IMPLEMENTING NETWORK QUALITY OF SERVICE AT OHIO UNIVERSITY

A thesis presented to

the faculty of

the College of Engineering and Technology of Ohio University

In partial fulfillment

of the requirements for the degree

Master of Science

Brandon Saunders

November 2005
This thesis entitled

IMPLEMENTING NETWORK QUALITY OF SERVICE AT OHIO UNIVERSITY

by

BRANDON SAUNDERS

has been approved for

the School of Electrical Engineering and Computer Science

and the Russ College of Engineering and Technology by

Shawn D. Ostermann

Associate Professor of Electrical Engineering and Computer Science

Dennis Irwin

Dean, Russ College of Engineering and Technology
Network Quality of Service (QoS) implementations such as Diffserv and Intserv have focused on an end-to-end solution to providing alternative levels of service. Applications needing guaranteed bounds on qualities like throughput, delay, and packet loss were the target of such implementations. QoS can also be used to solve a different class of problem: that of network contention. There also exists a need for Quality of Service on edge links, where bandwidth is scarce. In a university setting, the network is serving the needs of both the mission of the university (teaching and research) as well as the residential life of its students, faculty, and staff. The traffic contending for limited network resources is caused by a mixture of work and pleasure. An effective Quality of Service system at a university should both prioritize traffic and apportion bandwidth to that traffic accordingly. Additionally, a complete Quality of Service solution will include network infrastructure as well as network policy.

This thesis shows that using Class Based Weighted Fair Queuing can solve a link congestion problem, and describes a measurement system developed to determine the effectiveness of the implementation.

Approved:

Shawn D. Ostermann
Associate Professor of Electrical Engineering and Computer Science
ACKNOWLEDGEMENTS

I am eternally grateful to my supporting and understanding family. Thank you Dr. Ostermann for being a fabulous professor and a wonderful mentor. I would also like to thank Terry Kelleher for creating a wonderful foundation for this thesis. I also must thank my colleges and the management at CNS for the use of the University’s network, and their diligent critique of this work.

Ninety five percent of this thesis was written on the road, Myrtle Beach four times, Washington D.C. four times, Philadelphia twice, Monterey, and an uncountable trips to Columbus, Parkersburg, and Gallipolis. This thesis would not have been possible without the people that drove me to these destinations.
# TABLE OF CONTENTS

**ABSTRACT** 3

**ACKNOWLEDGEMENTS** 4

**TABLE OF CONTENTS** 5

**LIST OF TABLES** 7

**LIST OF FIGURES** 8

## 1 INTRODUCTION
1.1 Overview ................................. 9
1.2 Background ................................. 11
  1.2.1 Differentiated Services ....................... 12
  1.2.2 Scheduling Policy ............................. 12
  1.2.3 Scheduling Discipline .......................... 13

## 2 DESIGN
2.1 Placement of the QoS Mechanism .................. 16
2.2 Queuing Discipline Selection ..................... 17
2.3 Queuing Policy .................................. 18
2.4 Monitoring Environment .......................... 19
  2.4.1 Quality of Service Monitor ..................... 20
  2.4.2 TCPtrace on Test Connections .................. 21

## 3 TESTING AND MONITORING
3.1 Data Manipulation .............................. 25
  3.1.1 TCPtrace Processing ......................... 25
  3.1.2 Database ..................................... 26
3.2 Data ........................................ 27
  3.2.1 Throughput .................................... 30
  3.2.2 Packet Loss .................................... 31
LIST OF TABLES

1.1 Breakdown of IP Protocols by Percentage of Total .......................... 12
2.1 File Transfer Size for Test Connections ....................................... 24
3.1 Connection Keys .............................................................. 27
3.2 Captured Connections ....................................................... 28
3.3 Capture Failures .............................................................. 28
3.4 Complete Connections ....................................................... 29
LIST OF FIGURES

1.1 Graph of Internet 1 Connectivity with Rate Limiting ............... 11
2.1 Ohio University Border Network ............................................ 17
2.2 The Network ................................................................. 22
3.1 Overall Throughput Graph .................................................. 30
3.2 Throughput after the Close of the Quarter ............................ 32
3.3 Overall Packet Loss .......................................................... 33
3.4 Packet Loss after the Close of the Quarter ......................... 34
3.5 Packet Loss for the Last Day ............................................... 35
3.6 Round Trip Time ............................................................... 36
3.7 Round Trip Time after the Close of the Quarter ................... 37
3.8 Round Trip Time for the Last Day ....................................... 38
Chapter 1

INTRODUCTION

Network Quality of Service (QoS) implementations such as Diffserv and Intserv have focused on an end-to-end solution to provide levels of service beyond the standard best effort. Applications needing guaranteed bounds on qualities such as throughput, delay, and packet loss were the target of such implementations. The methods developed for Diffserv and Intserv can be used to solve a different class of problem: that of network contention on a single link[29].

The current set of network monitoring tools give little indication of the expected performance of a connection within a particular link. With a new combination of existing tools it is possible to gauge the performance of a particular link.

1.1 Overview

Network bandwidth is generally most scarce at the border between networks. At these points in the network, multiple high bandwidth Local Area Network (LAN) connections are often aggregated by a router into one lower bandwidth Wide Area Network (WAN) connection. This occurs at the border between an ISP and its customers. In this case, it is possible for the LAN connections to carry more traffic to the router than can be transmitted on the WAN connection. The router can
handle this condition for short periods of time by queuing the traffic until it can be transmitted on the WAN connection. The queue space on routers is limited by available memory, so as queues overflow, packets are dropped. The delay caused by long queues and packet drops result in the condition called congestion\[14, 8\]. TCP is designed to avoid congestion by reducing its transmission rate. The reduction in rate lengthens the amount of time necessary to complete the connection. This will, for example, require web pages or file transfers to take longer.

In the case of Ohio University, much of the traffic on its Internet connections originated from the residential networks. Each of these connections were receiving their fair share of the Internet connection, but the sum of the bandwidth consumed by all of the ResNet machines were much greater than the bandwidth consumed by the Faculty and Staff because the ResNet machines were producing many more connections. This created an unfair situation that the network and the protocols that operate on it were not designed to handle.

Previous work at Ohio University has shown that rate limiting traffic on the congested Internet connection does improve the performance of other traffic\[15\]. In that research, a static rate was set for a low priority class of traffic, which was used to control peer-to-peer traffic being sourced by the residential networks. Limiting this traffic removes the congestion from the link, which allows the remaining high priority traffic to operate. The high priority traffic centers on business operations which primarily occur on weekdays from 7:30am and 6:00pm. Outside of those hours the volume of high priority traffic decreases significantly. Figure 1.1 shows the resulting traffic pattern with rate limiting the low priority traffic on the University’s Internet connection. During the day the link is congested, and at night the throughput is less than the capacity. Both of these conditions are inefficient modes of operation.

The Internet 2 QoS Working Group worked to implement Quality of Service on the Internet 2 networks in order to create a premium IP service that would maintain good
Figure 1.1: Graph of Internet 1 Connectivity with Rate Limiting - The blue line represents the volume of bandwidth being used outgoing in Kilobytes/Second. The green line represents the volume of bandwidth being used incoming in Kilobytes/Second.

performance during any network load[23]. While the service was not implemented, it was concluded the functions could be implemented successfully on smaller scales[22].

This thesis will describe and test a QoS implementation for Ohio University. The solution requires the implementation of a traffic control technique, a schema to prioritize traffic, and instrumentation to monitor the system.

1.2 Background

The Internet Protocol (IP) has no method to indicate network congestion to the source or destination of the traffic[13]. This makes IP a best effort network protocol. This property makes the IP protocol simple enough to be universally implementable, but it complicates QoS initiatives. The Internet Control Message Protocol (ICMP) does have a source quench function, but it is listed as optional in the requirements for Internet hosts[4].

