the formula, says that both are TCP-Friendly, except for when MM-App-New reaches its threshold at which it then starts dropping packets.

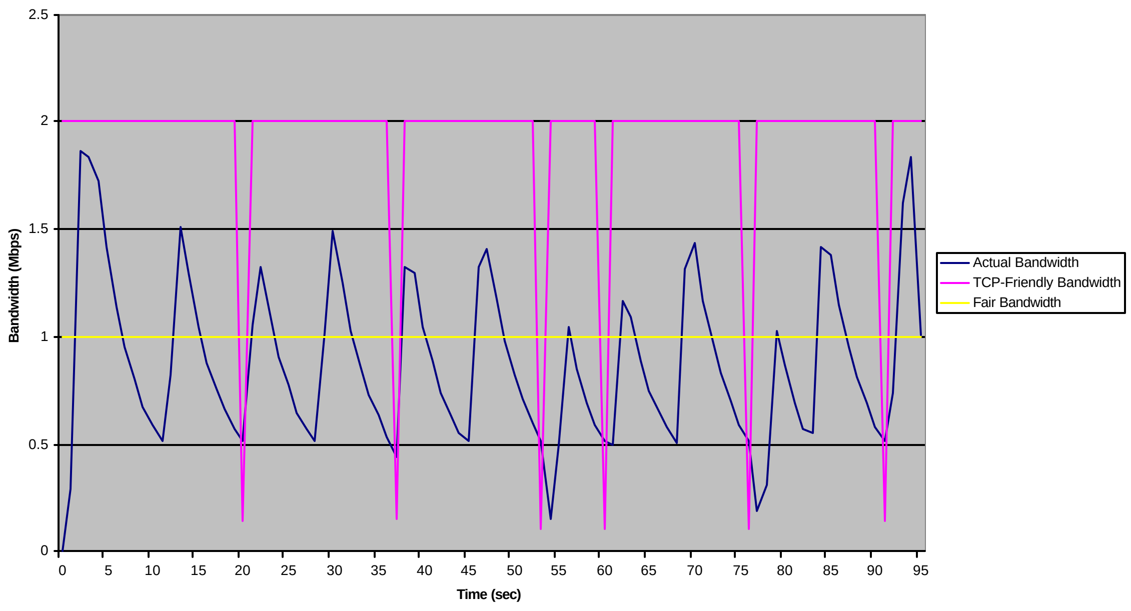

Figure 4-29: TCP Friendly & Fair Bandwidth Overlay for TCP in TCP vs. MM-App-New

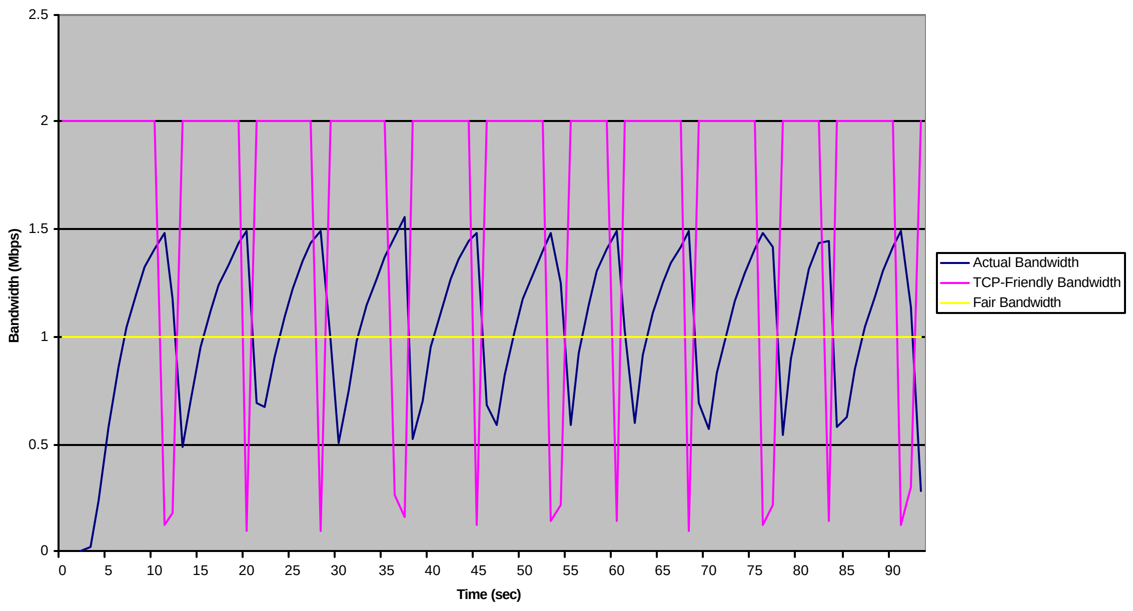

Figure 4-30: TCP Friendly & Fair Bandwidth Overlay for MM-App-New in TCP vs. MM-App-New

4.3.2.2 Effect of Fragile Flows

The results of this set of simulations, shown in Figures 4-31 through 4-34, also correspond with the results previously discussed. When TCP is the fragile flow it receives less bandwidth than MM-App-New, and when MM-App-New is the fragile flow they tend to share the bandwidth relatively equal. According to the TCP-Friendly calculations, when TCP is fragile the drop rate is higher, but still warrants a 5-8 second interval for gaining any concrete data. When MM-App-New is fragile the drop rate is less warranting an interval is too large for us to examine without losing valuable data points.

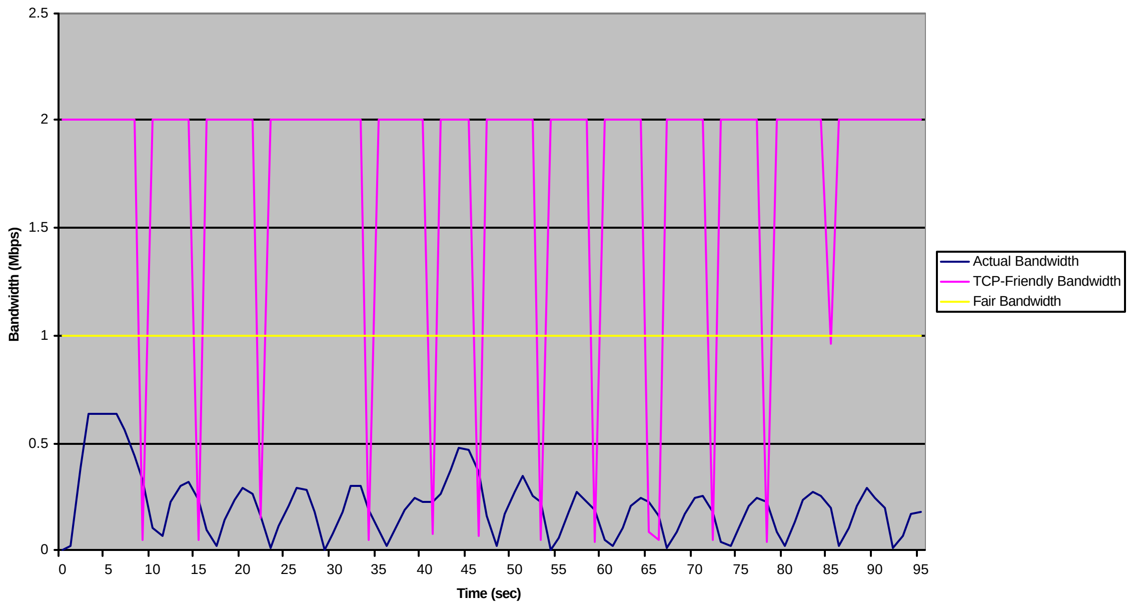

Figure 4-31: TCP Friendly & Fair Bandwidth Overlay for TCP in MM-App-New Vs. TCP where TCP is Fragile

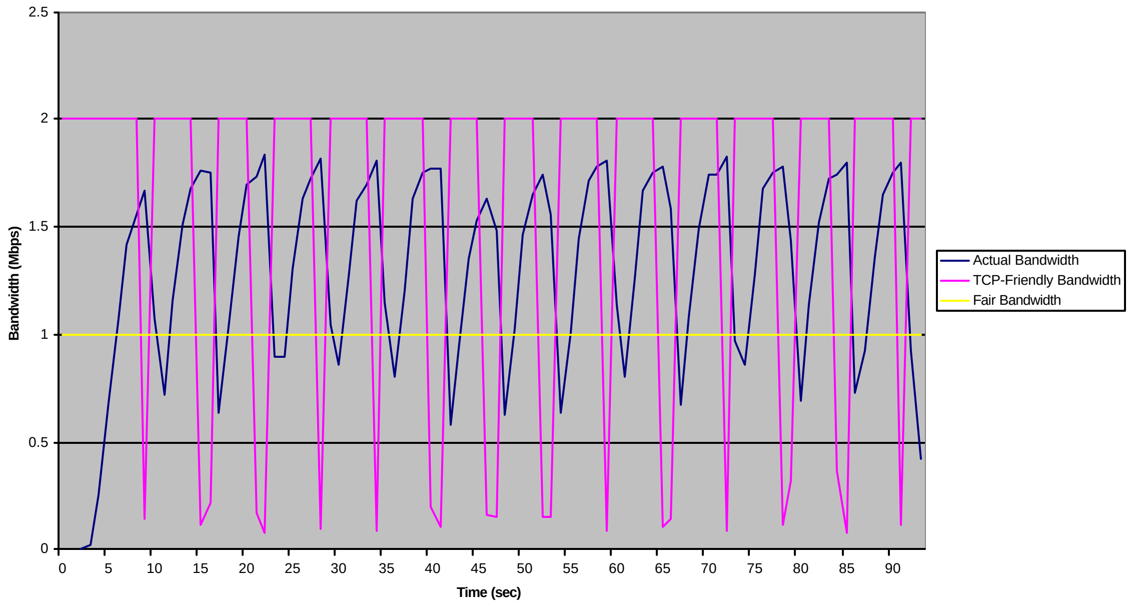

Figure 4-32: TCP Friendly & Fair Bandwidth Overlay for MM-App-New in MM-App-New vs. TCP where TCP is Fragile

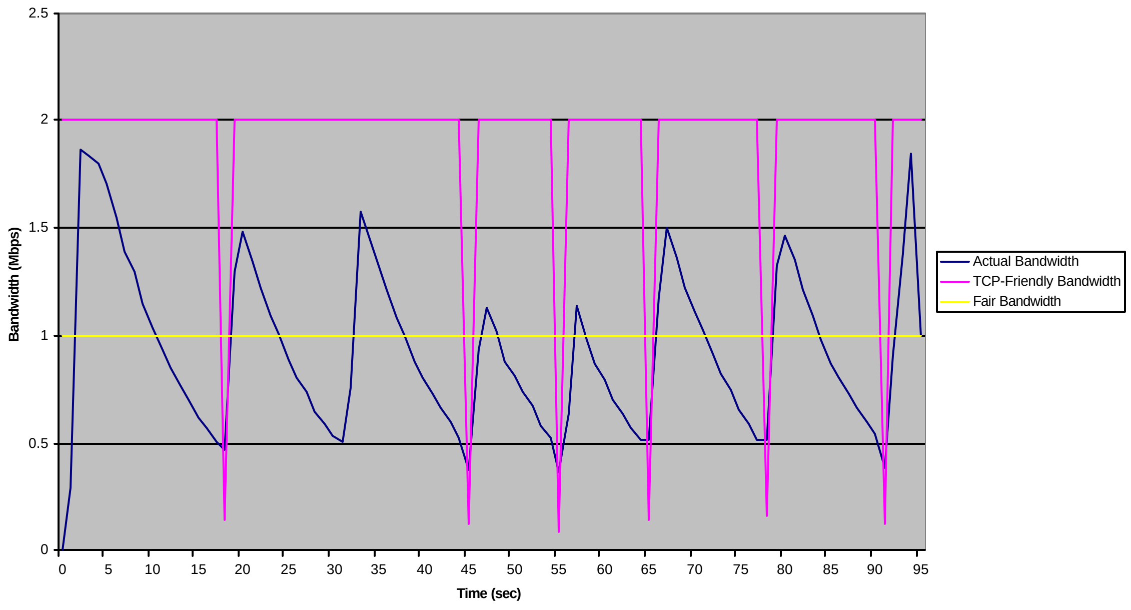

Figure 4-33: TCP Friendly & Fair Bandwidth Overlay for TCP in MM-App-New Vs. TCP where MM-App-New is Fragile

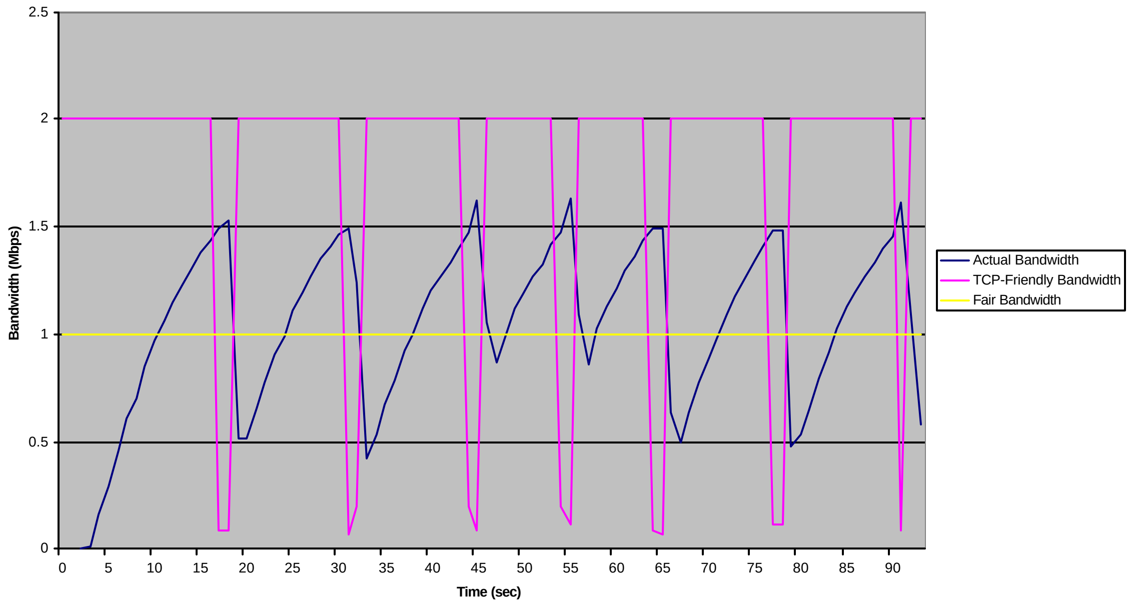

Figure 4-34: TCP Friendly & Fair Bandwidth Overlay for MM-App-New in MM-App-New Vs. TCP where MM-App-New is Fragile

4.3.2.3 Effect of Delay

This simulation shows us that, under greater delay, it takes longer for TCP to get its ACKS/NACKS, such that MM-App-New tends to become greedier in terms of bandwidth usage. Figures 4-35 and 4-36 shows that in terms of fairness, TCP averages below the fair line, where as MM-App-New has an average above the fair line. In terms of TCP-Friendliness, both are friendly, except when MM-App-New experiences a drop.

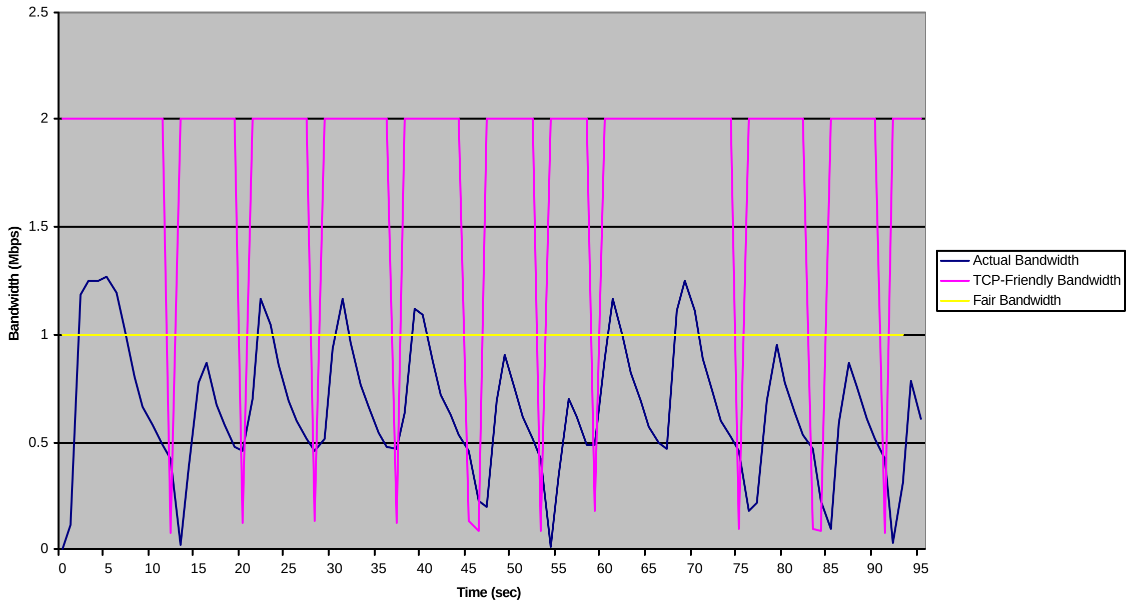

Figure 4-35: TCP Friendly & Fair Bandwidth Overlay for TCP in MM-App-New vs. TCP with Delay 40

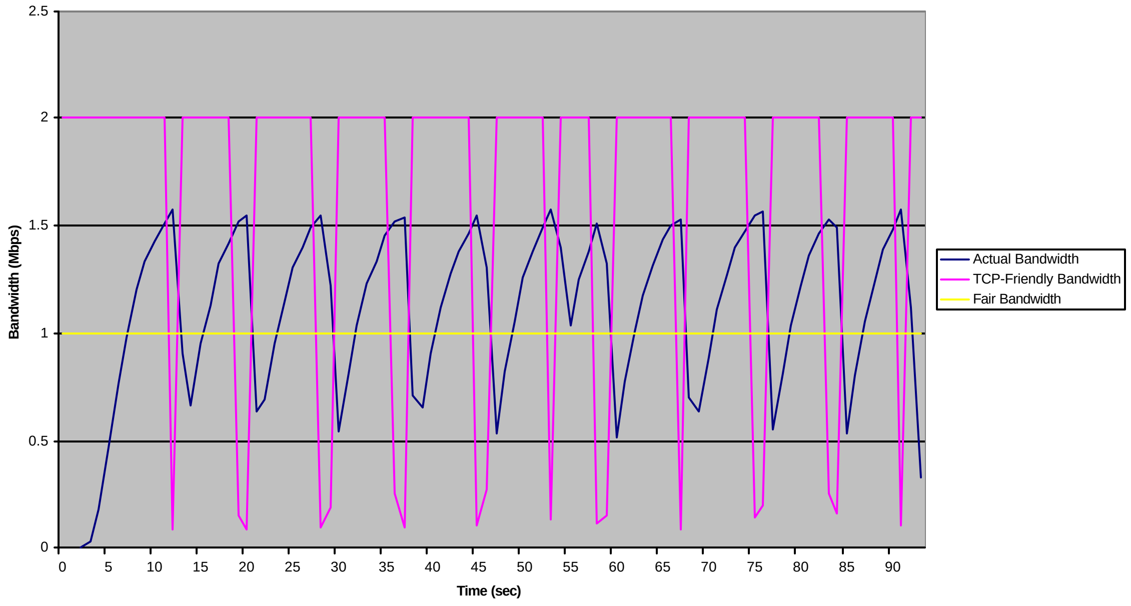

Figure 4-36: TCP Friendly & Fair Bandwidth Overlay for MM-App-New in MM-App-New vs. TCP with Delay 40

4.3.3 TFRC vs. TCP

In order to be able to compare TFRC with MM-App-New equally, we feel it is necessary to compare it to a TCP flow as well.