Table 1.1 shows the breakdown of IP protocols on a typical day at Ohio University. The Transmission Control Protocol (TCP) is the dominate transport protocol. TCP is designed to react to and avoid congestion on the Internet, by increasing the transmission rate slowly and decreasing the transmission rate on packet loss[14].

The Internet Engineering Task Force (IETF) has developed two systems, Intserv and Diffserv, for end-to-end quality of service[26, 27]. They require the coordination
Table 1.1: Breakdown of IP Protocols by Percentage of Total - A listing of all of IP protocols seen in a half hour sample of data. TCP is the dominate IP protocol.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcp</td>
<td>94.679%</td>
</tr>
<tr>
<td>udp</td>
<td>5.301%</td>
</tr>
<tr>
<td>rtp</td>
<td>0.015%</td>
</tr>
<tr>
<td>esp</td>
<td>0.003%</td>
</tr>
<tr>
<td>gre</td>
<td>0.002%</td>
</tr>
</tbody>
</table>

and support of all ISPs and intermediate carriers to be truly effective. Implementation of these functions has been prohibitive on a global scale because of this complexity[22].

1.2.1 Differentiated Services

Diffserv contains two components. First, the Scheduling Policy, also called Forwarding Policy, determines what traffic gets through and when. Second, The Scheduling Discipline which is the algorithm for implementing the policy[3].

1.2.2 Scheduling Policy

The scheduling policy will define a rate for a connection or set of connections. In Diffserv definitions Scheduling Policy is part of a Per Hop Behavior (PHB). RFC-2475 defines a PHB as the externally observable forwarding behavior applied at a Diffserv compliant node. For example the default PHB is to forward packets marked with Diffserv code point ”000000” with best effort, and forward any packet that is marked with a diffserv code point that cannot be mapped to any other configured PHB as best effort. [5]
1.2.3 Scheduling Discipline

Qualities

The scheduling discipline is the algorithm that is implemented in the queue. The qualities of a good scheduling discipline are[3]:

- Work Conserving
- Ease of Implementation
- Fairness
- Protection
- Performance Bounds
- Admission Control

Work Conserving - Network link is utilized any time there is a packet in the queue. Intuitively, this is a desirable quality, but some queuing disciplines that guarantee latency and jitter might not be work conserving[28].

Ease of Implementation - Can be implemented in the limited memory and processing power of a router. The number of operations to implement a scheduling discipline should not be dependent on the number of connections being serviced by the router. The preferred Scheduling discipline would operate in O(1) time. Scheduling disciplines that operate in O(n) time or greater would limit scalability, as routers in the core of the network service hundreds of thousands of concurrent connections.

Fairness - An allocation of bandwidth is considered fair if the scheduling discipline follows the Max-Min Fairness criterion. Max-Min Fairness is a sharing technique when the unused bandwidth of one connection is divided equally among all of the remaining connections[28].

Protection - Keeping misbehaving connections from affecting the performance of other connections. First-Come-First-Server, for example, does not provide protection.
One misbehaving source that transmits too fast will raise the mean delay of all other connections.

**Performance Bounds** - Guarantee performance to an arbitrary number of connections.

**Admission Control** - A set of rules that are used to determine what connections receive different performance bounds[28].

**Implementations**

Custom queuing and Weighted Fair Queuing (WFQ) place traffic in one of several queues and each queue is serviced in a round-robin fashion according to configurable weights applied to each queue. This prevents any one queue from starving, as is the case with priority queuing. The weights, in essence, provide the bandwidth limitation. The difference between the two lies in how they are implemented[29].

The Weighted Fair Queuing (WFQ) algorithm was first described in Demers, Shenker and Keshav[9] and further refined under the name Generalized Processor Sharing (GPS) by Parekh and Gallagher[3, 21]. WFQ is based on the notion of max-min fairness as it relates to traffic flows. Max-min fairness orders flows by increasing demand, and allocates bandwidth to the flow with the least demand first.[3]

Fairness is a very relevant concept in relation to Internet traffic because of a phenomenon known as “lock-out” or “packet train” [3, 2, 6]. “A single connection or a few flows [can] monopolize queue space, preventing other connections from getting room in the queue”[2]. Small packets/flows will either experience greater delay relative to larger flows since they will have a longer wait in the queue, or they will be dropped because the queue is full. If there are TCP packets, their retransmission will be further delayed by TCP’s congestion control mechanism. Small packets and short flows can thus suffer at the hands of long flows[3, 2]. Min-max fairness dictates that packets/flows with the smallest resource demand are serviced first. GPS, which is more theoretical than practical, implements fairness in the following way. To be
treated fairly, each packet is put into its own queue. Each queue is serviced once per round, and during each servicing, one bit from each packet is removed from the queue. Since every packet is visited exactly once per round, each receives a fair share of the resources. The bit-by-bit processing means that the smallest packet would finish first. Because queue processors in routers have no notion of bit-by-bit processing and packets are variable length, WFQ assigns a weight to each packet that approximates the time it would take to finish processing if GPS were used, thus ensuring fairness. This weight, or time-stamp, assigned to packets is “based on their arrival rate at the router, their scheduled departure time [if GPS was used] and their length”[17]. The order in which packets are put in the departure queue is based on this time-stamp. Those packets with the smallest time-stamp get put into the departure queue first [17, 3, 21, 29].

Two other properties of WFQ make it an attractive QoS algorithm. The first, termed flow protection is related to fairness. Fairness guarantees that a flow will receive its allotted bandwidth no matter what other flows are being serviced by the router. An errant or misbehaving flow cannot disrupt another flow, as is the case in FIFO[17, 3, 21]. Second, WFQ is a work-conserving algorithm[17, 3, 21]. Work conserving means the router does not sit idle if packets are in any queues, as is the case in schemes like priority queuing[29].
Chapter 2

DESIGN

For the 2003 School year, CNS had the following design goals for the Network Quality of Service System:

- Multiple levels of service
- Levels of service can be assigned different volumes of bandwidth
- Flexible Traffic Selection
- Fit into a No Single Point of Failure network design

2.1 Placement of the QoS Mechanism

As can be seen in Figure 2.1, the border is connected at 100Mb/s to the core of the network and 32Mb/s to the primary ISP and 8Mb/s to the secondary ISP. Analysis using network utilization graphs of the traffic volumes shows that none of the connections in the network core exceed 20% utilization for the time preceding the Fall of 2003. However, the links connecting the University to its ISPs are running at 100% utilization for long periods of time.

In the Ohio University network, congestion only occurs in the outbound direction of the Internet connection, since the 100Mb/s links to the core can transfer all of the
Figure 2.1: Ohio University Border Network - Ohio University’s network core, border, two Internet 1 connections and one Internet 2 connection.

offered traffic from the ISPs. Queuing or policing in the inbound direction after the low bandwidth WAN connection would result in decreased network efficiency. It would cause traffic to traverse the ISP links more than once in the form of retransmissions.

All of the routers in this system support queue management techniques beyond First-in First-out (FIFO). Network Latency and Jitter control are not a focus of this design, so implementing queue management on the interfaces that are not congested is of little value. Implementing at these points would increase router load, network complexity and management for little return. This focuses efforts on the congested WAN interfaces to the ISP.

2.2 Queuing Discipline Selection

The queuing discipline options are limited by the offerings of the router vendor, Cisco in the case of Ohio University. Cisco supports a derivative of Weighted Fair Queuing call Class Based Weighted Fair Queuing (CBWFQ).
CBWFQ has all the properties of WFQ, plus the ability to guarantee bandwidth to designated traffic[6]. With CBWFQ, a router can be configured to add weights to the time-stamp. These weights are based on classes of traffic that a network administrator defines. Thus, certain classes of traffic may receive more bandwidth (because of how they are placed on the queue) than they would receive under WFQ. CBWFQ can ensure that Ohio University’s faculty and staff traffic receives a certain level of bandwidth without being unfair to other traffic. It also means that if the faculty and staff traffic is light (or non-existent), then the bandwidth can be used by all other traffic[29].