4.3.3.1 Basic Simulation

This particular simulation showed us some surprising results regarding TFRC. Again, because there were few drops in this scenario, our TCP-Friendly data values are usually equal to the maximum bandwidth. First, Figure 4-37 shows the values for the TCP flow. As can be seen, it starts climbing high at first, but then comes down rapidly. This is due to the fact that TFRC starts running. TCP tries to recover and climbs back up to where its fair bandwidth usage should be, but is forced down.

Now, let’s take a look at the values for TFRC. Figure 4-38 shows that TFRC starts and immediately climbs as high as possible. After adjusting in accordance to the drop it experiences, TFRC begins to climb again, however it never falls below 1.3 Mbps again until the end of the simulation. Due to the low drop rate, both flows are considered to be TCP-friendly. By inspection, however, we can easily see that TFRC is not being fair by consuming much more bandwidth than it should. Based on the information gathered in this simulation, we conclude that TFRC is not as fair as MM-App-New when running against TCP in the standard bottleneck layout.

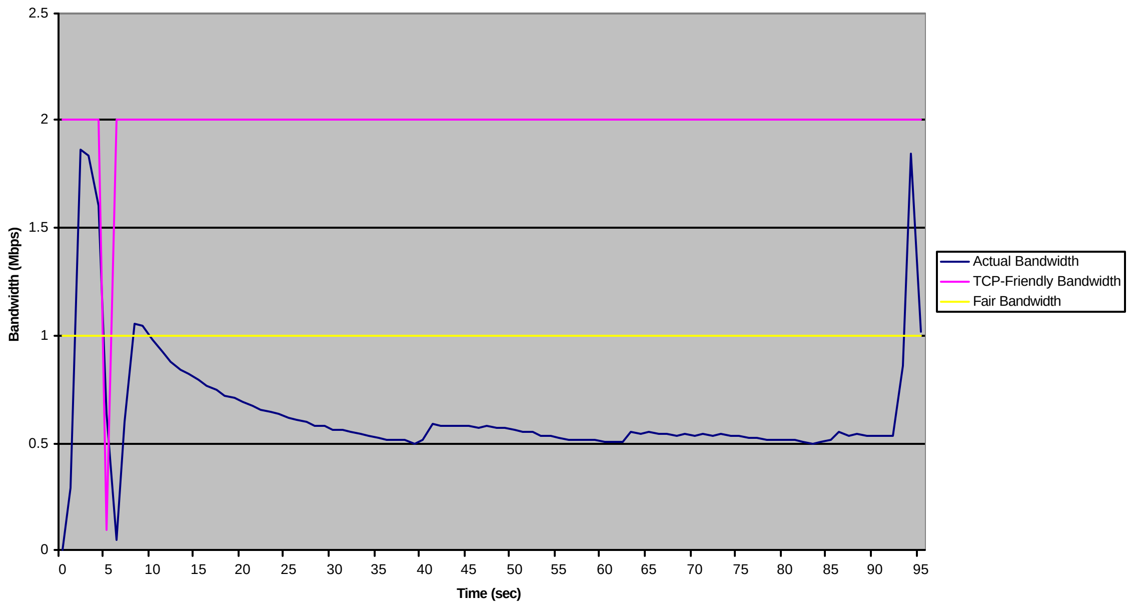

Figure 4-37: TCP Friendly & Fair Bandwidth Overlay for TCP in TFRC vs. TCP

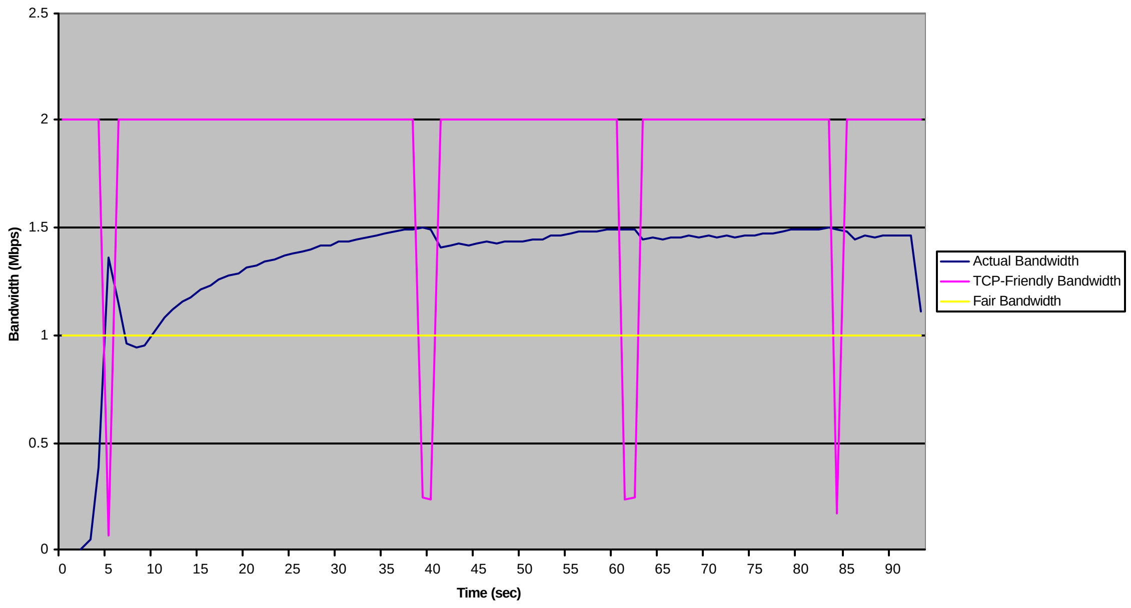

Figure 4-38: TCP Friendly & Fair Bandwidth Overlay for TFRC in TFRC vs. TCP

4.3.3.2 Effect of Fragile Flows

This simulation shows us how both TFRC and TCP react when each one is set to be a fragile flow. First we made TCP the fragile flow. The TCP-friendly and Fair Bandwidth overlays for TCP and TFRC under these circumstances are shown in Figures 4-39 and 4-40 respectively. Next we made TFRC the fragile flow. The TCP-friendly and Fair Bandwidth overlays for TCP and TFRC under these circumstances are shown in Figures 4-41 and 4-42 respectively.

In both sets of figures it is apparent that both TCP and TFRC are within the limits of TCP-friendliness. It is also obvious that TFRC is occupying way more than its fair share of bandwidth, even when it is the fragile flow. This seems to be a recurring theme among simple TFRC simulations.

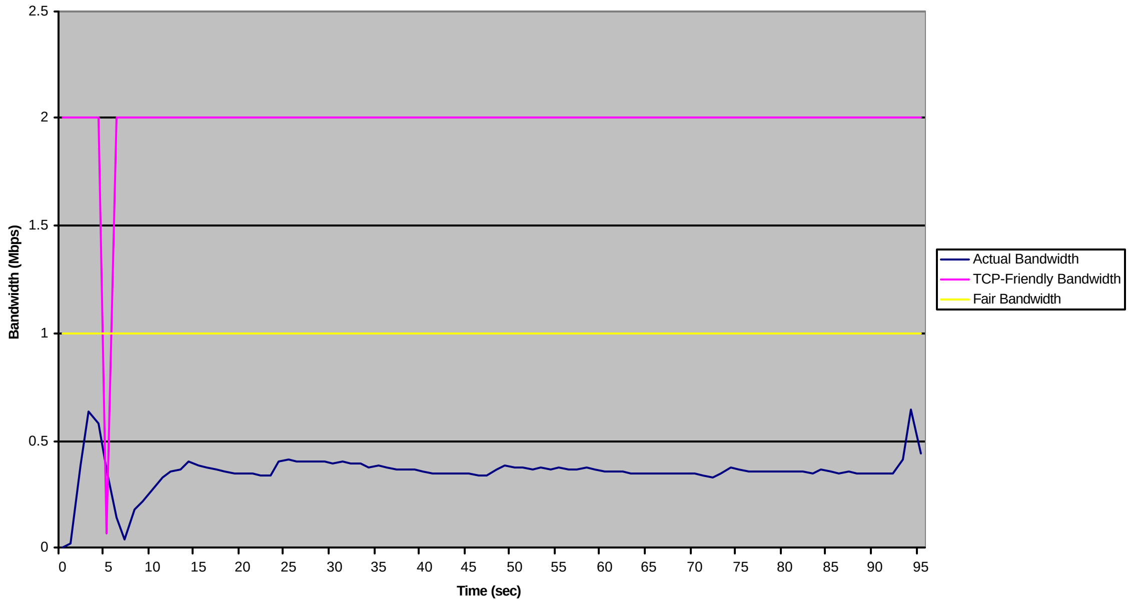

Figure 4-39: TCP Friendly & Fair Bandwidth Overlay for TCP in TFRC vs. TCP when TCP is Fragile

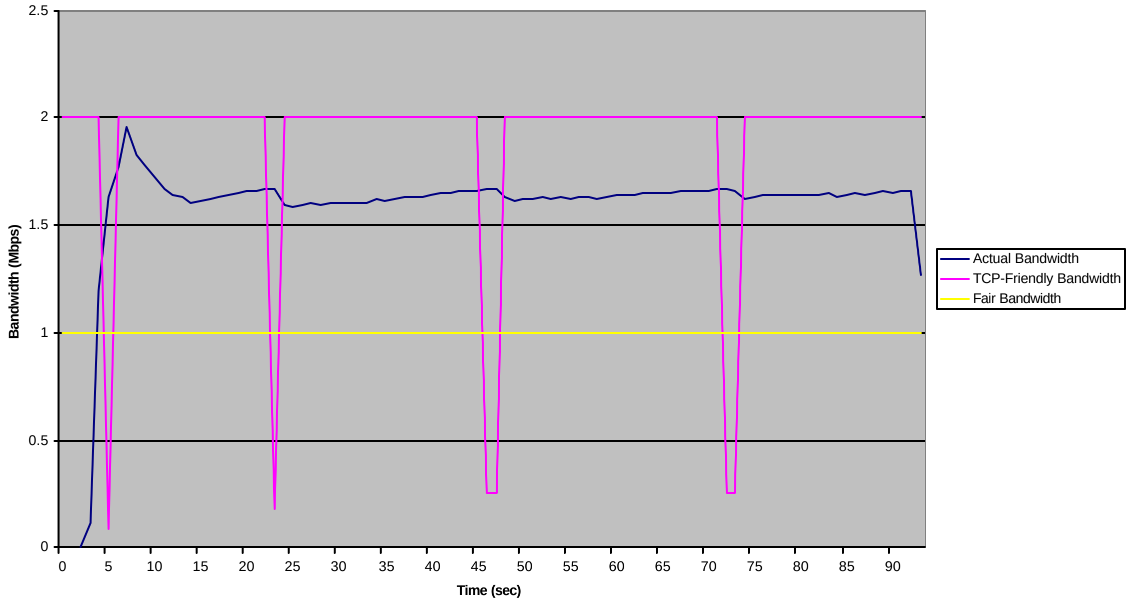

Figure 4-40: TCP Friendly & Fair Bandwidth Overlay for TFRC in TFRC vs. TCP when TCP is Fragile

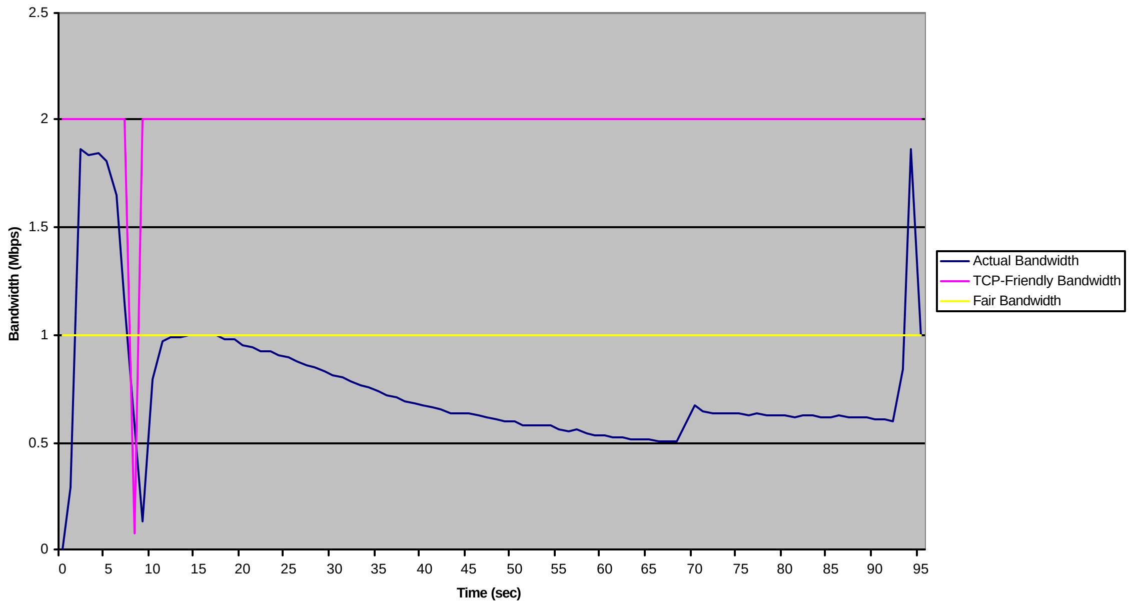

Figure 4-41: TCP Friendly & Fair Bandwidth Overlay for TCP in TFRC vs. TCP when TFRC is Fragile

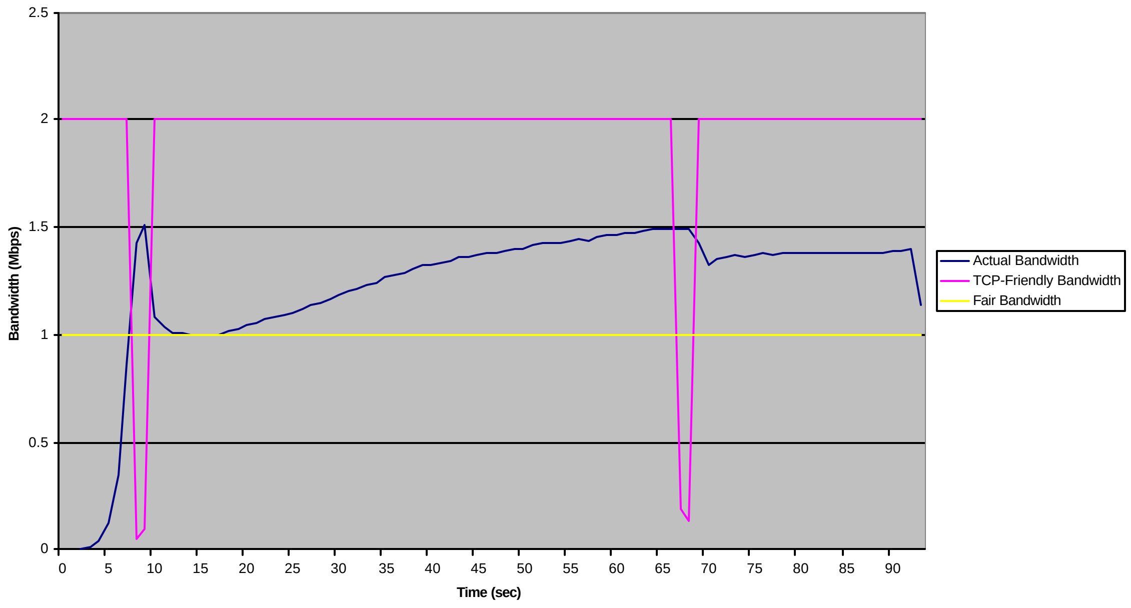

Figure 4-42: TCP Friendly & Fair Bandwidth Overlay for TFRC in TFRC vs. TCP when TFRC is Fragile

4.3.3.3 Effect of Delay

Finally, we tested TFRC vs. TCP with the standard delay layout. The TCP-friendly and Fair Bandwidth measurements for TCP and TFRC are shown in Figures 4-43 and 4-44 respectively. Again, we see that both are within the limits of TCP-friendliness, but TFRC forces TCP to receive an unfair share of the bandwidth.

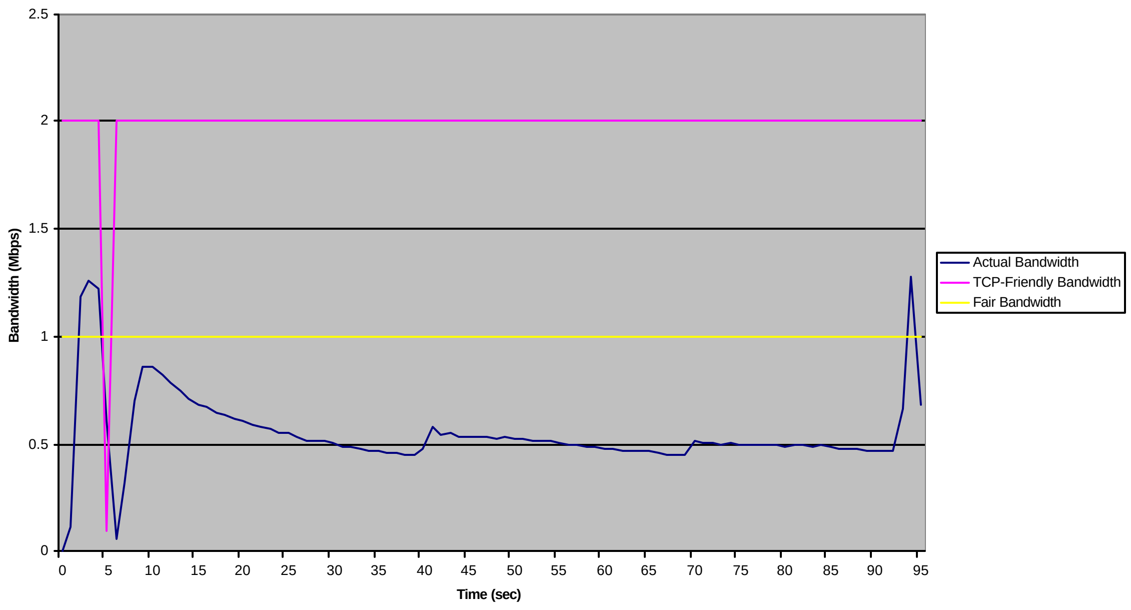

Figure 4-43: TCP Friendly & Fair Bandwidth Overlay for TCP in TFRC vs. TCP with Longer Delay

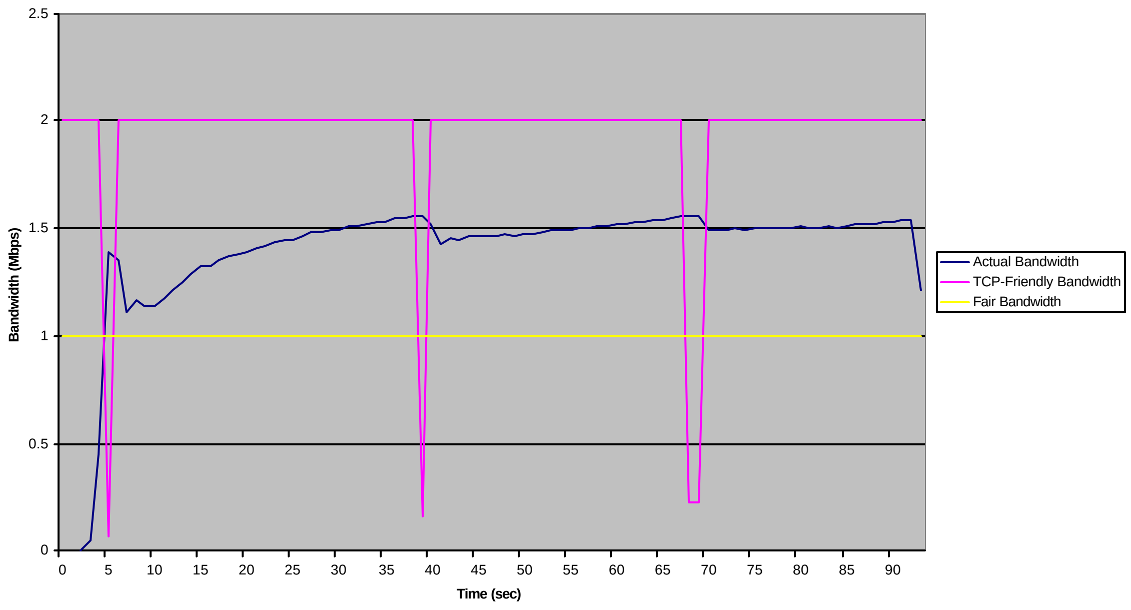

Figure 4-44: TCP Friendly & Fair Bandwidth Overlay for TFRC in TFRC vs. TCP with Longer Delay

4.3.4 MM-App-New vs. TFRC

The simulations so far have showed us that for the most part TFRC is generally over the fair bandwidth line and MM-App-New is generally on average at the fair bandwidth line. These tests are performed against TCP, however, so we decided to test MM-Flow against TFRC in order to gain a better understanding of how they act together. We only ran the basic test for this simulation, as we did not have time to go as in depth on the two. This is partly due to the fact that we are still unsure as to the settings required for TFRC, and therefore do not wish to observe poor performance, only then to find out we had it set up incorrectly.

4.3.4.1 Basic Simulation

Running MM-App-New against TFRC we first examined the TCP-Friendly and Fair bandwidths for TFRC. As can be seen in Figure 4-45, the interval is still too small to gain any useful TCP-Friendly measurements, however we are able to see that on average TFRC seems to be fair once in a stable state. The downside to TFRC is that it takes a good 25 seconds before it reaches this equilibrium, which is a long time, in terms of network traffic.

We next examine MM-App-New in this simulation in terms of TCP-Friendly and fair bandwidths. We expect that, similar to TFRC, our TCP-Friendly values will be inflated due to few drops in the simulation. The actual bandwidth for MM-App-New, as seen in Figure 4-46, shows us upon first inspection that it seems to be relatively close to where TFRC's bandwidth was. The TCP-Friendly values, however, are still relatively inconclusive. The average values for bandwidth and TCP-Friendliness are 1.09 and 1.59 Mbps, respectively.

As shown, MM-App-New quickly gets up to speed in the simulation, taking all the available bandwidth, as it is the only one running for the first two seconds. Then as TFRC starts running, MM-App-New compensates and comes down accordingly in order to be fair. We feel that because the max bandwidth is 2 Mbps, that MM-App-New is running at perfect fairness, due to the fact that its average utilization is 1.09 Mbps.

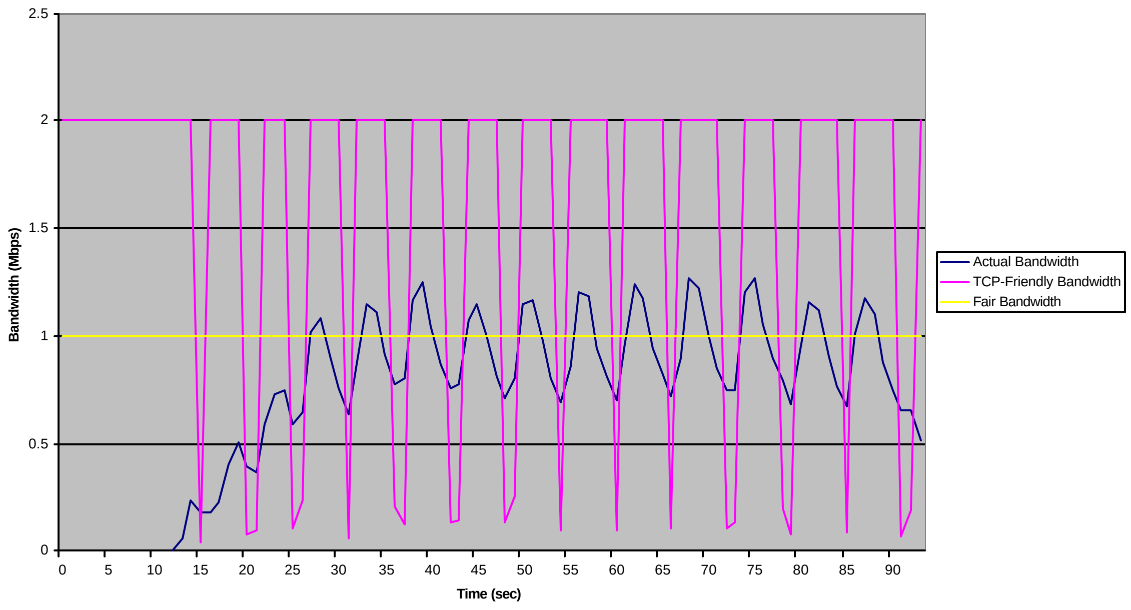

Figure 4-45: TCP Friendly & Fair Bandwidth Overlay for TFRC in TFRC vs. MM-App-New

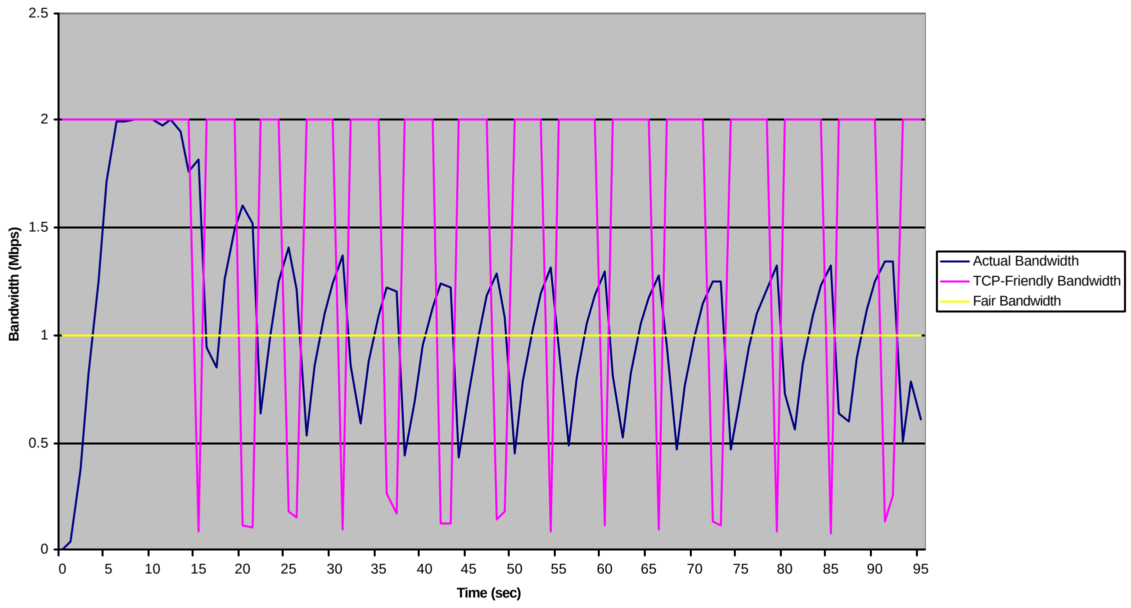

Figure 4-46: TCP Friendly & Fair Bandwidth Overlay for MM-App-New in TFRC vs. MM-App-New

4.3.5 Multiple Protocol Simulation

After running all the above tests separate so that we could find out how each protocol acted against each other, we felt it necessary to perform a more realistic test in which several instances of the protocols ran at the same time. This multiple protocol simulation includes four TCP flows, two MM-App-New flows, and two TFRC flows. We then picked one instance of each flow to evaluate the TCP-friendly and Fair bandwidth values. This simulation provided us with accurate TCP-friendly measurements with only a three second interval since the existence of many protocols insured there were enough drops in the system.

TCP is the first protocol that we examined from this test. As shown in Figure 4-47, TCP's actual bandwidth is above both the TCP-friendly and Fair bandwidths throughout most of the simulation. Figures 4-48 and 4-49 show TCP-friendly and Fair bandwidth measurements for TFRC and MM-App-New respectively. As indicated, TFRC perform very similarly. They are both above the TCP-Friendly bandwidth, but well within the Fair bandwidth, and each provides relatively smooth transmission rates.

Perhaps this simulation raises more questions than it answers, such as why TCP is not TCP-Friendly and why the TCP flows occupy so much bandwidth. Nevertheless, we felt this simulation shows that TFRC and MM-App-New on top of MM-Flow perform rather well in situations that are close to real-world scenarios. As for some of the anomalies in data, we believe they require further examination in future work.

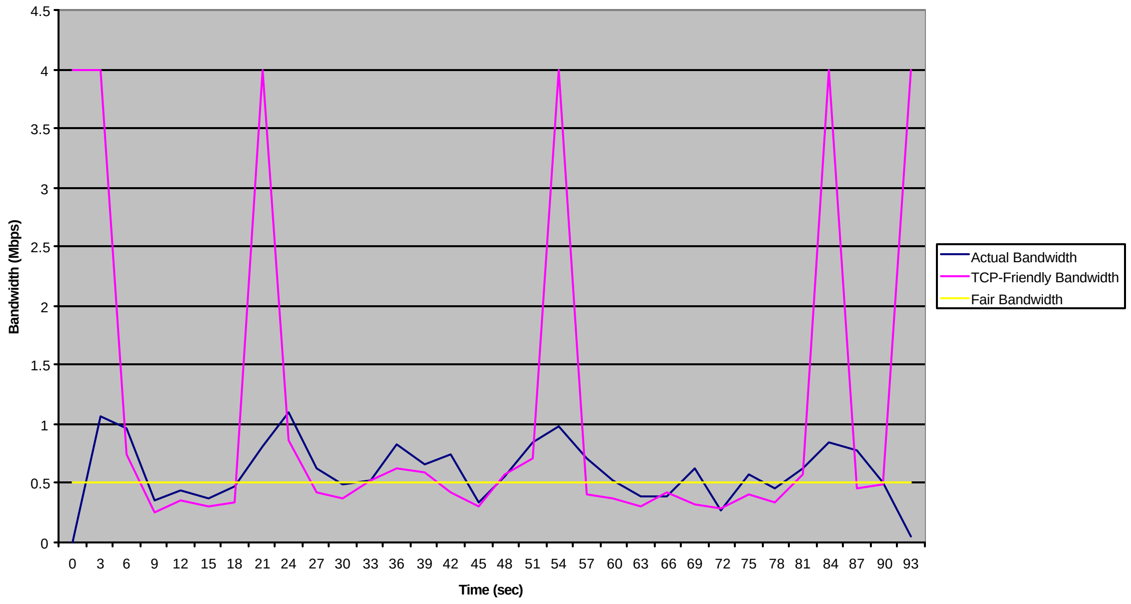

Figure 4-47: TCP Friendly & Fair Bandwidth Overlay for TCP in a Multi-Protocol Environment

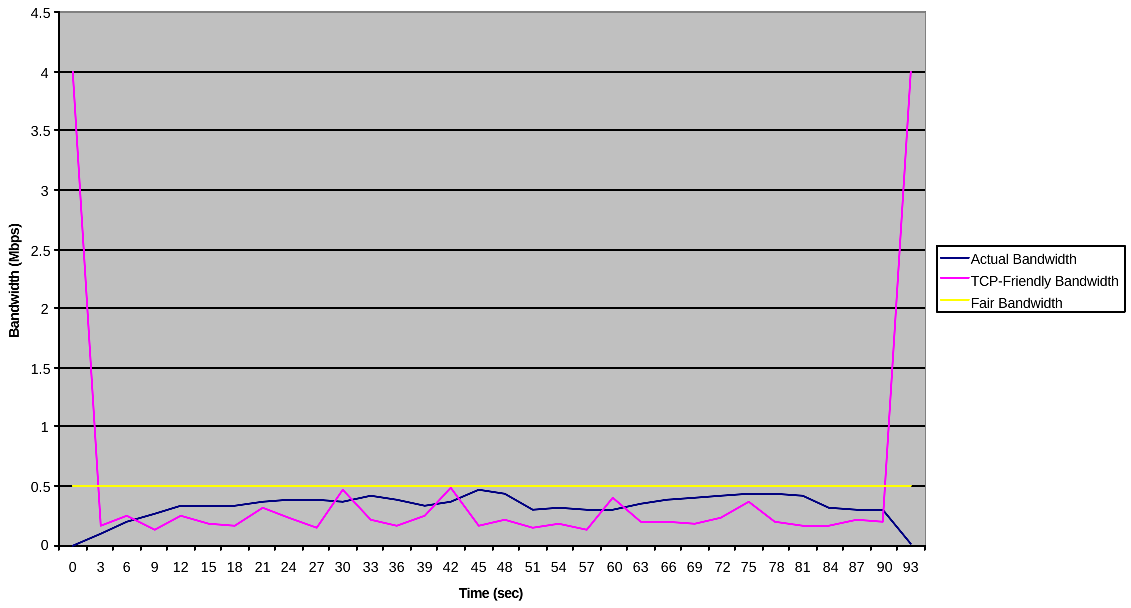

Figure 4-48: TCP Friendly & Fair Bandwidth Overlay for TFRC in a Multi-Protocol Environment

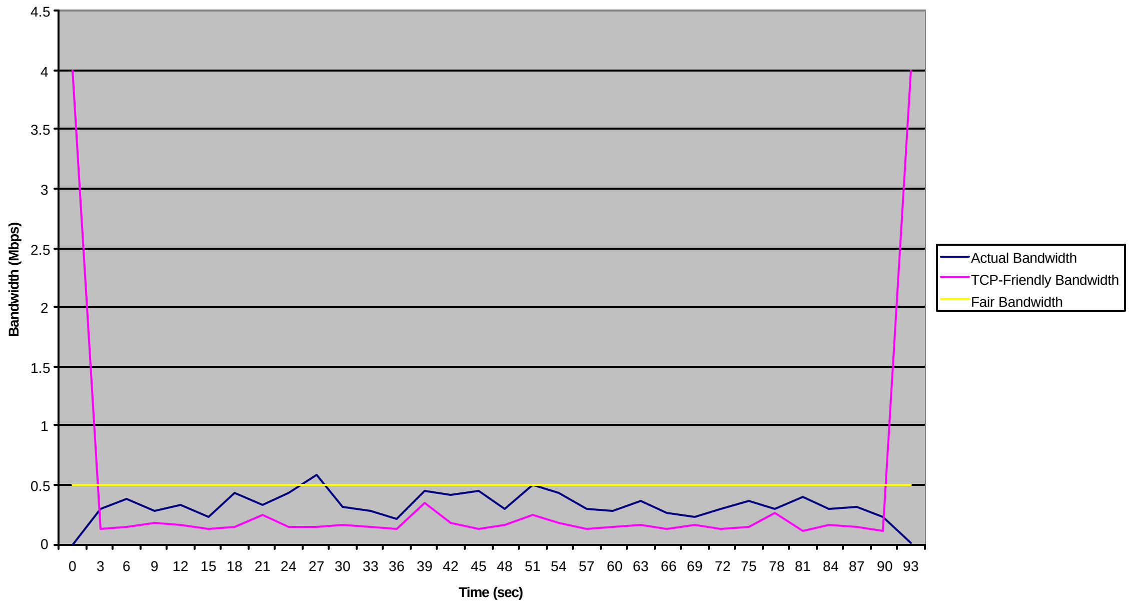

Figure 4-49: TCP Friendly & Fair Bandwidth Overlay for MM-App-New in a Multi-Protocol Environment

5 Conclusion

Before our project there were two application layer multimedia protocols known as MM-App and MPEG-App. Each of these protocols implemented the same set of rules for responding to congestion, but varied in how they mapped scale values to transmission rates. Each used an enhanced version of UDP as its transport agent. The interaction of all of these components and the rules for responding to congestion became known as MM-Flow.

To improve this system, we first sought to split its functionality into an application layer and a new transport layer. This allows for greater extensibility, in that the application layer now runs independent of the decisions that determine what scale value to send at. Applications with new encoding schemes can built on top of the transport layer without having to reproduce its functionality. For clarity, we redefined the term MM-Flow to explicitly refer to the transport layer protocol, while the application layers remained as MM-App and MPEG-App. Another improvement is the option to weight scale values to obtain smoother transmission rates. Also, simulation designers are able to alter the AIMD configuration and number of scale values without recompiling the source code.

After our re-engineering effort, we thoroughly tested our new code under varying circumstances. To aid in our analysis, as well as benefit fellow NS users, we developed a series of generic scripts that transform standard NS trace files into usable data. Our basic simulation scenarios involved a standard bottleneck layout, a standard delay layout, and a standard fragile layout.

We first compared MM-App-Old versus MM-App-New. MM-App-Old occupied more than its fair share of the bandwidth and was limited only by the fact that it had reached it highest transmission rate. MM-App-New, on the other hand, responded to congestion quite well and shared available bandwidth with TCP fairly. We concluded that MM-App-New is an improvement over MM-App-Old, with regards to fairness.

We then continued our analysis of MM-App-New. We have shown that larger packet sizes result in smoother data because it takes longer to send large packets. Changing the frame size had no effect. Increasing the number of scale values created smoother data and caused MM-App-New to use less of the available bandwidth, with MM-App-New and TCP sharing the best when 150 scales was used. When the network involved longer delays, MM-App-New tended to accumulate more of the bandwidth, which was expected since TCP is window-based and MM-Flow is rate based. Fragile flow tests have shown that a fragile TCP flow will lose to a competing MM-App-New, but a fragile MM-App-New is able to fight its way to relatively equal bandwidth against a TCP flow. We have also shown that weighted scale values for MM-Flow can be used to smooth transmission rates, with only a small increase in bandwidth taken.

Similarly, we tested MPEG-App-Old against MPEG-App-New. Results indicated that neither provided smooth data and MPEG-App-New dominated the system. These results were expected, as we believe that MPEG current restriction to five scale values is far too coarse to achieve the desired performance.

We then examined what it means to be a TCP-friendly protocol and found that a widely accepted definition claims that a TCP-friendly flow is one that transmits at a rate less than or equal to a TCP flow under the same conditions. The TCP-friendly bandwidth for a given flow can be calculated by using Equation 2-2. We also have shown that there are some considerations that must be taken into account when using this formula. First, when a flow experiences a drop rate of zero the resulting TCP-friendly bandwidth approaches infinity. For our purposes, we assumed this value to be the same as the maximum bandwidth of the link. We also discussed how increasing the interval size can help us obtain useful measurements, but at the expense of smoothing the actual bandwidth and losing potentially valuable data points.

We conducted a series of tests involving TCP, MM-App-New, and TFRC to examine how the TCP-friendly measurement applies to each. In most of our tests there were extremely low drop rates. Rather than continually increase our interval size, we chose to introduce a measurement of “fair” bandwidth. In general, we found that TCP-friendly measurements obtained from simple simulations did not provide us with much insight.

MM-App-New seemed to compete fairly, but TCP-friendly measurements still were not very helpful due to few drops. In general, measurements of fair bandwidth were consistent with results discussed earlier. TFRC seemed to take on a very odd behavior in simple simulations, such as taking more bandwidth than it should when going against a single TCP flow, regardless of the network layout. When MM-App-New and TFRC competed in a simple simulation, we found that it took a long time for TFRC to reach a stable state with MM-App-New. Once that state was reached, however, they competed rather fairly.