2.3 Queuing Policy

In the case of Ohio University, CNS’s goal is to ensure that academic and business traffic takes priority over all other traffic. The ability to prioritize this traffic is limited by Kleinrock’s Conservation Law, which states, the mean delay can be reduced for a particular connection only at the expense of the mean delay of another connection[16].

The selection of traffic for the QoS system poses the greatest challenge to the success of the QoS system. Intserv and Diffserv have concentrated on creating services where certain classes of traffic are given priority over others based upon markings in the TCP headers. Without strong authentication and trust models, these implementations are hazardous to implement because anyone could mark their packets with the highest priority. Ohio University’s QoS implementation is contained in one router, and it is not coordinated with any other organization. To simplify the configuration the router will place traffic directly into queues with Access Control Lists (ACLs).

The first characterization developed was to split Faculty and Staff from ResNet using IP address range. ResNet was dedicated 30% of the bandwidth and the remainder was left for Faculty/Staff. This scheduling policy was sufficient to allow Faculty and Staff to resume using the network. The imbalance would now be contained within the ResNet class.
Within the ResNet traffic several TCP ports stood out as consuming a lot of bandwidth. These TCP ports were linked to peer-to-peer file sharing applications. These are distributed applications that act both client and server to move resources to the edge of the network.[7] The most common application at the time was called Kazza, and was reported to be used for sharing MP3 encoded music files. These applications were placed in a separate class that were dedicated 5% of the bandwidth. It was observed that many of the peer-to-peer utilities used server port greater than 1024, and few real applications used ports in that range. As the 4th traffic class all packets with TCP or UDP ports greater that 1024 were placed in a queue with 5% of the bandwith. The resulting Cisco QoS configuration can be found in Appendix A.2

The testing of this QoS system is broken down into two parts; proof of concept in a lab situation, and monitoring of the production environment. The production monitoring will result in data that will be used to verify the design.

2.4 Monitoring Environment

Continuous monitoring is necessary to ensure that the designs created and tested in the lab operate correctly with the loads and connections of the production environment. Monitoring is also necessary to maintain the QoS classifications and rates as users and applications change. The existing Network monitoring tools give an indication about the volume and type of traffic that is traversing a network link. MRTG is a public domain graphing program that has an integrated SNMP agent. Network equipment that reflects interface counters via SNMP together with MRTG can provide graphs of link utilization[18]. Those graphs can be used to identify congested links that may need QoS applied to them.

Argus and Flow-Tools are both public domain applications that gather information about the type of traffic that traverses a link. Argus gets its data from a mirror of the traffic, and Flow-tools gathers its data from export of connection data from high end routers. The output of these programs can be used to create profiles of the traffic.
based upon values like source or destination IP address; source or destination port numbers and the volume of traffic generated by those connections[11, 1]. Flow data for congested links can help network operators identify problem machines and create and maintain QoS classifications.

TCPtrace is a public domain application that can calculate the performance characteristics of a TCP connection from a live traffic mirror or packet captures[20]. The statistics derived by TCPtrace can be used to determine changes in the QoS state, and can be used to give an indication of user experience once they have been calibrated.

2.4.1 Quality of Service Monitor

Two solutions exist for applying TCPtrace to gather the necessary statistics. The first is to apply TCPtrace to all of the known connections traversing an Internet link. The second is to apply TCPtrace to a set of test connections that are representative of the user experience.

TCPtrace on All Traffic

In this system, a mirror of the production network links are directed into a network monitoring system that captures the traffic and processes it through TCPtrace. The data for the Kelleher thesis was generated using a variation of this process in which samples of the data were used.

TCPtrace is not aware of network topology. The statistics that are generated are a resulting data is an aggregate reflection of all of the links that the TCP connection traverses. Those connections include those at Ohio University and in the Internet. Without packet captures on both ends of a particular network link, it is not possible to determine what the performance of a particular TCP connection across that link.

It takes approximately one hour for TCPtrace to process 5 minutes of traffic captured with TCPdump on an Alpha 4100 with 4 CPUs running Tru64. The packet capturing environment for the border network has also been shown to be losing data.
Using the Missed Data value in TCPtrace, lost data has been shown to be as high as 25%. This is a result of using a standard network interface card in the computer that is capturing the traffic. These cards must set an interrupt per packet. The capture computer architecture cannot keep up with this interrupt rate, and as a result, data is lost[10].

To get accurate data with this system, it will be necessary to have traffic capture points in the links between the border and core at Ohio University and in the links between the border and core at the ISP. This process presents a significant logistical, coding, and computational effort.

2.4.2 TCPtrace on Test Connections

For this thesis the following application was built to monitor the state of Quality of Service on the network. Test connections are periodically placed on the network. Tcpdump is used to capture the packets of this connection. TCPtrace is then used to calculate the TCP performance characteristics.

Environment

Netmon-FacStaff (Netmon4) is a Tru64 V5.1a system attached directly to the Ohio University Network core, and is located in the Faculty-Staff Address space. This system can be used to sink or source traffic for tests involving the Faculty-Staff priority classes. Netmon-ResNet (Netmon6) is a second interface on the Netmon4 hardware, and is addressed in the ResNet Address space. This system can be used to sink or source traffic for testing involving the ResNet priority classes.

A dual network machine like the Netmon-FacStaff, Netmon-ResNet machine, will send packets for all connections received on either interface to the default route. This results in packets with the source address of one network being transmitted in another. To reduce the impact of spoofed denial of service attacks, University Network Administrators turned on Reverse Path Forwarding in the network core. To
eliminate this conflict, it is necessary to install host specific routes for machines that need to communicate to the second network.

Netmon-Offcampus (Netmon5) is a Solaris 8 machine that is located in the OAR-net machine room. The connectivity of Netmon-Offcampus allows it to be an effective destination for test traffic. This machine is also configured to respond to the name Netmon-Offcampusb (Netmon5b).

![Network Diagram](image)

**Figure 2.2: The Network - Portion of the network that the test traffic traverses.**

The connections that are needed to test the Internet 1 link to OARnet can be generated between these netmon machines. These connections provided the data necessary to verify this thesis. For the long term monitoring of the system it will be necessary to monitor the University’s two other Internet connections, the Internet 1 link to Genuity and the Internet 2 link to OARnet.
Traffic Generation

Any system that generates a TCP connection can be used for the traffic generation. In the typical connection like HTTP, peer-to-peer, or FTP, the data is generally sourced from the machine that is receiving the connection and sinked at the machine that is opening the connection. Two of the primary traffic generation systems, TTCP and Iperf, source the data from the machines that opened the connection. This makes it reasonably difficult to control the source destination port pair that is the key of the QoS system. Another client/server system can easily be automated is a HTTP server with the curl client.

Instrumentation

TCPtrace will be used to analyze the transfers. To ensure integrity in the analysis, a packet capture of each side of each transfer will be collected. The client side packet capture can be started as part of the traffic generation process. The server side packet captures will have to run constantly and be rotated every day at midnight.

These known connections must be instrumented to collect information about transfer time, packet loss, round trip time, and jitter.

Timing and Transfer Size

It is not possible with this type of monitoring system to accurately measure computer networks without disrupting them. This is a result of the additional traffic that is generated with the test transfers. The effectiveness of the QoS testing is a balance of size and rate versus the volume of traffic used. Sampling the QoS status too often will consume too much bandwidth. Not sampling often enough will make it difficult to reproduce trends or patterns.

Fluctuations in traffic volume have a frequency of two cycles a day. Using the Nyquist theorem a reasonable starting point would be to sample a minimum of four times a day[12]. More subtle fluctuations in the traffic flow will affect the test trans-
fers. To ensure that these fluctuations are mitigated and smaller, higher frequency traffic patterns can be observed, a period of one hour will be used. Depending upon early results, this value may have to be adjusted.