Finally, we placed 4 TCP flows, 2 MM-App-New flows, and 2 TFRC flows in a simulation at the same time. This situation created enough drops to obtain actual lines for

TCP-friendly bandwidth. These values raised more questions than they answered because all flows, including TCP, were indicated as not being TCP-friendly. Also somewhat mysterious is the fact that both MM-App-New and TFRC transmitted below the fair bandwidth, yet TCP was well above it. These results suggest that the TCP-friendly measurement requires more analysis.

In the end, the question remains whether MM-Flow is at the level necessary for today's demands on multimedia protocols. The answer is "not yet." We have given MM-Flow some much-needed improvements, particularly in the realm of fairness and extensibility, but still lacks the smoothness of TFRC. In the next chapter, Future Work, we discuss a vast number of extensions to this project that could help bring MM-Flow up to the desired level of performance.

6 Future Work

MM-Flow is envisioned as an enabling protocol for a multitude of multimedia-based application layer protocols. It is an ongoing project; we would like to finish by presenting a list of ideas for future work, in the hope that it will prove useful to those who wish to study and extend our work. We dealt primarily with the simulated application MM-App, and did not make many changes to MPEG-App, as meaningful changes would require a more in-depth knowledge of MPEG video standards. As a result, the number of scales in MM-App increased from five to a user-specifiable number (defaulting to 50) but MPEG-App remained at only five. It would be interesting to see if MPEG-App could benefit from more scale values in the same way that MM-App has.

It is generally true that increasing the number of scales allows MM-Flow to be more responsive, and at the same time causes it to take up less bandwidth. Ideally, we would want it to share equally with TCP, which pointed us toward the idea of an optimal number of scales. In our preliminary testing, we found that less than 150 scale values caused MM-Flow to take up too much bandwidth, and more than 150 scale values caused MM-Flow to take up too little bandwidth (see Figure 4-14). If 150 scales is indeed an optimal number, perhaps applying our tests to MM-Flow while using 150 scale values would yield better performance.

Much of the success of the TCP protocol is due to its application of the principles of AIMD; one additional packet is sent per round trip time on success and half as many packets are sent per round trip on failure. MM-Flow follows the lead in this regard, using the same parameters to determine how many scales to increase or decrease. This is not quite the same as TCP, in that the scale values are mapped to rates rather than packets.

An increase could signify several additional packets, making MM-Flow potentially more aggressive. As MM-Flow differs from TCP, perhaps different values could be used in the AIMD process; one alternative could be to increase by half of a scale value and decrease to 75% of its former scale value. We decided not to explore this ourselves, as there are many different combinations of these choices available, but we have provided the functionality in MM-Flow to change these values; both the additive increase and multiplicative decrease values may be changed while designing an OTcl simulation scenario.

Since the scale values of MM-Flow are known to increase slowly the first time, another parameter that could be worth changing is the initial scale value. For maximum fairness, we start the protocol at the minimum scale value, but it is possible that another value would prove more optimal. While starting at the maximum would unfairly crowd out other flows, starting in the middle may be reasonable. Further experimentation could determine if starting at a point such as the arithmetic or geometric mean between the two extremes would be better.

Other potential adjustments to MM-Flow's scale system are the advent of a slow-start phase and non-linear scales. While viewing graphs in which MM-Flow and TCP compete for bandwidth, we noticed that TCP quickly takes up all of the free bandwidth, given the chance, whereas MM-Flow's utilization only increases at an approximately linear pace. This appears to be due to TCP's "slow start" algorithm, which contrary to its name allows TCP to achieve substantial bandwidth quickly at the beginning of its run. It does this by increasing its rate after each receipt of an ACK caused by a successful packet arrival, instead of once per round trip time. Adding a slow start phase could be beneficial to MM-Flow in the short-term, as it consistently uses less bandwidth than it can at the beginning of its run. Non-linear scales could achieve the same effect by a different process. If the bandwidth gaps between lower scale values were greater than between higher scale values, MM-Flow could quickly increase in bandwidth before leveling off. However, in a low-bandwidth environment, MM-Flow may be hurt by this scheme by rapidly fluctuating between scale values that are too far apart. Perhaps using a dynamic number of scale values is the key; using a small number of values would allow MM-Flow to increase in bandwidth quickly to start, but switching to a greater number after reaching a predetermined threshold would allow MM-Flow to become smooth after getting established. It could also increase the number of rates used when it is approaching capacity, as determined by longer round trip times. This will cause the protocol to increase its utilization more slowly, in an attempt to avoid congestion. The receiver could effect this change as well, by not sending an ACK when an ACK would normally be sent, or even by sending a NACK. This would be similar to what the RED router does, in that it would force MM-Flow to slow down before it became truly necessary.

For the sake of simplicity, all of our simulations were done using DropTail routers. These routers drop packets only when the queue is full, taking a passive role in congestion situations. RED routers, on the other hand, start dropping packets probabilistically when the queue nears capacity, in order to stave off future congestion. It is possible that the performance of MM-Flow could be affected by using RED instead of DropTail routers. Perhaps with this knowledge, routers could be configured specifically to take advantage of MM-Flow. We decided not to investigate further, as protocols do not typically get to choose which type of routers to use.

One observation made, which we could never satisfactorily explain, is that MM-App and MPEG-App tend to dominate scenarios involving other protocols. We refer to the fact that viewing graphs gives the impression that the applications operate mostly independently of the other protocol (typically TCP), and the other protocol takes the bandwidth that the MM application is not using. We expected to see utilization much lower than full as the two protocols constantly competed over the resource, whereas utilization was at or near full in most circumstances. Our desire is for MM-Flow to compete fairly against other protocols, so we would like to see if this behavior is working against that goal.

MM-Flow's behavior against several protocols was tested, mostly in the situation that the protocols attempted to run continuously for a period of time. On the Internet, much traffic such as web browsing behaves in a way that produces utilization bursts rather than continuous streams of data. Running a web traffic simulator would be useful to see how MM-Flow would compete in real-world situations. Even better would be to adapt MM-Flow for use outside of the NS simulator and test how it fares competing against whatever flows might also be present on the Internet. Unfortunately, the testing situation would be difficult, as the experiments could not be controlled.

Testing MM-Flow against the TEAR protocol would be useful. Created as a response to TFRC, it is suggested that TEAR solves some of the problems we discovered with it. Our intention was originally to test TEAR as we tested TFRC, but we were forced to abandon this goal when we were unable to get the TEAR simulation code to run properly. As this is merely an issue of technical difficulties, we are sure that interesting research can be done once this is working.

Testing MM-Flow's multicast performance by comparison to TEAR could be especially informative. TEAR's system of making decisions at the receiver helps a multicast system by not forcing them to be concentrated in a single host, the sender, but instead distributing the burden to multiple receivers. MM-Flow should be tested using multicast streams, against TEAR and other protocols. One idea from TEAR that could be utilized in a multicast-enabled version of MM-Flow is that of ACKing less often. The current frequency of ACKs sent by the receivers may prove to take too much bandwidth at the sender, as many receivers may be ACKing at the same time.

We considered the implications of losing ACKs due to congestion and drops on the return path. This would cause the sender to lose information on the condition of the path to the receiver, and possibly behave inappropriately. Detecting this situation could be implemented in the form of a timer on the sender; if an ACK or NACK weren't received in the specified time period, the timer would go off, initiating a response on the sender. However, since a NACK could be lost as well as an ACK, there is no easy way to tell what the appropriate response of the sender would be in this circumstance. Our decision was to simply ignore it and continue sending at the rate the sender was currently using. If further study shows that the percentage of lost response packets leans toward ACK or NACK, an "ACK expected" timer could be employed to perform the correct action.

Measuring TCP-Friendly values was problematic for us in some respects. The TCP-Friendly equation relies heavily on the number of dropped packets in a scenario and works best when there are a substantial number of drops. A considerable amount of the time our simulations yielded few drops, and so we could not obtain good TCP-Friendly data. It would have been beneficial to our analysis to have access to a tool for measuring this value independent of the number of drops; perhaps a tool based solely on round trip time. Developing this tool (and investigating if it could be done) would prove a useful avenue for future research.

Finally, another interesting extension to MM-Flow would be writing applications to take advantage of existing multimedia formats and to port existing applications. For example, an application could be developed for streaming audio, based on principles similar to MM-App or MPEG-App, depending on the file format specification. Also, applications that currently use another protocol such as TCP or UDP for multimedia streaming could be adapted to use MM-Flow instead. User experience testing of these applications would be a good real-world measure of the value of MM-Flow. MPEG-App could benefit from user testing as well, in regard to the amount of jitter experienced. While MM-Flow is intended to reduce jitter by avoiding retransmissions and behaving more smoothly than TCP, this was not specifically tested.

7 References

[CC2000] Chung, J. and Claypool, M., “Better-Behaved, Better-Performing Multimedia Networking”, *Society for Computer Simulation Euromedia Conference (COMTEC)*, Antwerp, Belgium, May 8-10, 2000.

[FF1999] Floyd, S. and Fall, K., “Promoting the Use of End-to-End Congestion Control in the Internet”, *IEEE/ACM Transactions on Networking*, May 3, 1999.

[FHPW2000] Floyd, S., Handley, M., Padhye, J., and Widmer, J., “Equation-Based Congestion Control for Unicast Applications”, *SIGCOMM 2000*, May 2000.

[ROY2000] Rhee, I., Ozdemir, V., and Yi, Y., “TEAR: TCP emulation at receivers – flow control for multimedia streaming”, April 28, 2000.

[perform] Chung, J. and Claypool, M., “NS by Example”, *WPI Department of Computer Science*, http://perform.wpi.edu/NS.

Appendix A: MM-Flow.h

```
//
// Modified:  Joel Thibault
//            Jason Ingalsbe
//            Keith Barber
// Date:      02/27/2001
// File Name: mm-flow.h
//
//
// Author:    Jae Chung
// Date:      7/17/99
// File Name: udp-mm.h
//

#ifndef ns_mm_flow_h
#define ns_mm_flow_h

#include "timer-handler.h"
#include "packet.h"
#include "udp.h"
#include "ip.h"

#define PTYPE_UNKNOWN 0
#define PTYPE_MM 1
#define PTYPE_ACK 2
#define PTYPE_NACK 3

#define SCALE_WINDOW_SIZE 8

class MmFlowAgent;

// Reciver uses this timer to schedule
// next ack/nack packet transmission time
class MmFlowRespTimer : public TimerHandler {
 public:
  MmFlowRespTimer(MmFlowAgent* t) : TimerHandler(), t_(t) {}
  inline virtual void expire(Event*);
 protected:
  MmFlowAgent* t_;
};


// Header for MM-App frames and MM-Flow packets
struct hdr_mm_flow {
  int frm_seq;        // frame sequence number
  int frm_tot_bytes;  // total bytes for frame
  char frm_type;      // frame type
  int frm_num;        // frame number

  int pkt_seq;       // packet sequence number
  int pkt_type;      // packet type
  double pkt_time;   // time packet sent
  int pkt_tot_bytes; // size of message

  double max_interval; // maximum length of time to wait for a packet
  static int offset_;
  inline static int& offset() { return offset_; }
  inline static hdr_mm_flow* access(const Packet* p) {
    return (hdr_mm_flow*) p->access(offset_);
  }
};



// Used to re-assemble segmented (by UDP) frames
struct asm_mm {
  int fseq;       // frame sequence number
  int recv_bytes; // currently received bytes
  int tot_bytes;  // total bytes to receive for frame
};


// This is used for receiver's packet accounting
struct pkt_accounting {
        int last_pseq;      // sequence number of last received pkt
        double last_time;   // local time of last received pkt
        int lost_pkts;      // number of lost pkts since last ack
        int recv_pkts;      // number of pkts received since last ack
        int tot_recv_pkts;  // number of total pkts received
        double rtt;         // round trip time
};


// MmFlowAgent Class definition
class MmFlowAgent : public UdpAgent {
 public:
  MmFlowAgent();
  MmFlowAgent(packet_t);
  virtual int supportMM() { return 1; }
  virtual void sendmsg(int nbytes, const char *flags = 0);
  void recv(Packet*, Handler*);
  void send_response();
  pkt_accounting p_accnt;
  int get_scale();
  void set_max_interval(double max_interval);
  void set_max_min_scale(int max, int min);
 protected:
  int command(int argc, const char*const* argv);
  int mm_bit_;      // user supplied response (use mm bit?)
 private:
  void init();
  void init_recv_pkt_accounting();
  void account_recv_pkt(const hdr_mm_flow *mh_buf);
  void send_ack();
  void send_nack();
  void set_scale(const hdr_mm_flow *mm_pkt);

  asm_mm asm_info; // packet re-assembly information

  int nack_flag_;    // should a NACK packet be sent
  int scale_;        // scale value for sending rate
  double max_interval_; // maximum frame transmission interval
  int add_inc_;      // amount to increase the scale upon ACK
  double mult_dec_;  // amount to decrease the scale upon NACK
  int max_scale_;    // maximum scale value
  int min_scale_;    // minimum scale value
  int flow_control_; // should flow control be on
  int weighted_;     // if scale values are weighted over time
  int first_pkt_;    // 1 if this is the first packet received
  int last_pseq_;    // seq number of last packet (sender)

  int scale_window_[SCALE_WINDOW_SIZE]; // holds values for updating
scale

  int weighted_scale_; // value of scale value to use
  int scale_weight_counter_; // keeps order of scale values

  FILE* fd_delay_;

  MmFlowRespTimer  resp_timer_;  // Ack/Nack Timer
};

#endif
```