Within testing windows, the transfers should be started at a random time. This ensures that smaller, less important traffic patterns, such as those generated by the periodic nature of class schedules, will be minimized.

During initial testing of the transfer scripts, the high priority traffic classes used transfers of 25Mb. In the lower priority classes 25Mb transfers were not completing or overlapping with the next transfer. This complicates the traffic capture instrumentation. A list of the initial transfer sizes for each priority class is listed in table 2.1.

The total transferred on an hourly basis is equal to 61,840Kb. At the current rate of 50 Mb/s on the OARnet I1 pipe, the total hourly transfer volume is 184,320,000 Kb/Hour. This makes the volume of the test transfers 0.03% of the total volume.

\[
\frac{61,840Kb/\text{Hour}}{184,320,000Kb/\text{Hour}} \times 100.0 = 0.03\%
\]

A complete listing of all of the configurations and scripts for the Test Traffic generation can be found in Appendix B.

<table>
<thead>
<tr>
<th>Traffic Class</th>
<th>File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kazza</td>
<td>100Kb</td>
</tr>
<tr>
<td>Student Low Port</td>
<td>5Mb</td>
</tr>
<tr>
<td>Student High Port</td>
<td>100Kb</td>
</tr>
<tr>
<td>Faculty/Staff</td>
<td>25Mb</td>
</tr>
</tbody>
</table>
Chapter 3

TESTING AND MONITORING

3.1 Data Manipulation

Once the raw data is collected, it must be ran through the following process, which returns usable data.

3.1.1 TCPtrace Processing

All of the TCPdump files were moved into one directory structure and TCPtrace is run on all of the files. TCPtrace could not handle all of the files at once, so they were broken into sections of 50 files. TCPtrace was run with the long output and the round trip time calculation flags. The output was stored in a file for later processing and piped through the qos-table-mod.awk script to convert the comma separated data into insert statements. See appendix C.2 for the qos-table-mod.awk script. This script was written for the CSV output of TCP trace version 6.6.1
3.1.2 Database

A database was selected as the easiest most automatable process to manipulate, collate, and digest the data. Copies of all of the database commands that were used are located in Appendix C.2.

Based upon the CSV TCPtrace output, a flat table layout was created. A more detailed database structure could have been developed, but it would have made analyzing the data more difficult with no immediate benefit. The database remained effective in light of this inefficient table layout.

The insert statements created by the qos-table-mod.awk script were loaded into this table from within a MySQL command window. A unique key consisting of the UNIX timestamp with the seconds removed was added to the table to simplify correlating the server side and client side packet captures from the same connection.

There were a number of connections that showed up incomplete because the client side TCPdump was terminated before it could write the final buffers. There were also occasions where connections spanned the midnight restart of the server side dumps, creating two incomplete connections. The connection will also show up as incomplete if the transfer takes longer than the timeout of two hours. All of these incomplete connections were moved to an incomplete table and were not further processed, because calculations from incomplete transactions are not accurate.

The client and server tables were then split into different tables separated by traffic classes. Separating the data into different tables simplified the remainder of the queries. The source IP, destination IP, and server port create a unique key on which the connections were separated. These key are listed in table 3.1.

New tables with a set of all known good connections are then created from the class tables for the server and client. All of the server connections are inserted, and then all of the connections in the client side are inserted if the corresponding server connection is not present.
Table 3.1: Connection Keys

<table>
<thead>
<tr>
<th>Direction</th>
<th>Class</th>
<th>Client</th>
<th>Server</th>
<th>Server port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incoming</td>
<td>Faculty/Staff</td>
<td>netmon4</td>
<td>netmon5</td>
<td>80</td>
</tr>
<tr>
<td>Outgoing</td>
<td>Faculty/Staff</td>
<td>netmon5</td>
<td>netmon4</td>
<td>80</td>
</tr>
<tr>
<td>Incoming</td>
<td>Kazza</td>
<td>netmon4</td>
<td>netmon5</td>
<td>1214</td>
</tr>
<tr>
<td>Outgoing</td>
<td>Kazza</td>
<td>netmon5</td>
<td>netmon4</td>
<td>1214</td>
</tr>
<tr>
<td>Incoming</td>
<td>Student-Low</td>
<td>netmon6</td>
<td>netmon5b</td>
<td>80</td>
</tr>
<tr>
<td>Outgoing</td>
<td>Student-Low</td>
<td>netmon5b</td>
<td>netmon6</td>
<td>80</td>
</tr>
<tr>
<td>Incoming</td>
<td>Student-high</td>
<td>netmon6</td>
<td>netmon5b</td>
<td>8080</td>
</tr>
<tr>
<td>Outgoing</td>
<td>Student-high</td>
<td>netmon5b</td>
<td>netmon6</td>
<td>8080</td>
</tr>
</tbody>
</table>

The throughput and average round trip time (RTT) data can be directly queried from the common table. Any packets that are retransmitted or received out of order is considered packet loss. Packet loss is calculated with the following formula, with values from the common table.

\[
\frac{\text{RetransmittedDataPackets} + \text{OutOfOrderPackets}}{\text{TotalPackets}} \times 100.0
\]

Lost packets appear as retransmissions on the sender side, and out of order packets on the receiver side. For example, a sender sends packets A, B, C, D. C is lost, but is later retransmitted. The receiver gets A, B, D. So a packet capture and TCPtrace calculation on the sender side lists one retransmission, and a packet capture and TCPtrace calculation on the receiver side lists one out of order packet. In both cases the packet loss is 1.

3.2 Data

The system began operation on May 30, 2003. The server side packet captures were not stabilized until June 2, 2003. On June 18th the Netmon4 server had technical difficulties with caused the system to fail. The data set will cover the time from June 2nd through June 17th. This time period traverses the school closing, which creates
Table 3.2: Captured Connections

<table>
<thead>
<tr>
<th>Class</th>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faculty/staff</td>
<td>379</td>
<td>379</td>
</tr>
<tr>
<td>Student-Low</td>
<td>383</td>
<td>382</td>
</tr>
<tr>
<td>Student-high</td>
<td>302</td>
<td>350</td>
</tr>
<tr>
<td>Kazza</td>
<td>360</td>
<td>375</td>
</tr>
</tbody>
</table>

Table 3.3: Capture Failures

<table>
<thead>
<tr>
<th>Class</th>
<th>Already Running</th>
<th>Timeout</th>
<th>Connection failed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faculty/staff</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Student-Low</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Student-high</td>
<td>81</td>
<td>47</td>
<td>1</td>
</tr>
<tr>
<td>Kazza</td>
<td>22</td>
<td>15</td>
<td>2</td>
</tr>
</tbody>
</table>

distinct periods with and without congestion. The desired scheduling policy dictates that while in congestion there should be a differentiation in the traffic classes, but while out of congestion traffic classes should receive equal treatment.

There were 384 potential data points per traffic class. The Faculty/Staff, Student-Low, and Student-high classes were all launched this number of times. One capture file was lost in the Student-high traffic class. Table 3.2 contains a list of the number of captured connections in each class.

The client scripts were designed so that only one test in each traffic class would occur at a time. Instances of this case were logged as “Already running”. There were also cases where the connection lasted longer than the two hour time limit; these errors were logged as “Timeouts”. One June 17, 2004 a network outage caused curl to report a “connection failed” error. Table 3.3 lists the number of each of the errors.

The sum of the number of completed connections, and the relevant errors should equal 384. For the Client side, “Already running” is the only relevant error. All of the traffic classes balance for the client side.
Table 3.4: Complete Connections

<table>
<thead>
<tr>
<th>Class</th>
<th>Client</th>
<th>Server</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faculty/staff</td>
<td>11</td>
<td>378</td>
<td>378</td>
</tr>
<tr>
<td>Student-Low</td>
<td>5</td>
<td>382</td>
<td>382</td>
</tr>
<tr>
<td>Student-high</td>
<td>1</td>
<td>192</td>
<td>192</td>
</tr>
<tr>
<td>Kazza</td>
<td>0</td>
<td>313</td>
<td>313</td>
</tr>
</tbody>
</table>

One the server side, there should be one server connection for every test that did not encounter an Already Running or a Connection failed. This is true for the Faculty/Staff and Student-Low traffic classes, but not for the Student-high and Kazza traffic classes.

By running TCPtrace on the server side files for these classes it was observed that the number of connections often exceeded twenty-four per day. Further investigation showed that the number of additional connections was related to the number of timed out connections. Observing the connections in TCPdump, it appears that the communication stops with a lost data packet from the server. The client continuously waits for more data and never retransmits its last acknowledgment. After a timeout, the server sends a reset packet which is also lost. After the two hour time limit the client closes the connection with a FIN packet, and server acknowledges with a reset. TCPtrace sees the connection close packets as a new connection, which effects the balance.

Some of the captured connections can not be used, because they do not contain a complete connection. A complete connection must constrain two SYN packets and two FIN packets, and is necessary to get an accurate calculation. The number of valid connections are listed in table 3.4.

The incomplete connections on the client side were caused by a bug in the client script, which terminated the TCPdump before it could write its final buffer. On the server side two incomplete connections occurred for every timeout. It is also possible
for two incomplete server connections to exist if a test transfer traversed the midnight server log rotation. There are no examples of this failure in the data set.

### 3.2.1 Throughput

Throughput data for all four traffic classes are shown in Figure 3.1.

![Overall Throughput Graph](image)

Figure 3.1: Overall Throughput Graph - Throughput of all four traffic classes from May 30, 2003 until June 17, 2003

Starting at the beginning of the data set, a weekly cycle can be observed. Faculty/staff throughput drops regularly on the 5 weekdays with minimums in the early afternoon. The two weekend minimums are noticeably smaller than the weekday minimums. This pattern also exists in the student low port traffic class. The student low port throughput remains below the faculty/staff throughput until Tuesday
June 9. Focusing on June 14, the last day of the quarter and the closing of the dorms, throughput for all of the traffic classes increase as students leave the dorms and the Internet connection becomes uncongested. This was most prominent in the low priority ResNet high port and Kazza traffic classes.

As students begin to finish classes and leave the University, the demand on the student traffic class and the load on the Internet connection as a whole began to diminish. During this time the demand for bandwidth in the faculty/staff class remained. At peek traffic times the faculty and staff offered more bandwidth than they were guaranteed. Packets in the bandwidth exceeding the guaranteed rate are subject to packet loss. After the close of the quarter, this effect is very noticeable in the Faculty/Staff traffic class. Figure 3.2 details this time period.

### 3.2.2 Packet Loss

The packet loss graph, figure 3.3, shows that during congestion, the lower priority classes received a higher packet loss rate. Loss rates decrease for the higher priority traffic. Figure 3.4 and figure 3.5 detail the time period after the close of the quarter and the last day of the data set. Once the University closed, packet loss reduces and is often zero.

### 3.2.3 Round Trip Time

The round trip time graph, figure 3.6, shows that during congestion, the lower priority classes received a higher round trip time. Round trip time decreases for the higher priority traffic. Once the University closed, all of the round trip times are reduced and became approximately equal. This is observed in figure 3.7 and figure 3.8.
Figure 3.2: Throughput after the Close of the Quarter - Throughput after the close of the quarter, June 15 through June 17.
Figure 3.3: Overall Packet Loss - Packet loss for all four traffic classes from May 31, 2003 until June 17, 2003.
Figure 3.4: Packet Loss after the Close of the Quarter - Packet loss for all four traffic classes after the school closes.
Figure 3.5: Packet Loss for the Last Day - Packet loss for all 4 traffic classes on the last day of the data set, June 17, 2003.
Figure 3.6: Round Trip Time - Round Trip time for all four traffic classes from May 31, 2003 until June 17, 2003
Figure 3.7: Round Trip Time after the Close of the Quarter - Round Trip time for all four traffic classes after the close of school.
Figure 3.8: Round Trip Time for the Last Day - Round Trip time for all four traffic classes for the last day of the data set, June 17, 2003.
Chapter 4

CONCLUSIONS

4.1 Conclusion

The goal of the queuing policy was to maintain the performance of the faculty and staff traffic in light of high traffic volumes. The data shows, that during congestion the Faculty and Staff traffic class received the most throughput, the lowest packet loss and the lowest round trip times. The remaining classifications received a decreasing throughput, higher packet loss, and higher round trip times in order of priority. Once the offered load reduced to the point that congestion was eliminated, all of the traffic classes operated equally.

The changes in the network performance have the following effects on user perception. Users outside of the University’s network, attached to an Athens area ISP, would be able to get University web pages faster, as well as download their email faster. People downloading files from machines in ResNet with peer-to-peer applications would get a very slow download.

On campus, the outbound congestion was affecting the acknowledgment stream for incoming traffic. By allowing the Faculty and Staff more outbound bandwidth and, hence, more acknowledgments, the overall user experience is improved. Web pages download faster and large files could be uploaded and downloaded faster. Students
in ResNet saw peer-to-peer applications slow down, while services like web browsing would be improved. Web pages would start to load faster, and should finish faster.

These observations were confirmed with the QoStest application that was written for this research. QoStest observed the relative performance of TCP connections in different traffic classes using four open-source software packages; the Apache web server, the Curl web client, the TCPtrace TCP performance calculator, and the MySQL database. QoStest was a very lightweight application compared to the alternatives, and was also able to focus on measuring the performance across the University’s Internet connection.

4.2 Overview

During the time that this QoS system was in place, Ohio University became active in the Internet 2 QoS working group. The QoS working group concluded, characterizing traffic on TCP header information for the purpose of degrading service can pose a particular problem with network stability. If the characterization is used to degrade the quality of the connections, the applications will eventually move to a different characterization to avoid the degraded service[24, 25].

This was proven at Ohio University during the initial spread of the Napster file sharing utility[19]. At first the traffic was characterized by a specific TCP port and the service was blocked. Users quickly began to move the TCP port on which the application operated, thereby changing the characterization and avoiding the blocks.

The end result of any characterization is uncharacterizable traffic. As applications are developed to subvert these QoS mechanisms, the programmers will eventually resort to encrypted traffic on a well known port. One likely transport is https, a service that cannot be degraded without having an adverse effect of the University’s business and educational process.
4.3 Future Work

Through the period that this QoS Mechanism was in place, a number of questions were uncovered that were beyond the scope of this thesis.

While TCP accounted for a large percentage of the traffic, other protocols exist on the network. What effect does this QoS mechanism have on these protocols?

A solution to the traffic classification problem is to implement a per user policing system. This would limit each user to a certain volume of traffic on the congested links. Traffic above and beyond the negotiated rate would be marked as Scavenger traffic, or dropped. Would this queuing policy be beneficial?

QoS mechanisms may be utilized in other portions of the University’s network. For each of these installations what would the queuing policy and queuing mechanism be?
BIBLIOGRAPHY


http://www.splintered.net/sw/flow-tools/.


http://people.ee.ethz.ch/~oetiker/webtools/mrtg/.