Appendix B: mm-flow.cc

```
//
// Modified:  Joel Thibault
//            Jason Ingalsbe
//            Keith Barber
// Date:      02/27/2001
// File Name: mm-flow.cc
//
//
// Author:    Jae Chung
// Date:      7/17/99
// File Name: udp-mm.cc
//

#include "mm-flow.h"
#include "rtp.h"
#include "random.h"
#include <string.h>

int hdr_mm_flow::offset_;

// Multimedia Header Class
static class MmFlowPacketHeaderClass : public PacketHeaderClass {
public:
  MmFlowPacketHeaderClass() : PacketHeaderClass("PacketHeader/MmFlow",
                                    sizeof(hdr_mm_flow)) {
    bind_offset(&hdr_mm_flow::offset_);
  }
} class_mm_flow_hdr;


// MmFlowAgent OTcl linkage class
static class MmFlowAgentClass : public TclClass {
public:
  MmFlowAgentClass() : TclClass("Agent/UDP/MmFlow") {}
  TclObject* create(int, const char*const*) {
    return (new MmFlowAgent());
  }
} class_udpmm_agent;


// Constructor (with no arg)
MmFlowAgent::MmFlowAgent() : UdpAgent(), resp_timer_(this)
{
  init();
}


// Constructor (with one arg)
MmFlowAgent::MmFlowAgent(packet_t type) : UdpAgent(type),
resp_timer_(this)
{
  init();
}
```

```
void MmFlowAgent::init() {

  bind("mm_bit_", &mm_bit_);                      // (default 1 = true)
set the priority in the IP packet
  bind_bool("weighted_", &weighted_);             // (default) false:
AIMD scales    true: use weighted scales
  bind_bool("flow_control_", &flow_control_);     // (default) true:
adjust scales to fit conditions    false: stay at max_scale_
  bind("add_inc_", &add_inc_);                    // (default 1) number
to increase scale by in AIMD
  bind("mult_dec_", &mult_dec_);                  // (default 0.5)
newscale = mult_dec_ * oldscale, in AIMD

  last_pseq_ = 0;                                 // sender: last
packet sent (= next) is 0
  first_pkt_ = 1;                                 // set to 1 until
first packet is sent
  scale_weight_counter_ = 0;                      // start at scale
number 0
  fd_delay_ = NULL;                               // file has not been
selected yet
  nack_flag_ = 0;                                 // no need to send a
nack yet
  max_interval_ = 0.0;                            // initialize to
dummy value 0

  asm_info.fseq = -1;                             // expected received
packet = 0.  So the previous was -1

  init_recv_pkt_accounting();

}

// When resp_timer_ expires call MmFlowAgent::send_response()
void MmFlowRespTimer::expire(Event*)
{
  t_->send_response();
}

// OTcl command interpreter
int MmFlowAgent::command(int argc, const char*const* argv)
{
  Tcl& tcl = Tcl::instance();

  // Record mm-packet-arrival to a file
  if(strcmp(argv[1], "record-mm-packet-arrival") == 0) {
    if((fd_delay_ = fopen(argv[2], "w")) == NULL) {
      tcl.resultf("cannot create mm-packet-arrival file \"%s\"",
argv[2]);
      return(TCL_ERROR);
    }
    return(TCL_OK);
  }

  return (Agent::command(argc, argv));
}
```

```
// Add Support of Multimedia Application to UdpAgent::sendmsg
void MmFlowAgent::sendmsg(int nbytes, const char* flags)
{
  Packet *p;
  int n, remain;

  if (size_) {
    n = (nbytes/size_ + (nbytes%size_ ? 1 : 0));
    remain = nbytes%size_;
  }
  else
    printf("Error: UDPmm size = 0\n");

  if (nbytes == -1) {
    printf("Error:  sendmsg() for UDPmm should not be -1\n");
    return;
  }

  double local_time = Scheduler::instance().clock();

  while (n-- > 0) {
    p = allocpkt();
    if(n==0 && remain>0)
      hdr_cmn::access(p)->size() = remain;

    hdr_rtp* rh = hdr_rtp::access(p);
    rh->flags() = 0;
    rh->seqno() = ++seqno_;
    hdr_cmn::access(p)->timestamp() =
(u_int32_t)(SAMPLERATE*local_time);

    // create outgoing header
    hdr_mm_flow* mm = hdr_mm_flow::access(p);
    // cast header coming from app to the appropriate type
    hdr_mm_flow* msg = (hdr_mm_flow*)flags;

    // set frame header values to the values coming in from application
level
    mm->frm_seq       = msg->frm_seq;
    mm->frm_tot_bytes = msg->frm_tot_bytes;
    mm->frm_type      = msg->frm_type;
    mm->frm_num       = msg->frm_num;


    // MM-Flow packets are distinguished by setting the ip
    // priority bit to 15 (Max Priority).
    if(mm_bit_) {  // if user want to set it as mm packet
      hdr_ip* ih = hdr_ip::access(p);
      ih->prio_ = 15; // used by CBT routers
    }

    // set packet header values
    mm->pkt_seq = last_pseq_++;
    mm->pkt_type = PTYPE_MM;
    mm->pkt_time = local_time;
    // give the sender's max interval to the receiver
    mm->max_interval = max_interval_;

    // add "beginning of talkspurt" labels (tcl/ex/test-rcvr.tcl)
    if (flags && (0 ==strcmp(flags, "NEW_BURST")))
      rh->flags() |= RTP_M;

    target_->recv(p);
  }
  idle();

}


// Support Packet Re-Assembly and Multimedia Application
void MmFlowAgent::recv(Packet* p, Handler*)
{
  int bytes_to_deliver = hdr_cmn::access(p)->size();
  hdr_mm_flow *mm = hdr_mm_flow::access(p);

  // check packet type - ACK, NACK, MM, or other
  if (mm->pkt_type == PTYPE_ACK || mm->pkt_type == PTYPE_NACK)
    set_scale(mm);
  else if (mm->pkt_type == PTYPE_MM) {
    account_recv_pkt(mm);

    // sets the appropriate maximum interval on the receiver's side
    max_interval_ = mm->max_interval;

    // if this is the first packet received, start the ACK/NACK timer
    if (first_pkt_) {
      send_response();
      first_pkt_ = 0;
    }

    if(app_) {  // if MM Application exists

      // re-assemble MM Application frame if segmented
      if(mm->frm_seq == asm_info.fseq)
      asm_info.recv_bytes += hdr_cmn::access(p)->size();
      else {
      asm_info.fseq = mm->frm_seq;
      asm_info.tot_bytes = mm->frm_tot_bytes;
      asm_info.recv_bytes = hdr_cmn::access(p)->size();
      }
      // if fully reassembled, pass the frame to application
      if(asm_info.tot_bytes == asm_info.recv_bytes) {
      hdr_mm_flow mh_buf;
      memcpy(&mh_buf, mm, sizeof(hdr_mm_flow));
      app_->recv_msg(mh_buf.frm_tot_bytes, (char*) &mh_buf);
      }
    }
  }

  Packet::free(p);

}
```

```
void MmFlowAgent::init_recv_pkt_accounting()
{
  // initialize packet accounting values
  p_accnt.last_pseq = -1;
  p_accnt.last_time = 0;
  p_accnt.lost_pkts = 0;
  p_accnt.recv_pkts = 0;
  p_accnt.tot_recv_pkts = 0;
}


void MmFlowAgent::account_recv_pkt(const hdr_mm_flow *mh_buf)
{
  double local_time = Scheduler::instance().clock();

  // Count Received packets and Calculate Packet Loss
  p_accnt.tot_recv_pkts ++;
  p_accnt.recv_pkts ++;
  p_accnt.lost_pkts += (mh_buf->pkt_seq - p_accnt.last_pseq - 1);
  p_accnt.last_pseq = mh_buf->pkt_seq;
  p_accnt.last_time = local_time;

  // Calculate RTT
  if(p_accnt.tot_recv_pkts == 1)
    p_accnt.rtt = 2*(local_time - mh_buf->pkt_time);
  else
    p_accnt.rtt = 0.95 * p_accnt.rtt + 0.05 * 2*(local_time - mh_buf-
>pkt_time);

  // Record mm-packet-arrival
  if(fd_delay_ != NULL) {
    fprintf(fd_delay_, "%lf\t%d\t%lf\n",
          local_time, mh_buf->pkt_seq, local_time - mh_buf->pkt_time);
  }

}

// when timer expires, send ACK or NACK depending on circumstances
void MmFlowAgent::send_response() {

  if (p_accnt.recv_pkts > 0 && p_accnt.lost_pkts == 0) {
    send_ack();
  }
  else if (p_accnt.recv_pkts == 0) {
    if (nack_flag_ == 0 && max_interval_ - p_accnt.rtt > 0) {
      nack_flag_ = 1;
      resp_timer_.resched(max_interval_ - p_accnt.rtt);
    }
    else
      send_nack();
  }
  else                     // p_accnt.recv_pkts > 0 &&
p_accnt.lost_pkts > 0
    send_nack();
```

```
}

void MmFlowAgent::send_ack() {
  Packet *p;
  double local_time = Scheduler::instance().clock();

  nack_flag_ = 0;

  p = allocpkt();

  // send ack packet
  hdr_mm_flow* ack_buf = hdr_mm_flow::access(p);
  ack_buf->pkt_seq = 0;
  ack_buf->pkt_type = PTYPE_ACK;  // this packet is ack packet
  ack_buf->pkt_time = local_time;
  ack_buf->pkt_tot_bytes = 40;  // Ack packet size is 40 Bytes
  target_->recv(p);

  resp_timer_.resched(p_accnt.rtt);

  p_accnt.recv_pkts = 0;
  p_accnt.lost_pkts = 0;
}

void MmFlowAgent::send_nack() {
  Packet *p;
  double local_time = Scheduler::instance().clock();

  nack_flag_ = 0;

  p = allocpkt();

  // send nack packet
  hdr_mm_flow* nack_buf = hdr_mm_flow::access(p);
  nack_buf->pkt_seq = 0;
  nack_buf->pkt_type = PTYPE_NACK;  // this packet is nack packet
  nack_buf->pkt_time = local_time;
  nack_buf->pkt_tot_bytes = 40;  // Nack packet size is 40 Bytes
  target_->recv(p);

  resp_timer_.resched(p_accnt.rtt);

  p_accnt.recv_pkts = 0;
  p_accnt.lost_pkts = 0;
}


void MmFlowAgent::set_scale(const hdr_mm_flow *mm_pkt) {
  float newscale;

  //printf("set_scale %d\n", scale_);
  fflush(stdout);

  if (flow_control_)
    {
      // calculate unweighted scale value
      if (mm_pkt->pkt_type == PTYPE_ACK)
      scale_ += add_inc_;
      else if (mm_pkt->pkt_type == PTYPE_NACK)
      scale_ = static_cast<int>(scale_ * mult_dec_);

      // constrain scale to max/min values
      if (scale_ > max_scale_)
      scale_ = max_scale_;
      else if (scale_ < min_scale_)
      scale_ = min_scale_;

      // weight scale values
      scale_window_[scale_weight_counter_] = scale_;

      newscale  = .20 * scale_window_[scale_weight_counter_];
      newscale += .15 * scale_window_[(scale_weight_counter_ + 7) %
SCALE_WINDOW_SIZE];
      newscale += .15 * scale_window_[(scale_weight_counter_ + 6) %
SCALE_WINDOW_SIZE];
      newscale += .10 * scale_window_[(scale_weight_counter_ + 5) %
SCALE_WINDOW_SIZE];
      newscale += .10 * scale_window_[(scale_weight_counter_ + 4) %
SCALE_WINDOW_SIZE];
      newscale += .10 * scale_window_[(scale_weight_counter_ + 3) %
SCALE_WINDOW_SIZE];
      newscale += .10 * scale_window_[(scale_weight_counter_ + 2) %
SCALE_WINDOW_SIZE];
      newscale += .10 * scale_window_[(scale_weight_counter_ + 1) %
SCALE_WINDOW_SIZE];

      scale_weight_counter_ = (scale_weight_counter_ + 1) %
SCALE_WINDOW_SIZE;

      weighted_scale_ = static_cast<int>(newscale + 0.5);
    }


  //printf("set_scale out scale = %d weighted = %d\n", scale_,
weighted_scale_);
  fflush(stdout);


}

int MmFlowAgent::get_scale() {

  //printf("get_scale scale_ = %d w_scale_ = %d\n", scale_,
weighted_scale_);
  fflush(stdout);

  // both scales are calculated just in case
  if (weighted_) {
    return weighted_scale_;
  }
  else {
    return scale_;
  }
```

```
}

// called by application layer to set the max interval so that mm-flow
doesn't time out inappropriately
void MmFlowAgent::set_max_interval(double max_interval) {

  max_interval_ = max_interval;
}


void MmFlowAgent::set_max_min_scale(int max, int min) {

  //printf("set max min max_in = %d min_in = %d\n", max, min);
  fflush(stdout);

  max_scale_ = max;
  min_scale_ = min;

  if (flow_control_) {
    scale_ = min_scale_;
    weighted_scale_ = min_scale_;

    // initialize window to minimum values
    for (int loop = 0; loop < SCALE_WINDOW_SIZE; loop++)
      scale_window_[loop] = min_scale_;
  }
  else {
    scale_ = max_scale_;
    weighted_scale_ = max_scale_;
  }

  //printf("set max min max_out = %d min_out = %d scale = %d w_scale =
%d\n",
  // max_scale_, min_scale_, scale_, weighted_scale_);
  fflush(stdout);

}
```

Appendix C: MM-Flow Parameters

- `flow_control_ (default = true)`
  This value determines if MM-Flow uses flow control algorithms to avoid congestion. If false, MM-Flow will send constantly at the maximum rate.
- `weighted_ (default = false)`
  This value determines if MM-Flow uses weighted scale values or AIMD scale values in calc ulating the current scale.
- `mm_bit_ (default = 1)`
  A value of 1 indicates that packets will have the "mm" priority bit set at the IP level. Other values will cause this bit to not be set.
- `add_inc_ (default = 1)`
  This value sets the number of scale values to increase by on receipt of an ACK packet, in AIMD.
- `mult_dec_ (default = 0.5)`
  This value sets the percentage of scale values to decrease by on receipt of a NACK packet, in AIMD.

Additionally, an output file can be specified to record frame arrival times, in the format "<flow> record-mm-packet-arrival <tracefile>."

Appendix D: mm-app-new.h

```
//
// Modified:  Joel Thibault
//            Jason Ingalsbe
//            Keith Barber
// Date:      02/27/2001
// File Name: mm-app-new.h
//
//
// Author:    Jae Chung
// Date:      10/05/99
// File Name: mm-app.h
//

#ifndef ns_mm_app_new_h
#define ns_mm_app_new_h

#include "timer-handler.h"
#include "app.h"
#include "mm-flow.h"


class MmAppNew;


// Sender uses this timer to
// schedule next frame transmission time
class MmAppNewSendTimer : public TimerHandler {
 public:
  MmAppNewSendTimer(MmAppNew* t) : TimerHandler(), t_(t) {}
  inline virtual void expire(Event*);
 protected:
  MmAppNew* t_;
};


// Multimedia Application Class Definition
class MmAppNew : public Application {
 public:
  MmAppNew();
  void send_frame();  // called by SendTimer:expire (Sender)
 protected:
  int command(int argc, const char*const* argv);
  void start();       // Start sending frames (Sender)
  void stop();        // Stop sending frames (Sender)
 private:
  inline double next_snd_time();                          // (Sender)
  virtual void recv_msg(int nbytes, const char *msg = 0); //
(Sender/Receiver)
  void calc_rates();     // Binds TCL rates to internal rates

  double *rate;          // Transmission rates associated to scale
values
  double interval_;      // Application frame transmission interval
  double max_bandwidth_; // Maximum possible bandwidth
  int min_scale_;        // Minimum scale allowed for the application
  int max_scale_;        // Maximum scale allowed for the application
  int frmsize_;          // Application frame size
  int random_;           // If 1 add randomness to the interval
  int running_;          // If 1 application is running
  int fseq_;             // Application frame sequence number
  int scale_;            // Media scale parameter

  FILE* fd_scale_;       // file to write scale values to
  int file_closed_;      // input file closed flag

  pkt_accounting p_accnt;
  MmAppNewSendTimer snd_timer_;  // SendTimer
};

#endif
```

Appendix E: mm-app-new.cc

```
//
// Modified:  Joel Thibault
//            Jason Ingalsbe
//            Keith Barber
// Date:      02/27/2001
// File Name: mm-app-new.cc
//
//
// Author:    Jae Chung
// Date:      10/05/99
// File Name: mm-app.cc
//

#include "random.h"
#include "mm-app-new.h"

// MmApp OTcl linkage class
static class MmAppNewClass : public TclClass {
 public:
  MmAppNewClass() : TclClass("Application/MmAppNew") {}
  TclObject* create(int, const char*const*) {
    return (new MmAppNew);
  }
} class_app_mm_new;


// When snd_timer_ expires call MmAppNew:send_frame()
void MmAppNewSendTimer::expire(Event*)
{
  t_->send_frame();
}


// Constructor (also initialize instances of timers)
MmAppNew::MmAppNew() : running_(0), snd_timer_(this)
{
  bind("min_scale_", &min_scale_);                             //
minimum scale for the application
  bind("max_scale_", &max_scale_);                             //
maximum scale for the application
  bind_bw("max_bandwidth_", &max_bandwidth_);                  //
bandwidth used at max_scale_
  bind("frmsize_", &frmsize_);                                 //
size of one frame
  bind_bool("random_", &random_);                              // use
randomness in intervals

  fd_scale_ = NULL;                                            // no
file has been opened ...
  file_closed_ = 0;                                            // ...
or closed
}
```

```
// Linearly interpolates what the scale rates should be
//
// rate[min_scale_ - 1] would be 0 if it were set
// rate[max_scale_] is equal to max_bandwidth_
//
void MmAppNew::calc_rates() {

  int num_rates = max_scale_ - min_scale_ + 1;   // the number of valid
rates
  double stepsize = max_bandwidth_ / num_rates;   // the difference
between one rate and the next

  // allocates space in the array from 0 to max_scale
  rate = (double *)calloc(max_scale_ + 1, sizeof(double));

  for(int looper = min_scale_; looper <= max_scale_; looper++) {
    rate[looper] = stepsize * (looper - min_scale_ + 1);
  }

}

// OTcl command interpreter
int MmAppNew::command(int argc, const char*const* argv)
{

  Tcl& tcl = Tcl::instance();

  if (argc == 3) {
    // Attach Agent
    if (strcmp(argv[1], "attach-agent") == 0) {
      agent_ = (Agent*) TclObject::lookup(argv[2]);
      if (agent_ == 0) {
      tcl.resultf("no such agent %s", argv[2]);
      return(TCL_ERROR);
      }

      // Make sure the underlying agent support MM
      if(!agent_->supportMM()) {
      tcl.resultf("agent \"%s\" does not support MM Application",
argv[2]);
      return(TCL_ERROR);
      }

      agent_->attachApp(this);
      return(TCL_OK);
    }

    // Record MM Scale Value to A File
    if(strcmp(argv[1], "record-mm-scale-value") == 0) {
      if((fd_scale_ = fopen(argv[2], "w")) == NULL) {
      tcl.resultf("cannot create mm-scale-value file \"%s\"", argv[2]);
      return(TCL_ERROR);
      }
      return(TCL_OK);
    }

  }
  return (Application::command(argc, argv));
}


void MmAppNew::start()
{
  //printf("start 1\n");
  fflush(stdout);

  calc_rates();

  //printf("start 2\n");
  fflush(stdout);

  agent_->set_max_interval((double)(frmsize_ <<
3)/(double)rate[min_scale_]);

  //printf("start 3 max = %d min = %d\n", max_scale_, min_scale_);
  fflush(stdout);

  agent_->set_max_min_scale(max_scale_, min_scale_);

  //printf("start 4 scale_ = %d \n", scale_);
  fflush(stdout);

  interval_ = (double)(frmsize_ << 3)/(double)rate[min_scale_];

  //printf("start 5\n");
  fflush(stdout);

  running_ = 1;

  //printf("start 6\n");
  fflush(stdout);

  fseq_ = 0;

  //printf("start 7\n");
  fflush(stdout);

  send_frame();

  //printf("start 8\n");
  fflush(stdout);
}


void MmAppNew::stop()
{
  running_ = 0;

  if (file_closed_ == 0) {
    fclose(fd_scale_);
    file_closed_ = 1;
  }
}
```

```
// Send application frame
void MmAppNew::send_frame()
{
  double local_time = Scheduler::instance().clock();

  hdr_mm_flow mh_buf;

  if (running_) {
    // the below info is passed to MmFlow agent, which will write it
    // to MM header after frame creation.
    mh_buf.frm_seq = fseq_++;         // MM sequence number
    mh_buf.frm_tot_bytes = frmsize_;  // Size of frame
    mh_buf.frm_type = 'N';            // Normal Frame
    mh_buf.frm_num = mh_buf.frm_seq;  // Frame-num is same as seq-num

    agent_->sendmsg(frmsize_, (char*) &mh_buf);  // send to UDP

    // gets scale to determine next send time
    scale_ = agent_->get_scale();

    //printf("send frame post getscale scale = %d\n", scale_);
    fflush(stdout);

    // Reschedule the send_pkt timer
    double next_time_ = next_snd_time();

    if(next_time_ > 0)
      snd_timer_.resched(next_time_);

    // Record mm-scale-value
    if(fd_scale_ != NULL)
      {
      fprintf(fd_scale_, "%lf\t%d\n", local_time, scale_);
      }

  }

}


// Schedule next frame transmission time
double MmAppNew::next_snd_time()
{
  // Recompute interval in case rate or size chages
  interval_ = (double)(frmsize_ << 3)/(double)rate[scale_];
  double next_time_ = interval_;
  if(random_)
    next_time_ += interval_ * Random::uniform(-0.2, 0.2);

  return next_time_;
}


// Receive message from underlying agent
void MmAppNew::recv_msg(int nbytes, const char *msg = 0)
```

```
{
  // does nothing
}
```

Appendix F: MM-App-New Parameters

- `min_scale_ (default = 0)`

  This is the lowest scale value to use.

- `max_scale_ (default = 50)`

  This is the largest scale value to use.

- `max_bandwidth_ (default = 1.5mb)`

  This value defines the sending rate used by the highest scale value.

  Sending rates will be interpolated between min_scale_ and max_scale_.

- `frmsize_ (default = 2000)`

  This value determines the size of frames to send.

- `random_ (default = false)`

  When set to true, this value adds randomness to the sending interval time,

  using the formula interval = interval * (1 + random(-0.2, 0.2))

MM-App scale values can be output to a trace file, by using “<flow> record-mm-scale-value <tracefile>.”

Appendix G: mm-app-mpeg-new.h

```
//
// Modified:  Joel Thibault
//            Jason Ingalsbe
//            Keith Barber
// Date:      02/27/2001
// File Name: mm-app-mpeg-new.h
//
//
// Author:    Jae Chung
// Date:      10/10/99
// File Name: mm-app-mpeg.h
//

#ifndef ns_mm_app_mpeg_new_h
#define ns_mm_app_mpeg_new_h

#include "timer-handler.h"
#include "packet.h"
#include "app.h"
#include "mm-flow.h"


class MmAppMpegNew;


// Sender uses this timer to
// schedule next app data frame transmission time
class MmAppMpegNewSendTimer : public TimerHandler {
 public:
  MmAppMpegNewSendTimer(MmAppMpegNew* t) : TimerHandler(), t_(t) {}
  inline virtual void expire(Event*);
 protected:
  MmAppMpegNew* t_;
};


// Multimedia Application Class Definition
class MmAppMpegNew : public Application {
 public:
  MmAppMpegNew();
  void send_frame();  // called by SendTimer:expire (Sender)
 protected:
  int command(int argc, const char*const* argv);
  void start();       // Start sending frames (Sender)
  void stop();        // Stop sending frames (Sender)
 private:
  void init();
  int  get_frame_size(char* ftype);                       // (Sender)
  void update_recv_frame_type(char frame_type);           // (Sender)
  virtual void recv_msg(int nbytes, const char *msg = 0); //
(Sender/Receiver)
```

```
  double rate[5];        // Transmission rates associated with scale
values
  double interval_;      // Application frame transmission interval
  double max_interval_;  // Maximum possible transmission interval
  int min_scale_;        // Minimum scale allowed for the application
  int max_scale_;        // Maximum scale allowed for the application
  int frame_per_sec_;    // Mpeg Frame Rate
  int framesize_;        // Application frame size
  int running_;          // If 1 application is running
  int fseq_;             // Application frame sequence number (sent
only)
  int fnum_;             // Frame number (account for not sent frames
also)
  int scale_;            // Media scale parameter

  FILE* fdr_;            // file descriptor for input file
  FILE* fd_scale_;       // file to write scale values to
  int file_closed_;      // input file closed flag

  char recv_frame_type[9]; // 9 previously read mpeg frame type
  int  p_frame_sent_;      // P frame drop flag for IBBPBBI at scale 1

  MmAppMpegNewSendTimer snd_timer_;  // SendTimer
};

#endif
```

Appendix H: mm-app-mpeg-new.cc

```
//
// Modified:  Joel Thibault
//            Jason Ingalsbe
//            Keith Barber
// Date:      02/27/2001
// File Name: mm-app-mpeg-new.cc
//
//
// Author:    Jae Chung
// Date:      10/10/99
// File Name: mm-app-mpeg.cc
//

#include "random.h"
#include "mm-app-mpeg-new.h"
#include <string.h>


// MmAppMpegNew OTcl linkage class
static class MmAppMpegNewClass : public TclClass {
 public:
  MmAppMpegNewClass() : TclClass("Application/MmAppMpegNew") {}
  TclObject* create(int, const char*const*) {
    return (new MmAppMpegNew);
  }
} class_app_mpeg_new;


// When snd_timer_ expires call MmAppMpegNew:send_frame()
void MmAppMpegNewSendTimer::expire(Event*)
{
  t_->send_frame();
}


// Constructor (also initialize instances of timers)
MmAppMpegNew::MmAppMpegNew() : running_(0), snd_timer_(this)
{
  bind("min_scale_", &min_scale_);                    // minimum scale
to use
  bind("max_scale_", &max_scale_);                    // maximum scale
to use
  bind("frame_per_sec_", &frame_per_sec_);            // number of
video frames that can be sent per second

  fdr_ = NULL;                                        // no file has
been opened or closed yet
  fd_scale_ = NULL;
  file_closed_ = 0;
}


// OTcl command interpreter
```

```
int MmAppMpegNew::command(int argc, const char*const* argv)
{
  Tcl& tcl = Tcl::instance();

  if (argc == 3) {
    // Attach Agent
    if (strcmp(argv[1], "attach-agent") == 0) {
      agent_ = (Agent*) TclObject::lookup(argv[2]);
      if (agent_ == 0) {
      tcl.resultf("no such agent %s", argv[2]);
      return(TCL_ERROR);
      }

       // Make sure the underlying agent support MM
      if(!agent_->supportMM()) {
      tcl.resultf("agent \"%s\" does not support MM Application",
argv[2]);
      return(TCL_ERROR);
      }

     agent_->attachApp(this);
      return(TCL_OK);
    }
    // Get Mpeg Trace Input File Name
    if (strcmp(argv[1], "mpeg-trace-input") == 0) {
      if ((fdr_ = fopen(argv[2], "r")) == NULL) {
      tcl.resultf("cannot open mpeg-trace-input file \"%s\"", argv[2]);
        return(TCL_ERROR);
      }
      return(TCL_OK);
    }

      // Record MM Scale Value to A File
    if(strcmp(argv[1], "record-mm-scale-value") == 0) {
      if((fd_scale_ = fopen(argv[2], "w")) == NULL) {
      tcl.resultf("cannot create mm-scale-value file \"%s\"", argv[2]);
      return(TCL_ERROR);
      }
      return(TCL_OK);
    }

  }

  return (Application::command(argc, argv));
}


void MmAppMpegNew::start()
{
  if(fdr_ == NULL) {
    printf ("MmAppMpegNew Error: specify mpeg-trace-input file.\n");
    exit(1);
  }

  init();

  running_ = 1;
  send_frame();
}


void MmAppMpegNew::init()
{
  fseq_ = 0;              // MM sequence number (start from 0)
  fnum_ = -1;             // MM frame number (start from 0)

  agent_->set_max_min_scale(max_scale_, min_scale_);
  agent_->set_max_interval((double)1/(double)frame_per_sec_);

  p_frame_sent_ = 0;       // Flag for IBBPBBI format: drop every
                           // other 'P' frame at scale 1.
  interval_ = ((double)1/(double)frame_per_sec_);

  recv_frame_type[0] = 'B';
  recv_frame_type[1] = 'B';
  recv_frame_type[2] = 'P';
  recv_frame_type[3] = 'B';
  recv_frame_type[4] = 'B';
  recv_frame_type[5] = 'P';
  recv_frame_type[6] = 'B';
  recv_frame_type[7] = 'B';
  recv_frame_type[8] = 'I';
}


void MmAppMpegNew::stop()
{
  running_ = 0;

  if (file_closed_ == 0) {
    fclose(fdr_);
    fclose(fd_scale_);
    file_closed_ = 1;
  }
}


// Send application frame
void MmAppMpegNew::send_frame()
{
  double local_time = Scheduler::instance().clock();

  hdr_mm_flow mh_buf;
  char ftype;

  if (running_) {

    // gets scale
    scale_ = agent_->get_scale();


    // Get Size of next frame
    if ((framesize_ = get_frame_size(&ftype)) == -1) {
      stop();       // if EOF then stop
      return;
    }

    // Increment frame number
    // (account for not sent frames also)
    fnum_++;

    // framesize_ = (positive integer, 0, -1)
    // where 0 means due to network congestion, sender will not
    // send the frame

    if (framesize_ > 0) {
      if(agent_->supportMM()) {
      // the below info is passed to MM-Flow agent, which will write it
      // to MM header after frame creation.
      mh_buf.frm_seq = fseq_++;       // MM sequence number

      mh_buf.frm_tot_bytes = framesize_;  // Size of frame
      mh_buf.frm_type = ftype;      // Frame-type (I,B,P)
      mh_buf.frm_num = fnum_;

      agent_->sendmsg(framesize_, (char*) &mh_buf);  // send to UDP

      }
      else {
      agent_->sendmsg(framesize_);
      }
    }

    // Reschedule the send_frame timer
    snd_timer_.resched(interval_);

    // Record mm-scale-value
    if(fd_scale_ != NULL)
      {
      fprintf(fd_scale_, "%lf\t%d\n", local_time, scale_);
      }

  }
}


// Read Next frame type and size from input file and
// determine whether or not transmit depending on current scale value
int MmAppMpegNew::get_frame_size(char* ftype)
{
  char frame_type_;
  int  size_read_;
  int  frame_size_ = 0;

  // if EOF return -1 to finish
  if (fscanf(fdr_, "%c\t%d\n", &frame_type_, &size_read_) == EOF)
return -1;

  // Different transmission policy for each media scale level
  // Initially, frame_size_ = 0
  switch (scale_) {
  case 0:
    if (frame_type_ == 'I') frame_size_ = size_read_;
    break;

  case 1:
    if (frame_type_ == 'I') frame_size_ = size_read_;
    if (frame_type_ == 'P') {
      if (recv_frame_type[2] == 'I') {
      // for IBBPBBPBBI and IBBPBBPBBPBBI format
      if (recv_frame_type[8] == 'P') {
        frame_size_ = size_read_;
      }
      // for IBBPBBI format
      else {
        if (p_frame_sent_ == 0) {
          frame_size_ = size_read_;
          p_frame_sent_ = 1;
        }
        else
          p_frame_sent_ = 0;
      }
      }
    }
    break;

  case 2:
    if (frame_type_ == 'I') frame_size_ = size_read_;
    if (frame_type_ == 'P') frame_size_ = size_read_;
    break;

  case 3:
    if (frame_type_ == 'I') frame_size_ = size_read_;
    if (frame_type_ == 'P') frame_size_ = size_read_;
    if ((frame_type_ == 'B') && (recv_frame_type[0] == 'B'))
frame_size_ = size_read_;
    break;

  case 4:
    frame_size_ = size_read_;
    break;

  default:
    printf("Error: unrecognized frame type\n");
    exit(1);
    break;
  }

  update_recv_frame_type(frame_type_);
  *ftype = frame_type_;
  return frame_size_;
}


// Keep track of 9 previously read frame type.
void MmAppMpegNew::update_recv_frame_type(char frame_type)
{
  int i;
  for (i=8; i>0; i--) {
    recv_frame_type[i] = recv_frame_type[i-1];
  }
  recv_frame_type[0] = frame_type;
}



// Receive message from underlying agent
void MmAppMpegNew::recv_msg(int nbytes, const char *msg = 0)
{
  // do nothing
}
```

Appendix I: MM-App-Mpeg-New Parameters

- `frame_per_second_ (default = 30)`

  This value sets the number of frames to send per second.

- `min_scale_ (default = 0)`

  This is the lowest scale value to use.

- `max_scale_ (default = 4)`

  This is the largest scale value to use.

While changing min_scale_ and max_scale_ is permitted, scale values outside of 0-4 have not yet been implemented. MPEG-App may also generate a scale values trace file, by using "<flow> record-mm-scale-value <tracefile>." It also requires an input file, specified by using "<flow> mpeg-trace-input <inputfile>." This file must follow the "IBBPBBPBBI" or "IBBPBBPBBPBBI" formats. Different sending rates will send differing numbers of frames, as follows:

```
rate 0 -  I        I   or  I           I
rate 1 -  I  P     I   or  I  P        I
rate 2 -  I  P  P  I   or  I  P  P  P  I
rate 3 -  I BP BP BI   or  I BP BP BP BI
rate 4 -  IBBPBBPBBI   or  IBBPBBPBBPBBI
```

Appendix J: OTcl Example – basic_MMAppNewUW.tcl

```
# File Name:      basic_MMAppNewUW.tcl
# Authors:        Keith Barber
#                 Joel Thibault
#                 Jason Ingalsbe
# Date:           2/28/01
# Description:    Simulation running MMAppNewUW vs. TCP in a standard
#                 bottleneck link layout.

#Create a simulator object
set ns [new Simulator]

#Define different colors for data flows
$ns color 0 Green
$ns color 1 Red

#Open the nam trace file
set nf [open basic_MMAppNewUW.nam w]
set tf [open basic_MMAppNewUW.tr w]
$ns namtrace-all $nf
$ns trace-all $tf

#Define a 'finish' procedure
proc finish {} {
        global ns nf tf
        $ns flush-trace
      #Close the trace file
        close $nf
        close $tf
      #Execute nam on the trace file
     #   exec nam lostAcks.nam &
        exit 0
}

#Create four nodes
#TCP Node
set s1 [$ns node]
#MMApp Node
set s2 [$ns node]
#Middle Node
set m [$ns node]
#Receiver
set r [$ns node]

#Create links between the nodes
$ns duplex-link $s1 $m 4Mb 20ms DropTail
$ns duplex-link $s2 $m 4Mb 20ms DropTail
$ns duplex-link $m $r 2Mb 20ms DropTail

#Set outbound queue limit
$ns queue-limit $m $r 60

#Set up orientation layout for nam
$ns duplex-link-op $s1 $m orient right-down
```

```
$ns duplex-link-op $s2 $m orient right-up
$ns duplex-link-op $m $r orient right

#Monitor the queue for the possibly congested links
$ns duplex-link-op $m $r queuePos 0.5


###################
# TCP Connections #
###################

#Setup 1st TCP connection
set tcp1_s [new Agent/TCP/Reno]
$tcp1_s set window_ 20
$tcp1_s set packetSize_ 1000
$ns attach-agent $s1 $tcp1_s
set tcp1_r [new Agent/TCPSink]
$ns attach-agent $r $tcp1_r
$ns connect $tcp1_s $tcp1_r
$tcp1_s set fid_ 0


######################
# MMFlow Connections #
######################

#Setup 1st MMF connection
set mmf1_s [new Agent/UDP/MmFlow]
$ns attach-agent $s2 $mmf1_s
set mmf1_r [new Agent/UDP/MmFlow]
$ns attach-agent $r $mmf1_r
$ns connect $mmf1_s $mmf1_r
$mmf1_s set packetSize_ 1000
$mmf1_s set fid_ 1
$mmf1_s set weighted_ false
$mmf1_s set add_inc_ 1
$mmf1_s set mult_dec_ 0.50
$mmf1_r set packetSize_ 1000
$mmf1_r set fid_ 1


#############
# FTP Setup #
#############

#Setup 1st FTP Application
set ftp1 [new Application/FTP]
$ftp1 attach-agent $tcp1_s
$ftp1 set type_ FTP


################
# MM_APP Setup #
################

#Setup 1st MM_Application
set mmapp1_s [new Application/MmAppNew]
```

```
$mmapp1_s attach-agent $mmf1_s
$mmapp1_s set frmsize_ 1000
$mmapp1_s set random_ true
$mmapp1_s record-mm-scale-value "basic_MMAppNewUW.mmappnew1.scl"
$mmapp1_s set max_bandwidth_ 2.0mb
$mmapp1_s set min_scale_ 0
$mmapp1_s set max_scale_ 49

set mmapp1_r [new Application/MmAppNew]
$mmf1_r record-mm-packet-arrival "basic_MMAppNewUW.mmappnew1.dly"
$mmapp1_r attach-agent $mmf1_r


##################
#Schedule events #
##################


$ns at 0.5 "$ftp1 start"
  #Let the ftp application get settled before starting MMapp
$ns at 2.5 "$mmapp1_s start"

$ns at 92.5 "$mmapp1_s stop"
$ns at 94.5 "$ftp1 stop"

#Call the finish procedure after 5 seconds of simulation time
$ns at 95.0 "finish"


#####################
#Run the simulation #
#####################

$ns run
```

Appendix K: OTcl Example – all.tcl

```
# File Name:      all.tcl
# Authors:        Keith Barber
#                 Joel Thibault
#                 Jason Ingalsbe
# Date:           2/28/01
# Description:    Simulation running MMAppNewUW vs. TCP vs. TFRC

#Create a simulator object
set ns [new Simulator]

########### Color vs. Protocol vs. FlowID #############
# Black  => TCP               => 0
# Red    => TCP               => 1
# Blue   => TCP               => 2
# Green  => TCP               => 3
# Orange => MM-AppNewUW       => 4
# White  => MM-AppNewUw       => 5
# Purple => TFRC              => 6
# Grey   => TFRC              => 7
#######################################################
#Define different colors for data flows
$ns color 0 Black
$ns color 1 Red
$ns color 2 Blue
$ns color 3 Green
$ns color 4 Orange
$ns color 5 White
$ns color 6 Purple
$ns color 7 Grey

#Open the nam trace file
set nf [open all.nam w]
set tf [open all.tr w]
$ns namtrace-all $nf
$ns trace-all $tf

#Define a 'finish' procedure
proc finish {} {
        global ns nf tf
        $ns flush-trace
      #Close the trace file
        close $nf
        close $tf
      #Execute nam on the trace file
        #exec nam out.nam &
        exit 0
}

#Create all the nodes
set s0 [$ns node]
set s1 [$ns node]
set s2 [$ns node]
set s3 [$ns node]
set s4 [$ns node]
set s5 [$ns node]
set s6 [$ns node]
set s7 [$ns node]
set r0 [$ns node]
set r1 [$ns node]
set r2 [$ns node]
set r3 [$ns node]
set r4 [$ns node]
set r5 [$ns node]
set r6 [$ns node]
set r7 [$ns node]
set n1 [$ns node]
set n2 [$ns node]

#Create links between the nodes
$ns duplex-link $s0 $n1 4Mb 5ms DropTail
$ns duplex-link $s1 $n1 4Mb 5ms DropTail
$ns duplex-link $s2 $n1 4Mb 5ms DropTail
$ns duplex-link $s3 $n1 4Mb 5ms DropTail
$ns duplex-link $s4 $n1 4Mb 5ms DropTail
$ns duplex-link $s5 $n1 4Mb 5ms DropTail
$ns duplex-link $s6 $n1 4Mb 5ms DropTail
$ns duplex-link $s7 $n1 4Mb 5ms DropTail

$ns duplex-link $n1 $n2 4Mb 20ms DropTail

$ns duplex-link $n2 $r0 4Mb 5ms DropTail
$ns duplex-link $n2 $r1 4Mb 5ms DropTail
$ns duplex-link $n2 $r2 4Mb 5ms DropTail
$ns duplex-link $n2 $r3 4Mb 5ms DropTail
$ns duplex-link $n2 $r4 4Mb 5ms DropTail
$ns duplex-link $n2 $r5 4Mb 5ms DropTail
$ns duplex-link $n2 $r6 4Mb 5ms DropTail
$ns duplex-link $n2 $r7 4Mb 5ms DropTail

$ns queue-limit $n1 $n2 60

$ns duplex-link-op $s0 $n1 orient left-down
$ns duplex-link-op $s1 $n1 orient down
$ns duplex-link-op $s2 $n1 orient right-down
$ns duplex-link-op $s3 $n1 orient right
$ns duplex-link-op $s4 $n1 orient right-up
$ns duplex-link-op $s5 $n1 orient up
$ns duplex-link-op $s6 $n1 orient left-up
$ns duplex-link-op $s7 $n1 orient left

$ns duplex-link-op $n1 $n2 orient right

$ns duplex-link-op $n2 $r0 orient left-up
$ns duplex-link-op $n2 $r1 orient up
$ns duplex-link-op $n2 $r2 orient right-up
$ns duplex-link-op $n2 $r3 orient right
$ns duplex-link-op $n2 $r4 orient right-down
$ns duplex-link-op $n2 $r5 orient down
$ns duplex-link-op $n2 $r6 orient left-down
$ns duplex-link-op $n2 $r7 orient right
```

```
#Monitor the queue for the possibly congested links
$ns duplex-link-op $s0 $n1 queuePos 0.5
$ns duplex-link-op $s1 $n1 queuePos 0.5
$ns duplex-link-op $s2 $n1 queuePos 0.5
$ns duplex-link-op $s3 $n1 queuePos 0.5
$ns duplex-link-op $s4 $n1 queuePos 0.5
$ns duplex-link-op $s5 $n1 queuePos 0.5
$ns duplex-link-op $s6 $n1 queuePos 0.5
$ns duplex-link-op $s7 $n1 queuePos 0.5
$ns duplex-link-op $n1 $n2 queuePos 0.5


###################
# TCP Connections #
###################

#Setup 1st TCP connection
set tcp1_s [new Agent/TCP/Reno]
$tcp1_s set window_ 20
$tcp1_s set packetSize_ 1000
$ns attach-agent $s0 $tcp1_s
set tcp1_r [new Agent/TCPSink]
$ns attach-agent $r0 $tcp1_r
$ns connect $tcp1_s $tcp1_r
$tcp1_s set fid_ 0

#Setup 2nd TCP connection
set tcp2_s [new Agent/TCP/Reno]
$tcp2_s set window_ 20
$tcp2_s set packetSize_ 1000
$ns attach-agent $s1 $tcp2_s
set tcp2_r [new Agent/TCPSink]
$ns attach-agent $r1 $tcp2_r
$ns connect $tcp2_s $tcp2_r
$tcp2_s set fid_ 1

#Setup 3rd TCP connection
set tcp3_s [new Agent/TCP/Reno]
$tcp3_s set window_ 20
$tcp3_s set packetSize_ 1000
$ns attach-agent $s2 $tcp3_s
set tcp3_r [new Agent/TCPSink]
$ns attach-agent $r2 $tcp3_r
$ns connect $tcp3_s $tcp3_r
$tcp3_s set fid_ 2

#Setup 4th TCP connection
set tcp4_s [new Agent/TCP/Reno]
$tcp4_s set window_ 20
$tcp4_s set packetSize_ 1000
$ns attach-agent $s3 $tcp4_s
set tcp4_r [new Agent/TCPSink]
$ns attach-agent $r3 $tcp4_r
$ns connect $tcp4_s $tcp4_r
$tcp4_s set fid_ 3
```

```
###################
# MMF Connections #
###################

#Setup 1st MMApp New connection
set mmf3_s [new Agent/UDP/MmFlow]
$ns attach-agent $s4 $mmf3_s
set mmf3_r [new Agent/UDP/MmFlow]
$ns attach-agent $r4 $mmf3_r
$ns connect $mmf3_s $mmf3_r
$mmf3_s set packetSize_ 1000
$mmf3_s set fid_ 4
$mmf3_s set weighted_ false
$mmf3_r set packetSize_ 1000
$mmf3_r set fid_ 4

set mmf4_s [new Agent/UDP/MmFlow]
$ns attach-agent $s5 $mmf4_s
set mmf4_r [new Agent/UDP/MmFlow]
$ns attach-agent $r5 $mmf4_r
$ns connect $mmf4_s $mmf4_r
$mmf4_s set packetSize_ 1000
$mmf4_s set weighted_ false
$mmf4_s set fid_ 5
$mmf4_r set packetSize_ 1000
$mmf4_r set fid_ 5


#Setup 1st TFRC
set tfrc1 [new Agent/TFRC]
$ns attach-agent $s6 $tfrc1
set tfrcsink1 [new Agent/TFRCSink]
$ns attach-agent $r6 $tfrcsink1
$tfrc1 set fid_ 6
$tfrc1 set packetSize_ 1000
$tfrc1 set discount_ 5
$tfrc1 set printLoss_ 1
$tfrc1 set smooth_ 1
$tfrc1 set printStatus_ 0
$tfrc1 set df_ 0.95
$tfrc1 set ca_ 1
$ns connect $tfrc1 $tfrcsink1

#Setup 2nd TFRC Connection
set tfrc2 [new Agent/TFRC]
$ns attach-agent $s7 $tfrc2
set tfrcsink2 [new Agent/TFRCSink]
$ns attach-agent $r7 $tfrcsink2
$tfrc2 set fid_ 7
$tfrc2 set packetSize_ 1000
$tfrc2 set discount_ 5
$tfrc2 set printLoss_ 1
$tfrc2 set smooth_ 1
$tfrc2 set printStatus_ 0
$tfrc2 set df_ 0.95
$tfrc2 set ca_ 1
```

```
$ns connect $tfrc2 $tfrcsink2


#############
# FTP Setup #
#############

#Setup 1st FTP Application
set ftp1 [new Application/FTP]
$ftp1 attach-agent $tcp1_s
$ftp1 set type_ FTP

#Setup 2nd FTP Application
set ftp2 [new Application/FTP]
$ftp2 attach-agent $tcp2_s
$ftp2 set type_ FTP

#Setup 3rd FTP Application
set ftp3 [new Application/FTP]
$ftp3 attach-agent $tcp3_s
$ftp3 set type_ FTP

#Setup 4th FTP Application
set ftp4 [new Application/FTP]
$ftp4 attach-agent $tcp4_s
$ftp4 set type_ FTP

##################
#### MMApp New ###
##################

#Setup 1st MM_Application
set mmappnew1_s [new Application/MmAppNew]
$mmappnew1_s attach-agent $mmf3_s
$mmappnew1_s set flow_control_ true
$mmappnew1_s set frmsize_ 1000
$mmappnew1_s set random_ true
$mmappnew1_s record-mm-scale-value "all.mmappnew1.scl"
$mmappnew1_s set max_bandwidth_ 4.0mb
$mmappnew1_s set min_scale_ 0
$mmappnew1_s set max_scale_ 49

set mmappnew1_r [new Application/MmAppNew]
$mmappnew1_r attach-agent $mmf3_r
$mmf3_r record-mm-packet-arrival "all.mmappnew1.dly"

#Setup 2nd MM_Application
set mmappnew2_s [new Application/MmAppNew]
$mmappnew2_s attach-agent $mmf4_s
$mmappnew2_s set flow_control_ true
$mmappnew2_s set frmsize_ 1000
$mmappnew2_s set random_ true
$mmappnew2_s record-mm-scale-value "all.mmappnew2.scl"
$mmappnew2_s set max_bandwidth_ 4.0mb
$mmappnew2_s set min_scale_ 0
$mmappnew2_s set max_scale_ 49
```

```
set mmappnew2_r [new Application/MmAppNew]
$mmappnew2_r attach-agent $mmf4_r
$mmf4_r record-mm-packet-arrival "all.mmappnew2.dly"


##################
#Schedule events #
##################

$ns at 0.0 "$ftp1 start"
$ns at 0.0 "$ftp2 start"
$ns at 0.0 "$ftp3 start"
$ns at 0.0 "$ftp4 start"

$ns at 0.0 "$mmappnew1_s start"
$ns at 0.0 "$mmappnew2_s start"

$ns at 0.00 "$tfrc1 start"
$ns at 0.00 "$tfrc2 start"

$ns at 90.0 "$ftp1 stop"
$ns at 90.0 "$ftp2 stop"
$ns at 90.0 "$ftp3 stop"
$ns at 90.0 "$ftp4 stop"

$ns at 90.0 "$mmappnew1_s stop"
$ns at 90.0 "$mmappnew2_s stop"

$ns at 90.0 "$tfrc1 stop"
$ns at 90.0 "$tfrc2 stop"

#Call the finish procedure after 5 seconds of simulation time
$ns at 95.0 "finish"

#####################
#Run the simulation #
#####################

$ns run
```