Appendix A

CONFIGURATION

A.1 Network Layout

At the time of the testing the 5 routers between the test clients and servers. The following traceroutes show the order of these routers.

```
$traceroute netmon5.cns.ohiou.edu
traceroute to 199.218.1.66: 1–30 hops, 38 byte packets
   1 132.235.250.252 0.977 ms 0.976 ms 0.0 ms
   2 132.235.195.29 0.977 ms 0.977 ms 0.977 ms
   3 199.18.18.25 14.6 ms 13.6 ms 10.7 ms
   4 199.18.202.1 15.6 ms 21.4 ms 21.4 ms
   5 199.18.198.8 19.5 ms 16.6 ms 15.6 ms
   6 199.218.1.66 21.4 ms 22.4 ms
```

```
$traceroute -i hme0 netmon4.cns.ohiou.edu
traceroute to 132.235.250.6, 30 hops max, 40 byte packets
   1 199.218.1.65 0.930 ms 0.714 ms 0.568 ms
   2 199.18.198.1 0.794 ms 0.607 ms 0.587 ms
   3 199.18.202.84 3.319 ms 3.486 ms 2.954 ms
   4 199.18.18.26 11.974 ms 10.992 ms 10.781 ms
   5 132.235.195.28 12.574 ms 11.245 ms 12.907 ms
   6 132.235.250.6 11.353 ms 9.995 ms 8.194 ms
```
netmon5:brando\$traceroute -i hme0:1 netmon6.cns.ohiou.edu
traceroute to 64.247.123.200, 30 hops max, 40 byte packets
  1  199.218.1.65  171.990 ms  0.624 ms  0.577 ms
  2  199.18.198.1  0.765 ms  0.574 ms  0.593 ms
  3  199.18.202.84  3.335 ms  2.984 ms  3.226 ms
  4  199.18.18.26  10.253 ms  8.580 ms  6.587 ms
  5  132.235.195.26  8.364 ms  9.782 ms 12.018 ms
  6  64.247.123.200 11.940 ms 12.561 ms 11.480 ms

[root@netmon3 saundebl]# traceroute -i eth0.340 netmon5.cns.ohiou.edu
traceroute to 199.218.1.66, 30 hops max, 38 byte packets
  1 64.247.123.254  0.913 ms  0.259 ms  0.232 ms
  2 132.235.195.25  0.360 ms  0.344 ms  0.314 ms
  3 199.18.17.249  0.434 ms  0.498 ms  0.382 ms
  4 199.18.17.17  1.966 ms  1.971 ms  1.936 ms
  5 199.18.152.49  3.167 ms 16.398 ms 13.967 ms
  6 199.18.198.8  2.748 ms  2.755 ms  2.902 ms
  7 199.218.1.66  2.837 ms  2.733 ms  2.871 ms

A.2 Router QoS Configuration

! Last configuration change at 16:44:24 est Mon Apr 21 2003
! NVRAM config last updated at 16:50:48 est Mon Apr 21 2003
!
version 12.1
!
hostname 7200a
!
class-map match-all Student
  match access-group 103
class-map match-all Kazza
  match access-group 104
class-map match-all Scavenger
  match access-group 102
!
!
policy-map Border-out
class Kazza
    bandwidth percent 5
class Scavenger
    bandwidth percent 5
class Student
    bandwidth percent 25
class class-default
    bandwidth percent 40
!
!
interface GigabitEthernet0/1
    description Border network
    bandwidth 200000
!
interface GigabitEthernet0/2
    description Link to 7200b
    bandwidth 1000000
!
interface ATM1/0
    description I2 OC-3
!
interface ATM1/0.100 point-to-point
    description Internet 2 PVC
    pvc 1/100
    abr 126000 126000
    service-policy output Border-out
!
!
interface ATM2/0
    description QARNET OC-3
!
interface ATM2/0.101 point-to-point
    description Internet 1 PVC
    pvc 1/101
    class vc class-default
    abr 66000 66000
    service-policy output Border-out
!
access−list 102 remark -----------------------
access−list 102 remark CBWFQ ACL (Scavenger)
access−list 102 remark Student High Port to High port
access−list 102 remark Do not limit this machine, it is used for auth.
access−list 102 remark for a Math program
access−list 102 deny  ip any host 128.121.14.101
access−list 102 permit tcp 64.247.64.0 0.0.63.255 gt 1023 any gt 1023
access−list 102 deny  ip any any
access−list 102 remark -----------------------
access−list 103 remark -----------------------
access−list 103 remark CBWFQ ACL (Student)
access−list 103 permit ip 64.247.64.0 0.0.63.255 any
access−list 103 deny  ip any any
access−list 103 remark -----------------------
access−list 104 remark -----------------------
access−list 104 remark CBWFQ ACL (Kazza)
access−list 104 permit tcp any eq 1214 any gt 1023
access−list 104 deny  ip any any
access−list 104 remark -----------------------

end

A.3 Apache Configuration Files

A.3.1 Netmon-FacStaff/Netmon-Resnet

The configuration of the two apache configurations.
# http.conf
#
# by Andrew Sutton
#
# setup to run standard http and SSL on different ports with different
# document roots.

#ServerType standalone
# ExtendedStatus controls whether Apache will generate "full" status
# information (ExtendedStatus On) or just basic information
# (ExtendedStatus Off) when the "server-status" handler is called. The default is Off.
# ExtendedStatus Off

# the "main" server (http) runs on port 80 (2080) and the SSL on 443 (2443)

Listen 80

# disable SSL for the main server
#SSLDisable

# turn off hostname lookups for performance
HostnameLookups off

ServerRoot /etc/httpd/
User nobody
Group nobody
ServerAdmin saundebl@ohio.edu

#logging directives
ErrorLog logs/error_log
#CustomLog logs/access_log common
PidFile logs/httpd.pid
#CustomLog logs/refer_log refer
#CustomLog logs/agent_log agent
#CustomLog logs/access_log combined

#logging levels (debug, info, notice, warn, error, crit, alert, emerg)
LogLevel warn

# custom log formats – use the string associated with a particular log
#LogFormat "%h %l %u %t "%r" %>s %b "%{Referer}i"
"%{User-Agent}i"" combined
# LogFormat "%h %l %u %t ""%r" %>s %b" common
# LogFormat "%{Referer}i -> %U" referer
# LogFormat "%{User-agent}i" agent

# scoreboard keeps track of internal status
ScoreBoardFile logs/apache_status

# TCP information (timeout, keepalive)
Timeout 300
KeepAlive on
MaxKeepAliveRequests 100
KeepAliveTimeout 15

# server pool information
MinSpareServers 5
MaxSpareServers 10
StartServers 5
MaxClients 150
MaxRequestsPerChild 30

<VirtualHost 132.235.250.5:80>
# the standard port 80 virtual host
DocumentRoot /var/www/html
</VirtualHost>

<VirtualHost 64.247.123.200:80>
# the standard port 80 virtual host
DocumentRoot /var/www/html
</VirtualHost>

### A.3.2 Netmon-Offcampus

# http.conf
#
# by Andrew Sutton
#
# setup to run standard http and SSL on different ports with different
# document roots.
ServerType standalone

# ExtendedStatus controls whether Apache will generate "full" status
# information (ExtendedStatus On) or just basic information
# (ExtendedStatus Off) when the "server-status" handler is called. The default is Off.
# ExtendedStatus Off

# the "main" server (http) runs on port 80 (2080) and the SSL on 443 (2443)

Port 80
Listen 80
Listen 8080
Listen 1214

# disable SSL for the main server
#SSLDisable

# turn off hostname lookups for performance
HostnameLookups off

# who to run the program as
User nobody
Group nobody
ServerAdmin saundeb1@ohio.edu

# the root for the conf and log directories
ServerRoot /usr/local/apache/qostest

#logging directives
ErrorLog logs/error_log
CustomLog logs/access_log common
PidFile logs/httpd.pid
CustomLog logs/refer_log refer
CustomLog logs/agent_log agent
CustomLog logs/access_log combined
#logging levels (debug, info, notice, warn, error, crit, alert, emerg)
LogLevel warn

# custom log formats — use the string associated with a particular log
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\"
\"%{User-Agent}i\"" combined
LogFormat "%h %l %u %t \"%r\" %>s %b" common
#LogFormat "%{Referer}i -> %U" referer
#LogFormat "%{User-agent}i" agent

# scoreboard keeps track of internal status
ScoreBoardFile logs/apache_status

# TCP information (timeout, keepalive)
Timeout 300
KeepAlive on
MaxKeepAliveRequests 100
KeepAliveTimeout 15

# server pool information
MinSpareServers 5
MaxSpareServers 10
StartServers 5
MaxClients 150
MaxRequestsPerChild 30

# SSL information
#SSLCACertificatePath /usr/local/apache/certs
#SSLCertificateKeyFile /usr/local/apache/certs/toddttest1-test.key
#SSLCertificateKeyFile /usr/local/apache/certs/toddttest1.key
#SSLCertificateFile /usr/local/apache/certs/toddttest1-test.cer
#SSLCertificateFile /usr/local/apache/certs/toddttest1.cer

#SSLCacheServerPath /usr/local/apache/qostest/bin/gcache
#SSLCacheServerPort /usr/local/apache/devel/logs/gcache_port
#SSLSessionCacheTimeout 15
#SSLVerifyClient 0
#SSLVerifyDepth 10
#SSLFakeBasicAuth

<VirtualHost 199.218.1.66:80>
# the standard port 80 virtual host
#ResourceConfig conf/ssl_srm.conf
DocumentRoot /usr/local/apache/qostest/htdocs
ResourceConfig conf/srm.conf
AccessConfig conf/access.conf
</VirtualHost>

<VirtualHost 199.218.1.67:80>
# the standard port 80 virtual host
#ResourceConfig conf/ssl_srm.conf
DocumentRoot /usr/local/apache/qostest/htdocs
ResourceConfig conf/srm.conf
AccessConfig conf/access.conf
</VirtualHost>

<VirtualHost 199.218.1.66:8080>
# the standard port 80 virtual host
#ResourceConfig conf/ssl_srm.conf
DocumentRoot /usr/local/apache/qostest/htdocs
ResourceConfig conf/srm.conf
AccessConfig conf/access.conf
</VirtualHost>

<VirtualHost 199.218.1.67:8080>
# the standard port 80 virtual host
#ResourceConfig conf/ssl_srm.conf
DocumentRoot /usr/local/apache/qostest/htdocs
ResourceConfig conf/srm.conf
AccessConfig conf/access.conf
</VirtualHost>

<VirtualHost 199.218.1.66:1214>
# the standard port 80 virtual host
#ResourceConfig conf/ssl_srm.conf
DocumentRoot /usr/local/apache/qostest/htdocs
</VirtualHost>
ResourceConfig  conf/srm.conf
AccessConfig    conf/access.conf
</VirtualHost>

# SSL virtual host
#<VirtualHost :9443>
#SSLEnable
#DocumentRoot   /usr/local/apache/qostest/shtdocs
#ResourceConfig conf/ssl_srm.conf
#AccessConfig   conf/ssl_access.conf
#</VirtualHost>

# add redirect of ohio.edu to www.ohiou.edu
Redirect /index.html http://www.ohiou.edu/
Appendix B

DATA GATHERING

B.1 Netmon-FacStaff/Netmon-Resnet

The tests were launch from Cron. The following is the crontab for the test user.

```
0 * * * *  /home/saundebl/qostest/bin/qos-test.staff  >>
/dev/null
0 * * * *  /home/saundebl/qostest/bin/qos-test.student-low  >>
/dev/null
0 0 * * *  /home/saundebl/qostest/bin/server-capture-restart  >>
/dev/null
30 0 * * *  /home/saundebl/qostest/bin/qos-test_cleanup
5 0 * * *  /home/saundebl/qostest/bin/qos-test_backup
```

B.1.1 Server

The following scripts were used to managed the server side of the outgoing packet captures.

```
#!/bin/bash

PATH=/bin:/usr/bin:/usr/local/bin:/usr/local/sbin

#Set file variables
DATE=`date '+%Y%m%d%H'`
CLASS="staff"
```
#Staff Variables
INTERFACE="eth0.90";
LOCAL_HOST="netmon3.cns.ohiou.edu";
REMOTE_HOST="netmon5.cns.ohiou.edu";
PORT="80";

BASE="/home/saundeb1/qostest";

export DATE CLASS INTERFACE LOCAL_HOST REMOTE_HOST PORT BASE PATH

$BASE/bin/server−capture−start;
There are 3 other scripts each corresponding to the other traffic classes.
#!/bin/bash

PATH=/bin:/usr/bin:/usr/local/bin:/usr/local/sbin

YEAR='date '+%Y'';
MONTH='date '+%m'';
DAY='date '+%d'';

LOG="$BASE/log/outgoing/$NAME/$YEAR/$MONTH/$DAY/";

test ! −d $LOG && mkdir −p $LOG

#Staff Class PCAP string
DATA="$LOG/outgoing-server−CLASS.$DATE";
VAR_FILE="$BASE/var/$CLASS.data";
PID_FILE="$BASE/var/$CLASS.pid";
SCRIPT_NAME="server−sapture−start.$CLASS";
PCAP_STRING="(src host $REMOTE_HOST and dst host $LOCAL_HOST and dst port $PORT) or (src host $LOCAL_HOST and src port $PORT and dst host $REMOTE_HOST)";

#Create Historical Record
echo "LOCAL_HOST: $LOCAL_HOST";
echo "REMOTE_HOST: $REMOTE_HOST";
echo "PORT: $PORT";
echo "PPID: $PPID";
echo "INTERFACE: $INTERFACE";
echo "DATA: $DATA";
echo "VAR_FILE: $VAR_FILE"

#Get this scripts PID
PID='ps -Af | grep "$PPID" | grep "server-capture-start" | grep -v grep | awk '{print $2}''

echo "PID: $PID";

#Start TCPdump
tcpdump -n -i $INTERFACE -w $DATA $PCAP_STRING &
TCPDUMP_PID='ps -Af | grep "$PID" | grep tcpdump | grep -v grep | awk '{print $2}''

#Write Data Files
echo "$DATA" > $VAR_FILE;
echo "$TCPDUMP_PID" > $PID_FILE;

#!/bin/bash

PATH=/bin:/usr/bin:/usr/local/bin:/usr/local/sbin

#Set file variables
BASE="/home/saundeb1/qostest";

$BASE/bin/server-capture-stop;
$BASE/bin/server-capture-start.staff;
$BASE/bin/server-capture-start.kazza;
$BASE/bin/server-capture-start.student-low;
$BASE/bin/server-capture-start.student-high;

PATH=/bin:/usr/bin:/usr/local/bin:/usr/local/sbin

#Set file variables
BASE="/home/saundeb1/qostest";
#Staff Variables
STAFF_DATA='cat $BASE/var/staff.data';
STAFF_PID='cat $BASE/var/staff.pid';

#Kazza Variables
KAZZA_DATA='cat $BASE/var/kazza.data';
KAZZA_PID='cat $BASE/var/kazza.pid';

#Student−low Variables
STUDENT_LOW_DATA='cat $BASE/var/student−low.data';
STUDENT_LOW_PID='cat $BASE/var/student−low.pid';

#Student−High Variables
STUDENT_HIGH_DATA='cat $BASE/var/student−high.data';
STUDENT_HIGH_PID='cat $BASE/var/student−high.pid';

staff () {
  echo $STAFF_PID | xargs kill;
gzip $STAFF_DATA;
}

kazza () {
  echo $KAZZA_PID | xargs kill;
gzip $KAZZA_DATA;
}

student−low () {
  echo $STUDENT_LOW_PID | xargs kill;
gzip $STUDENT_LOW_DATA;
}

student−high () {
  echo $STUDENT_HIGH_PID | xargs kill;
gzip $STUDENT_HIGH_DATA;
}

case "$1" in
  staff)
    staff
  )
B.1.2 Client

The following scripts were used to manage the client side of the incoming tests.