Appendix L: get_thruput_data.c

```
/*
 * File Name: get_thruput_data.c
 * Date:      02/28/2001
 * Author(s): Jason Ingalsbe
 *            Joel Thibault
 *            Keith Barber
 *
 * Description:
 *   This script collects data about events along
 *   a given link in the simulated network. Output
 *   data includes files for percent utilization,
 *   enqueues, dequeues, drops, receives, and queue size.
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX_FID_NUM      15
#define MAX_EVT_NUM      4
#define MIN_ARGC         8
#define MAX_ARGC         9
#define MAX_FILENAME_LEN 50

#define EVT_ENQ 0  /* Array index for enqueue event */
#define EVT_DEQ 1  /* Array index for dequeue event */
#define EVT_DRP 2  /* Array index for drop event */
#define EVT_RCV 3  /* Array index for receive event */

#define ARGV_FINNAME 1
#define ARGV_FNODE   2
#define ARGV_TNODE   3
#define ARGV_MAX_QUE 4
#define ARGV_MAX_BND 5
#define ARGV_FID     6
#define ARGV_INTERV  7
#define ARGV_LABEL   8

#define AVGQUE_WEIGHT 0.002

main(int argc, char *argv[]) {

  char fEnqName[MAX_FILENAME_LEN];
  char fDeqName[MAX_FILENAME_LEN];
  char fDrpName[MAX_FILENAME_LEN];
  char fRcvName[MAX_FILENAME_LEN];
  char fQueName[MAX_FILENAME_LEN];
  char fUtlName[MAX_FILENAME_LEN];

  int    fid[MAX_FID_NUM];
  int    mBytes[MAX_EVT_NUM][MAX_FID_NUM];
  int    queue_size = 0, max_queue_size=0, last_queue_size=0;
  double avg_queue_size=0.0;
  double mThruput[MAX_EVT_NUM][MAX_FID_NUM];
  double mThruTotal[MAX_EVT_NUM];

  double cTime, mTime, mInterval;
  float time;
  int cFnode, cTnode, cFid, cBytes, fNode, tNode;
  int fCount, i, j, evt, allFlow=0, addFid;
  char *fInName, *pNext, op;
  FILE *fd1, *fdENQ, *fdDEQ, *fdDRP, *fdRCV, *fdQUE;
  char LblSpc[] = "_";
  char EnqExt[] = ".enq";
  char DeqExt[] = ".deq";
  char DrpExt[] = ".drp";
  char RcvExt[] = ".rcv";
  char QueExt[] = ".que";
  char UtlExt[] = ".utl";
  FILE *fdUTL;

  double max_bandwidth = 0.0;

  /* for dummy data */
  char tmp1[10], tmp2[10];
  float tmp3, tmp4;
  int tmp5, tmp6;

  /*****************************/
  /* Handling Command Line Args */
  /*****************************/

  for(i = 0; i < MAX_FILENAME_LEN; i++) {
    fEnqName[i] = 0;
    fDeqName[i] = 0;
    fDrpName[i] = 0;
    fRcvName[i] = 0;
    fQueName[i] = 0;
    fUtlName[i] = 0;
  }


  if(argc < MIN_ARGC || argc > MAX_ARGC) {
    fprintf(stdout, "Usage: get_thruput_data ifname ofname fnode tnode
fid event interval\n\n");
    fprintf(stdout, "       TrcFileName    = Trace File Name (ex:
out.tr)\n");
    fprintf(stdout, "       fNode          = from node of the link
(integer)\n");
    fprintf(stdout, "       tNode          = to node of the link
(integer)\n");
    fprintf(stdout, "       max_queue_size = max queue size for link
(as defined tcl script)\n");
    fprintf(stdout, "       max_bandwidth  = max bandwidth (in Mbps)
for link (as defined tcl script)\n");
    fprintf(stdout, "       fid            = flow id to monitor
(integer); \"-1\" for all flows)\n");
    fprintf(stdout, "       interval       = measurement interval in
Sec (ex: 0.1)\n");
```

```
    fprintf(stdout, "       [label]        = Optional label to be added
to output file names\n\n");
    exit(1);
  }
  else {
    fInName = argv[ARGV_FINNAME];
    fNode = atoi(argv[ARGV_FNODE]);
    tNode = atoi(argv[ARGV_TNODE]);
    max_queue_size = atoi(argv[ARGV_MAX_QUE]);
    max_bandwidth = atof(argv[ARGV_MAX_BND]);

    if(max_bandwidth <= 0.0) {
      fprintf(stdout, "Error: max_bandwidth must be greater then
0.0\n");
      exit(1);
    }

    if(argc == MAX_ARGC) {
      if(!strncpy(fEnqName, fInName, strlen(fInName) - 3)) exit(1);
      if(!strcat(fEnqName, LblSpc)) exit(1);
      if(!strcat(fEnqName, argv[ARGV_LABEL])) exit(1);
    }
    else {
      if(!strncpy(fEnqName, fInName, strlen(fInName) - 3)) exit(1);
    }
    if(!strcpy(fDeqName, fEnqName)) exit(1);
    if(!strcpy(fDrpName, fEnqName)) exit(1);
    if(!strcpy(fRcvName, fEnqName)) exit(1);
    if(!strcpy(fQueName, fEnqName)) exit(1);
    if(!strcpy(fUtlName, fEnqName)) exit(1);

    if(!strcat(fEnqName, EnqExt)) exit(1);
    if(!strcat(fDeqName, DeqExt)) exit(1);
    if(!strcat(fDrpName, DrpExt)) exit(1);
    if(!strcat(fRcvName, RcvExt)) exit(1);
    if(!strcat(fQueName, QueExt)) exit(1);
    if(!strcat(fUtlName, UtlExt)) exit(1);

    if((strtol(argv[ARGV_FID], &pNext, 10) == -1) && (*pNext == '\0'))
{
      allFlow = 1;
      fCount = 0;
    }
    else {
      fCount = 0;
      pNext = argv[ARGV_FID];
      while(1) {
      if((*pNext < 48) || (*pNext >59)) {
        fprintf(stdout, "Error: Invalid Flow ID\n");
        exit(1);
      }
      fid[fCount++] = (int)strtol(pNext, &pNext, 10);
      if(*pNext == '\0') break;
      if((*pNext == '-') && (*(pNext+1)!= '\0')) pNext = pNext+1;
      else {
        fprintf(stdout, "Error: Invalid Flow ID Format\n");
        exit(1);
      }
      }
    }
    if((mInterval = atof(argv[ARGV_INTERV])) <= 0) {
      fprintf(stdout, "Error: Measurement Interval <= \"0\"\n");
      exit(1);
    }
  }

  /******************************/
  /* Opening Input Output Files */
  /******************************/

  if((fd1 = fopen(fInName, "r")) == NULL) {
    fprintf(stdout, "Cannot open \"%s\" for read.\n", fInName);
    exit(1);
  }
  if((fdENQ = fopen(fEnqName, "w")) == NULL) {
    fprintf(stdout, "Cannot create \"%s\" for write.\n", fEnqName);
    exit(1);
  }
  if((fdDEQ = fopen(fDeqName, "w")) == NULL) {
    fprintf(stdout, "Cannot create \"%s\" for write.\n", fDeqName);
    exit(1);
  }
  if((fdDRP = fopen(fDrpName, "w")) == NULL) {
    fprintf(stdout, "Cannot create \"%s\" for write.\n", fDrpName);
    exit(1);
  }
  if((fdRCV = fopen(fRcvName, "w")) == NULL) {
    fprintf(stdout, "Cannot create \"%s\" for write.\n", fRcvName);
    exit(1);
  }
  if((fdQUE = fopen(fQueName, "w")) == NULL) {
    fprintf(stdout, "Cannot create \"%s\" for write.\n", fQueName);
    exit(1);
  }
  if((fdUTL = fopen(fUtlName, "w")) == NULL) {
    fprintf(stdout, "Cannot create \"%s\" for write.\n", fUtlName);
    exit(1);
  }

  /***************************/
  /* Get all flows from file */
  /***************************/

  if(allFlow == 1){
    while(fscanf(fd1, "%c %f %d %d %s %d %s %d %f %f %d %d\n",
             &op, &time, &cFnode, &cTnode, tmp1, &cBytes,
             tmp2, &cFid, &tmp3, &tmp4, &tmp5, &tmp6) != EOF){
      addFid = 1;

      for(i=0; i<fCount; i++){
      if(fid[i]==cFid)
        addFid = 0;
      }
      if(addFid == 1) {
      fid[fCount++] = cFid;
      }
    }
    fseek(fd1, 0, 0);   /* Reset file pointer to top */
  }

  /*****************************************/
  /* Print Configuration Info to the screen */
  /*****************************************/

  printf("\nInput File Name:       %s\n", fInName);
  printf("Enqueue File Name:     %s\n", fEnqName);
  printf("Dequeue File Name:     %s\n", fDeqName);
  printf("Drop File Name:        %s\n", fDrpName);
  printf("Receive File Name:     %s\n", fRcvName);
  printf("Queue Size File Name:  %s\n", fQueName);
  printf("Band. Util. File Name: %s\n", fUtlName);
  printf("FID(s) entered:       ");
  for(i=0; i<fCount; i++)
    printf("fid(%d) ", fid[i]);
  printf("\nFrom Node:             %d\n", fNode);
  printf("To Node:               %d\n", tNode);
  printf("Max Queue Size:        %d\n", max_queue_size);
  printf("Max Bandwidth:         %f Mb\n", max_bandwidth);
  printf("Measurment Interval:   %f\n\n", mInterval);

  /*********************/
  /* Calculate Thruput */
  /*********************/

  /* Print Headers and initialize data */

  fprintf(fdENQ, "#Time\t");
  fprintf(fdDEQ, "#Time\t");
  fprintf(fdDRP, "#Time\t");
  fprintf(fdRCV, "#Time\t");
  fprintf(fdUTL, "#Time\t");

  for(i=0; i<fCount; i++) {

    fprintf(fdENQ, "Flow%d\t", fid[i]);
    fprintf(fdDEQ, "Flow%d\t", fid[i]);
    fprintf(fdDRP, "Flow%d\t", fid[i]);
    fprintf(fdRCV, "Flow%d\t", fid[i]);
    fprintf(fdUTL, "Flow%d\t", fid[i]);

    for (j=0; j<MAX_EVT_NUM; j++) {
      mBytes[j][i] = 0;
    }
  }

  fprintf(fdENQ, "Total\n");
  fprintf(fdDEQ, "Total\n");
  fprintf(fdDRP, "Total\n");
  fprintf(fdRCV, "Total\n");
  fprintf(fdUTL, "Total\n");
```

```
  mTime = 0;

  while(fscanf(fd1, "%c %f %d %d %s %d %s %d %f %f %d %d\n",
             &op, &time, &cFnode, &cTnode, tmp1, &cBytes,
             tmp2, &cFid, &tmp3, &tmp4, &tmp5, &tmp6) != EOF) {

    /* We only care about events that occur between the fromNode and
toNode */

    if((cFnode==fNode) && (cTnode==tNode)) {

      /* Determine which type of event this is */

      if     (op=='+') evt = EVT_ENQ;
      else if(op=='-') evt = EVT_DEQ;
      else if(op=='d') evt = EVT_DRP;
      else if(op=='r') evt = EVT_RCV;

      /* If cTime <= mTime we are within the current interval so add
bytes to correct fid and event */

      cTime = (double)time;
      if(cTime <= mTime) {
      for(i=0; i<fCount; i++) {
        if(cFid==fid[i])
          mBytes[evt][i] += cBytes;
      }
      }
      else {

      /* While cTime > mTime we are beyond the current interval so
write output */

      while(cTime > mTime) {

        /* Print time interval to output files */

        fprintf(fdENQ, "%f\t", mTime);
        fprintf(fdDEQ, "%f\t", mTime);
        fprintf(fdDRP, "%f\t", mTime);
        fprintf(fdRCV, "%f\t", mTime);
        fprintf(fdUTL, "%f\t", mTime);

        /* Zero out mThruTotal for each event */

        for(i=0; i<MAX_EVT_NUM; i++)
          mThruTotal[i] = 0;

        /* Print mThruput for each flow (i) for each event (j) and add
to mThruTotal */

        for(i=0; i<fCount; i++) {

          for(j=0; j<MAX_EVT_NUM; j++) {
            /* Need to convert packet size from Bytes to Mbps */
            mThruput[j][i] =
(((double)(mBytes[j][i]))*8/mInterval)/1000000;
            mThruTotal[j] += mThruput[j][i];
          }

          fprintf(fdENQ, "%f\t", mThruput[EVT_ENQ][i]);
          fprintf(fdDEQ, "%f\t", mThruput[EVT_DEQ][i]);
          fprintf(fdDRP, "%f\t", mThruput[EVT_DRP][i]);
          fprintf(fdRCV, "%f\t", mThruput[EVT_RCV][i]);

          /* Print Flow Bandwidth Utilization for RCV event */

          fprintf(fdUTL, "%f\t", (mThruput[EVT_RCV][i])/max_bandwidth);
        }

        /* Print mThruTotal for each event */

        fprintf(fdENQ, "%f\n", mThruTotal[EVT_ENQ]);
        fprintf(fdDEQ, "%f\n", mThruTotal[EVT_DEQ]);
        fprintf(fdDRP, "%f\n", mThruTotal[EVT_DRP]);
        fprintf(fdRCV, "%f\n", mThruTotal[EVT_RCV]);

        /* Print Total Bandwidth Utilization for RCV event */

        fprintf(fdUTL, "%f\n", (mThruTotal[EVT_RCV])/max_bandwidth);

        /* Increment interval and reset mBytes */

        mTime += mInterval;
        for(i=0; i<MAX_EVT_NUM; i++) {
          for(j=0; j<fCount; j++)
            mBytes[i][j] = 0;
        }
      }

      /* We are now within the correct interval so save to the array */

      for(i=0; i<fCount; i++)
        if(cFid==fid[i])
          mBytes[evt][i] = cBytes;
      }
    }
  }

  /* Run through print process one last time to get last interval */

  fprintf(fdENQ, "%f\t", mTime);
  fprintf(fdDEQ, "%f\t", mTime);
  fprintf(fdDRP, "%f\t", mTime);
  fprintf(fdRCV, "%f\t", mTime);
  fprintf(fdUTL, "%f\t", mTime);

  for(i=0; i<MAX_EVT_NUM; i++)
    mThruTotal[i] = 0;

  for(i=0; i<fCount; i++) {
    for(j=0; j<MAX_EVT_NUM; j++) {
      /* Need to convert packet size from Bytes to Mbps */
      mThruput[j][i] = (((double)(mBytes[j][i]))*8/mInterval)/1000000;
      mThruTotal[j] += mThruput[j][i];
    }

    fprintf(fdENQ, "%f\t", mThruput[EVT_ENQ][i]);
    fprintf(fdDEQ, "%f\t", mThruput[EVT_DEQ][i]);
    fprintf(fdDRP, "%f\t", mThruput[EVT_DRP][i]);
    fprintf(fdRCV, "%f\t", mThruput[EVT_RCV][i]);

    /* Print Flow Bandwidth Utilization for RCV event */

    fprintf(fdUTL, "%f\t", (mThruput[EVT_RCV][i])/max_bandwidth);
  }

  fprintf(fdENQ, "%f\n", mThruTotal[EVT_ENQ]);
  fprintf(fdDEQ, "%f\n", mThruTotal[EVT_DEQ]);
  fprintf(fdDRP, "%f\n", mThruTotal[EVT_DRP]);
  fprintf(fdRCV, "%f\n", mThruTotal[EVT_RCV]);

  /* Print Total Bandwidth Utilization for RCV event */

  fprintf(fdUTL, "%f\n", (mThruTotal[EVT_RCV])/max_bandwidth);

  /* Close file pointers */

  fclose(fdENQ);
  fclose(fdDEQ);
  fclose(fdDRP);
  fclose(fdRCV);
  fclose(fdUTL);

  /************************/
  /* Calculate Queue Size */
  /************************/

  fprintf(fdQUE, "#Time\tQueueSize\tAvgQueueSize\n");

  fseek(fd1, 0, 0);   /* Reset file pointer to top */

  while(fscanf(fd1, "%c %f %d %d %s %d %s %d %f %f %d %d\n",
             &op, &time, &cFnode, &cTnode, tmp1, &cBytes,
             tmp2, &cFid, &tmp3, &tmp4, &tmp5, &tmp6) != EOF) {

    /* We only care about events that occur between the fromNode and
toNode */

    if((cFnode==fNode) && (cTnode==tNode)) {

      /* Determine which type of event this is */

      if     (op=='+') evt = EVT_ENQ;
      else if(op=='-') evt = EVT_DEQ;
      else if(op=='d') evt = EVT_DRP;
      else if(op=='r') evt = EVT_RCV;
      /* Adjust Queue Size */

      if (evt == EVT_ENQ) {

      /* print to the file if <= max and is different */

      if(queue_size <= max_queue_size && queue_size != last_queue_size)
{
        last_queue_size = queue_size;
        fprintf(fdQUE, "%f\t%d\t%f\n", (double)time, queue_size,
avg_queue_size);
      }
      queue_size++;

      avg_queue_size *= 1.0 - AVGQUE_WEIGHT;
      avg_queue_size += AVGQUE_WEIGHT * queue_size;
      }
      else if(evt == EVT_DEQ || evt == EVT_DRP) {
      queue_size--;
      }
    }
  }

  fclose(fdQUE);

  fclose(fd1);

}
```

Appendix M: get_delay_data.c

```
/*
 * File Name: get_delay_data.c
 * Date:      02/28/2001
 * Author(s): Jason Ingalsbe
 *            Joel Thibault
 *            Keith Barber
 *
 * Description:
 *    This script calculates delay for each packet traveling
 *    along a given link in the simulated network. The output
 *    data includes a file with arrival time, packet id, and
 *    and delay in seconds.
 */

#include <stdio.h>

#define MAX_FID_NUM      15
#define MAX_EVT_NUM      4
#define MIN_ARGC         6
#define MAX_ARGC         6
#define MAX_FILENAME_LEN 50

#define ARGV_TRCNAME 1
#define ARGV_DLYNAME 2
#define ARGV_FLOWID  3
#define ARGV_SNDNODE 4
#define ARGV_RCVNODE 5

main(int argc, char *argv[]) {

  double cTime, mTime, mInterval;
  float time;
  int cFnode, cTnode, cFid, cBytes, cPkt;
  int flowID, sndNode, rcvNode, intMaxPkt, evt, i, intRcvIdx;
  char *fTrcName, *fDlyName, op;
  FILE *fdTRC, *fdDLY;

  double *arySndTime, *aryRcvTime;
  int *aryRcvPkt, *aryRcvFlag;

  double dblTime, dblDly;
  int intPkt;

  /* for dummy data */
  char tmp1[10], tmp2[10];
  float tmp3, tmp4;
  int tmp5, tmp7;

  /*****************************/
  /* Handling Command Line Args */
  /*****************************/

  if(argc < MIN_ARGC || argc > MAX_ARGC) {
    fprintf(stdout, "Usage: get_delay_data TrcFileName DlyFileName
flowID startNode endNode\n\n");
    fprintf(stdout, "       TrcFileName    = Trace File Name (ex:
out.tr)\n");
    fprintf(stdout, "       DlyFileName    = Delay File Name (ex:
out.tcp1.dly)\n");
    fprintf(stdout, "       flowID         = ID of the Flow to get
delay data for (integer)\n");
    fprintf(stdout, "       sndNode        = Node where the sender is
located (integer)\n");
    fprintf(stdout, "       rcvNode        = Node where the receiver is
located (integer)\n");
    exit(1);
  }
  else {
    fTrcName = argv[ARGV_TRCNAME];
    fDlyName = argv[ARGV_DLYNAME];
    flowID   = atoi(argv[ARGV_FLOWID]);
    sndNode  = atoi(argv[ARGV_SNDNODE]);
    rcvNode  = atoi(argv[ARGV_RCVNODE]);
  }

  /******************************/
  /* Opening Input Output Files */
  /******************************/

  if((fdTRC = fopen(fTrcName, "r")) == NULL) {
    fprintf(stdout, "Cannot open \"%s\" for read.\n", fTrcName);
    exit(1);
  }

  if((fdDLY = fopen(fDlyName, "w")) == NULL) {
    fprintf(stdout, "Cannot create \"%s\" for write.\n", fDlyName);
    exit(1);
  }


  /*******************************************/
  /* Print Configuration Info to the screen */
  /*******************************************/

  printf("\nTRC File Name:         %s\n", fTrcName);
  printf("DLY File Name:         %s\n", fDlyName);
  printf("FlowID:                %d\n", flowID);
  printf("Sender Node:           %d\n", sndNode);
  printf("Receiver Node:         %d\n\n", rcvNode);

  /*************************************/
  /* Get number of packets sent by flow */
  /* and allocate memory for array      */
  /*************************************/

  intMaxPkt = 0;

  while(fscanf(fdTRC, "%c %f %d %d %s %d %s %d %f %f %d %d\n",
           &op, &time, &cFnode, &cTnode, tmp1, &cBytes,
           tmp2, &cFid, &tmp3, &tmp4, &cPkt, &tmp5) != EOF){
```

```
    if( cFid == flowID && cFnode == sndNode && op == '+' ) {
      if( cPkt > intMaxPkt ) {
      intMaxPkt = cPkt;
      }
    }
  }
  fseek(fdTRC, 0, 0);   /* Reset file pointer to top */

  arySndTime  = (double *)calloc(intMaxPkt + 1, sizeof(double)); //
This will be indexed by the Packet Num
  aryRcvTime  = (double *)calloc(intMaxPkt + 1, sizeof(double)); //
This will be indexed in order they appear in Trc File
  aryRcvPkt   = (int *)calloc(intMaxPkt + 1, sizeof(int));       //
This will be indexed in order they appear in Trc File
  aryRcvFlag  = (int *)calloc(intMaxPkt + 1, sizeof(int));       //
This will be indexed by the Packet Num

  for(i = 0; i < intMaxPkt; i++) {
    arySndTime[i] = 0.0;
    aryRcvTime[i] = 0.0;
    aryRcvPkt[i]  = 0;
  }

  intRcvIdx = 0;

  while(fscanf(fdTRC, "%c %f %d %d %s %d %s %d %f %f %d %d\n",
             &op, &time, &cFnode, &cTnode, tmp1, &cBytes,
             tmp2, &cFid, &tmp3, &tmp4, &cPkt, &tmp5) != EOF){

    if( cFid == flowID && cFnode == sndNode && op == '+' ) {
      arySndTime[cPkt] = time;
    }

    if( cFid == flowID && cTnode == rcvNode && op == 'r' ) {

      if( aryRcvFlag[cPkt] == 0 ) {
      aryRcvTime[intRcvIdx] = time;
      aryRcvPkt[intRcvIdx] = cPkt;
      aryRcvFlag[cPkt] = 1;
      intRcvIdx++;
      }
    }
  }

  dblTime = 0.0;
  dblDly  = 0.0;
  intPkt  = 0;

  for(i=0; i < intRcvIdx; i++) {
    dblTime = aryRcvTime[i];
    intPkt  = aryRcvPkt[i];
    dblDly  = aryRcvTime[i] - arySndTime[intPkt];

    fprintf(fdDLY,"%lf\t%d\t%lf\n", dblTime, intPkt, dblDly);
  }
  /* Close file pointers */

  fclose(fdTRC);
  fclose(fdDLY);
}
```

Appendix N: get_tcpfriendly_data.c

```
/*
 * File Name: get_tcpfriendly_data.c
 * Date:      02/28/2001
 * Author(s): Jason Ingalsbe
 *            Joel Thibault
 *            Keith Barber
 *
 * Description:
 *    This script calculates the TCP-Friendly bandwidth, as
 *    determined by the formula presented in "Promoting the
 *    Use of End-to-End Congestion Control in the Internet"
 *    by Sally Floyd and Kevin (1999), as well as the actual
 *    bandwidth for a given flow along a given link in the
 *    simulated network. The output data includes a file
 *    containing the TCP-Friendly bandwidth and actual
 *    bandwidth used by the flow.
 */

#include <string.h>
#include <stdlib.h>
#include <math.h>

#define MAX_EVT_NUM      4
#define MIN_ARGC         10
#define MAX_ARGC         10
#define MAX_FILENAME_LEN 50

#define EVT_ENQ 0  /* Array index for enqueue event */
#define EVT_DEQ 1  /* Array index for dequeue event */
#define EVT_DRP 2  /* Array index for drop event */
#define EVT_RCV 3  /* Array index for receive event */

#define ARGV_TCPNAME 1
#define ARGV_TRCNAME 2
#define ARGV_DLYNAME 3
#define ARGV_FID     4
#define ARGV_FNODE   5
#define ARGV_TNODE   6
#define ARGV_INTERV  7
#define ARGV_PKTSIZE 8
#define ARGV_MAXBAND 9

main(int argc, char *argv[]) {

  int    fid;
  int    mBytes[MAX_EVT_NUM];
  double mThruput[MAX_EVT_NUM];

  double cTime, mTime, mInterval;
  float time;
  int cFnode, cTnode, cFid, cBytes, fNode, tNode, i, evt;
  char *fTCPName, *fTrcName, *fDlyName, op;
  FILE *fdTCP, *fdTRC, *fdDLY;
  int mDlyCnt, mDlyCntSim, intNumIntervals, intIntervalIdx;
  double dropRate = 0.0, totDrp = 0.0, totOut = 0.0, totDrpSim = 0.0,
totOutSim = 0.0;
  double mDlyTotal, mDlyTotalSim, lowerTime, upperTime, avgDly;
  double cDelay, cTimeDly, dblLastTime, Tfrd;
  double *aryAvgDelay;
  double maxPktSize = 0.0, maxBandwidth = 0.0;

  /* for dummy data */
  char tmp1[10], tmp2[10];
  float tmp3, tmp4;
  int tmp5, tmp6, tmp7;

  /******************************/
  /* Handling Command Line Args */
  /******************************/

  if(argc < MIN_ARGC || argc > MAX_ARGC) {
    fprintf(stdout, "Usage: get_tcpfriendly_data TCPFileName
TrcFileName DlyFileName fid fnode tnode interval maxPktSize
maxBandwidth\n\n");
    fprintf(stdout, "       TCPFileName    = TCP Output File Name (ex:
out.tfd)\n");
    fprintf(stdout, "       TrcFileName    = Trace File Name (ex:
out.tr)\n");
    fprintf(stdout, "       DlyFileName    = Delay File Name (ex:
out.dly)\n");
    fprintf(stdout, "       fid            = Flow ID to monitor
(integer)\n");
    fprintf(stdout, "       fnode          = from node of the
bottleneck (integer)\n");
    fprintf(stdout, "       tnode          = to node  of the bottleneck
(integer)\n");
    fprintf(stdout, "       interval       = measurement interval in
Sec (ex: 0.1)\n");
    fprintf(stdout, "       maxPktSize     = Maximum packet size in
bytes (ex: 1000)\n");
    fprintf(stdout, "       maxBandwidth   = Maximum bandwidth along
the bottleneck (ex: 2.0)\n");
    exit(1);
  }
  else {
    fTCPName = argv[ARGV_TCPNAME];
    fTrcName = argv[ARGV_TRCNAME];
    fDlyName = argv[ARGV_DLYNAME];
    fid = atoi(argv[ARGV_FID]);
    fNode = atoi(argv[ARGV_FNODE]);
    tNode = atoi(argv[ARGV_TNODE]);
    maxPktSize = atoi(argv[ARGV_PKTSIZE]);
    maxBandwidth = atof(argv[ARGV_MAXBAND]);
    if((mInterval = atof(argv[ARGV_INTERV])) <= 0) {
      fprintf(stdout, "Error: Measurement Interval <= \"0\"\n");
      exit(1);
    }
  }
  /*******************************/
  /* Opening Input Output Files */
  /*******************************/

  if((fdTCP = fopen(fTCPName, "w")) == NULL) {
    fprintf(stdout, "Cannot create \"%s\" for write.\n", fTCPName);
    exit(1);
  }
  if((fdTRC = fopen(fTrcName, "r")) == NULL) {
    fprintf(stdout, "Cannot open \"%s\" for read.\n", fTrcName);
    exit(1);
  }
  if((fdDLY = fopen(fDlyName, "r")) == NULL) {
    fprintf(stdout, "Cannot open \"%s\" for read.\n", fDlyName);
    exit(1);
  }

  /*******************************************/
  /* Print Configuration Info to the screen */
  /*******************************************/

  printf("\nTCP File Name:         %s\n", fTCPName);
  printf("TRC File Name:         %s\n", fTrcName);
  printf("DLY File Name:         %s\n", fDlyName);
  printf("FID entered:           %d\n", fid);
  printf("From Node:             %d\n", fNode);
  printf("To Node:               %d\n", tNode);
  printf("Measurment Interval:   %f\n", mInterval);
  printf("TCP Max Packet Size:   %lf\n\n", maxPktSize);


/*******************************************************************/
  /* Calculate Average Delay Per Interval Based On Specified DLY File
*/

/*******************************************************************/

  maxPktSize =  maxPktSize * 8.0 / 1000000; // Convert from Bytes to Mb
  intNumIntervals = (int)(dblLastTime/mInterval + 1.0);

  aryAvgDelay = (double *)calloc(intNumIntervals, sizeof(double));

  for(i = 0; i < intNumIntervals; i++)
    aryAvgDelay[i] = 0.0;

  mDlyTotal = 0.0;
  mDlyCnt = 0;
  mDlyTotalSim = 0.0;
  mDlyCntSim = 0;
  lowerTime = 0.0;
  upperTime = 0.0;
  intIntervalIdx = 0;

  while(fscanf(fdDLY,"%lf %d %lf\n", &cTimeDly, &tmp7, &cDelay) != EOF
) {
    mDlyTotalSim += (double)cDelay;
    mDlyCntSim++;


    if( (double)cTimeDly <= upperTime ) {
      mDlyTotal += (double)cDelay;
      mDlyCnt++;
    }
    else {

      if( mDlyCnt > 0 ) {
      avgDly = mDlyTotal / mDlyCnt;
      aryAvgDelay[intIntervalIdx] = avgDly;
      }

      while( (double)cTimeDly > upperTime ) {
      intIntervalIdx++;
      lowerTime = upperTime;
      upperTime += mInterval;
      }

      mDlyTotal = (double)cDelay;
      mDlyCnt = 1;
    }
  }

  if( mDlyCnt > 0 ) {
    avgDly = mDlyTotal / mDlyCnt;
    aryAvgDelay[intIntervalIdx] = avgDly;
  }

  /************************/
  /* Calculate Bandwidths */
  /************************/


  /* Print Headers and initialize data */

  fprintf(fdTCP, "#Time\tActual Bandwidth\tTCP-Friendly Bandwidth\n");

  for (i=0; i<MAX_EVT_NUM; i++) {
    mBytes[i] = 0;
  }

  mTime = 0;
  lowerTime = 3.14;
  upperTime = 0.0;
  intIntervalIdx = 0;


  while(fscanf(fdTRC, "%c %f %d %d %s %d %s %d %f %f %d %d\n",
             &op, &time, &cFnode, &cTnode, tmp1, &cBytes,
             tmp2, &cFid, &tmp3, &tmp4, &tmp5, &tmp6) != EOF) {

    /* We only care about events that occur between the fromNode and
toNode for this flow */
```

```
    if( (cFid==fid) && (cFnode==fNode) && (cTnode==tNode) ) {

      /* Determine which type of event this is */

      if     (op=='+') evt = EVT_ENQ;
      else if(op=='-') evt = EVT_DEQ;
      else if(op=='d') evt = EVT_DRP;
      else if(op=='r') evt = EVT_RCV;

      /* If cTime <= mTime we are within the current interval so add
bytes to event */

      cTime = (double)time;

      if(cTime <= mTime) {

      /* From TRC File */

      mBytes[evt] += cBytes;
      }
      else {

      /* While cTime > mTime we are beyond the current interval so
write output */

      while(cTime > mTime) {

        /* Need to convert packet size from Bytes to Mbps */

        for(i=0; i<MAX_EVT_NUM; i++) {
          mThruput[i] = (((double)(mBytes[i]))*8/mInterval)/1000000;
        }

        /* Print time interval to output files */

        fprintf(fdTCP, "%f\t", mTime);

        /* Print Actual Bandwidth */

        fprintf(fdTCP, "%f\t", mThruput[EVT_RCV]);

        /* Print TCP-Friendly Bandwidth */

        totOut = mThruput[EVT_DRP] + mThruput[EVT_RCV];
        totDrp = mThruput[EVT_DRP];

        totOutSim += totOut; // Keep track of total RCVs throughout the
simulation
        totDrpSim += totDrp; // Keep track of total DRPs throughout the
simulation

        if( totOut > 0.0 )
          dropRate = totDrp/totOut;
        else
          dropRate = 0.0;

        avgDly = aryAvgDelay[intIntervalIdx];
        if( avgDly > 0.0 && dropRate > 0.0 )
          Tfrd = (1.5 * sqrt(2.0/3.0) * maxPktSize) / (2.0 * avgDly *
sqrt(dropRate));
        else
          Tfrd = maxBandwidth;

        fprintf(fdTCP, "%lf\n", Tfrd);

        /* Increment interval and reset mBytes */

        lowerTime = mTime;
        mTime += mInterval;
        intIntervalIdx++;
        for(i=0; i<MAX_EVT_NUM; i++)
          mBytes[i] = 0;
      }

      /* We are now within the correct interval so save to the array */

      mBytes[evt] = cBytes;

      }
    }
  }

  /*****************************************************************/
  /* Run through print process one last time to get last interval */
  /*****************************************************************/

  /* Need to convert packet size from Bytes to Mbps */

  for(i=0; i<MAX_EVT_NUM; i++) {
    mThruput[i] = (((double)(mBytes[i]))*8/mInterval)/1000000;
  }

  fprintf(fdTCP, "%f\t", mTime);

  /* Print Actual Bandwidth */

  fprintf(fdTCP, "%f\t", mThruput[EVT_RCV]);

  /* Print TCP-Friendly Bandwidth */

  totOut = mThruput[EVT_DRP] + mThruput[EVT_RCV];
  totDrp = mThruput[EVT_DRP];

  totOutSim += totOut; // Keep track of total RCVs throughout the
simulation
  totDrpSim += totDrp; // Keep track of total DRPs throughout the
simulation

  if( totOut > 0.0 )
    dropRate = totDrp/totOut;
  else
    dropRate = 0.0;
  avgDly = aryAvgDelay[intIntervalIdx];

  if( avgDly > 0.0 && dropRate > 0.0 )
    Tfrd = (1.5 * sqrt(2.0/3.0) * maxPktSize) / (2.0 * avgDly *
sqrt(dropRate));
  else
    Tfrd = maxBandwidth;

  fprintf(fdTCP, "%lf\n", Tfrd);

  /**********************************************************/
  /* Print Tfrd Average Over Whole Simulation To The Screen */
  /**********************************************************/

  if( totOutSim > 0.0 )
    dropRate = totDrpSim/totOutSim;
  else
    dropRate = 0.0;

  avgDly = mDlyTotalSim / mDlyCntSim;

  if( avgDly > 0.0 && dropRate > 0.0 )
    Tfrd = (1.5 * sqrt(2.0/3.0) * maxPktSize) / (2.0 * avgDly *
sqrt(dropRate));
  else
    Tfrd = maxBandwidth;

  printf("Tfrd Over Entire Simulation = %lf\n\n", Tfrd);

  /* Close file pointers */

  fclose(fdTCP);
  fclose(fdTRC);
  fclose(fdDLY);
  }
```