There are 3 other scripts each corresponding to the other traffic classes.

```bash
#!/bin/bash

PATH=/bin:/usr/bin/:/usr/local/bin:/usr/local/sbin

# Set Application Variables
INTERFACE="eth0.90";
HOST1="netmon3.cns.ohiou.edu";
HOST2="netmon5.cns.ohiou.edu";
PORT="80";
PCAP_STRING="(src host $HOST1 and dst host $HOST2 and dst port $PORT)
or (src host $HOST2 and src port $PORT and dst host $HOST1)";
CURL_STRING="-m 7200 http://netmon5.cns.ohiou.edu/lf25Mb";
NAME="staff";
BASE="/home/saundeb1/qostest";

export INTERFACE PCAP_STRING CURL_STRING NAME BASE PATH;
```
$BASE/bin/qos-test;
#!/bin/bash

PATH=/bin:/usr/bin/:/usr/local/bin:/usr/local/sbin

DELAY='perl -e 'print int (rand 3600)'';
sleep $DELAY;

#Set file variables

DATE='date '+%Y%m%d%H%M'';
SCRIPT_NAME="qos-test.$NAME'';
LONG_SCRIPT_NAME="$BASE/bin/$SCRIPT_NAME'';

YEAR='date '+%Y'';
MONTH='date '+%m'';
DAY='date '+%d'';

LOG="$BASE/log/incoming/$NAME/$YEAR/$MONTH/$DAY/'';

test ! -d $LOG && mkdir -p $LOG

DUMP_FILE="$LOG/incoming-client-$NAME-dump.$DATE'';
OUTPUT_FILE="$LOG/incoming-client-$NAME-output.$DATE'';

HTTP_FILE="$BASE/http/incoming/$SCRIPT_NAME'';

#Create Historical Record
echo "DATE: $DATE" | tee -a $OUTPUT_FILE;
echo "DELAY: $DELAY" | tee -a $OUTPUT_FILE;
echo "BASE: $BASE" >> $OUTPUT_FILE;
echo "SCRIPT_NAME: $SCRIPT_NAME" | tee -a $OUTPUT_FILE;
echo "LONG_SCRIPT_NAME: $LONG_SCRIPT_NAME" >> $OUTPUT_FILE;
echo "DUMP_FILE: $DUMP_FILE" | tee -a $OUTPUT_FILE;
echo "OUTPUT_FILE: $OUTPUT_FILE" | tee -a $OUTPUT_FILE;
echo "SCRIPT:" >> $OUTPUT_FILE;
cat $LONG_SCRIPT_NAME >> $OUTPUT_FILE;
ps -Af | grep $SCRIPT_NAME | grep -v grep;

COUNT='ps -Af | grep $SCRIPT_NAME | grep bash | grep -v grep | wc -l'
echo "COUNT: $COUNT" | tee -a $OUTPUT_FILE;

if (( $COUNT > 1 )); then
echo "$0: $SCRIPT_NAME is already running" | tee -a $OUTPUT_FILE;
exit 0
fi

echo "THIS_PID: $$" >> $OUTPUT_FILE;

#Start TCPdump
tcpdump -n -i $INTERFACE -w $DUMP_FILE $PCAP_STRING 2>&1 | \
    tee -a $OUTPUT_FILE &

echo "Parent PID"
echo "$$"; >> $OUTPUT_FILE;

echo "Process list"
ps -Af >> $OUTPUT_FILE;

echo "Process list grep for parent"
ps -Af | grep "$$" >> $OUTPUT_FILE;

#Make sure TCPdump is running before you get its PID -BAS
#Had some troubles when the system got busy.
sleep 10;

TCPDUMP_PID='ps -Af | grep "$$" | grep tcpdump | grep -v grep | \
    awk '{print $2}'';
#echo "TCPDUMP_PID: $TCPDUMP_PID" | tee -a $OUTPUT_FILE;

#Wait for TCPdump to get started
#Had some troubles when the system got busy. -BAS
sleep 10;

echo "DATA:" >> $OUTPUT_FILE;
# Run traffic

curl $CURL_STRING > /dev/null 2>&1 $OUTPUT_FILE;
echo "END_DATA:" $OUTPUT_FILE;

sleep 5;

# Stop the TCPdump processes
echo $TCPDUMP_PID | xargs kill

sleep 5;

#echo "Post Kill processes:"
#ps -afx $OUTPUT_FILE

tcptrace -n lr --csv $DUMP_FILE | gawk -f $BASE/bin/csv.awk | tee -a $OUTPUT_FILE > $HTTP_FILE

gzip $DUMP_FILE;
gzip $OUTPUT_FILE;

B.2 Netmon-Offcampus

The following crontab is used for the incoming server and the outgoing clients.
# QoS Test Stuff
0 * * * * /home/brando/qostest/bin/qos-test.staff > /dev/null 2>&1
0 * * * * /home/brando/qostest/bin/qos-test.student-low > /dev/null 2>&1
0 0 * * * /home/brando/qostest/bin/server-capture-restart > /dev/null 2>&1
30 0 * * * /home/brando/qostest/bin/qos-test_cleanup > /dev/null 2>&1
5 0 * * * /home/brando/qostest/bin/qos-test_backup > /dev/null 2>&1
B.2.1 Server

The Offcampus side differs only in configuration

```bash
#!/usr/bin/bash

PATH=/usr/bin/:/usr/local/bin:/usr/local/sbin

#Set file variables
DATE='date '+%Y%m%d%H'';
BASE="/home/brando/qostest";
CLASS="staff";

#Staff Variables
INTERFACE="hme0";
LOCAL_HOST="netmon5.cns.ohiou.edu";
REMOTE_HOST="netmon3.cns.ohiou.edu";
PORT="80";

export DATE BASE CLASS INTERFACE LOCAL_HOST REMOTE_HOST PORT

$BASE/bin/server−capture−start;
There are 3 other scripts each corresponding to the other traffic classes.

B.2.2 Client

There are 3 other scripts each corresponding to the other traffic classes.

```bash
#!/usr/bin/bash

PATH=/usr/bin/:/usr/local/bin:/usr/local/sbin

#Set Application Variables
INTERFACE="hme0";
HOST1="netmon5.cns.ohiou.edu";
HOST2="netmon3.cns.ohiou.edu";
PORT="80";
PCAP_STRING="(src host $HOST1 and dst host $HOST2 and dst port $PORT) or (src host $HOST2 and src port $PORT and dst host $HOST1)";
CURL_STRING="−m 7200 http://netmon3.cns.ohiou.edu/lf25Mb";
```
NAME="staff";
BASE="/home/brando/qostest";

export INTERFACE PCAP STRING CURL STRING NAME BASE PATH;

/home/brando/qostest/bin/qos-test;
Appendix C

DATA MANIPULATION

C.1 TCPtrace processing

All of TCPtrace files were batched into one place and the following command was
ran on that directory.

```bash
find . -name "*dump*" | \
  grep -v "/2004/" | \
  xargs -n 50 tcptrace -nlr --csv | \
  tee tcptrace.output | \
  awk -f ../scripts/qos-table-mod.awk > table_data.sql
```

TCPtrace could not handle all of the connections at once, so it was broken into
sections of 50 Connections. TCPTrace’s output was set to long output with Round
Trip Time calculations The output was stored in a file for latter processing, and
piped through the qos-table-mod.awk script to convert the comma separated data
into insert statements. This script was writing for the csv output of TCP trace
version 6.6.1

```bash
BEGIN {
  count = 1;
}

{ }

gsub(" ", ",");
```
size = split($0, a, ",");

if(a[1] == "conn_#")
    next;

if(size < 100)
    next;

printf("insert into thesis_data values( '%i', count);

    count += 1;

    for(i = 2; i <= 5; i+=1) {
        printf("", 's', a[i]);
    }

    for(i = 6; i <= 7; i+=1) {
        split(a[i], b, ".")
        printf("", FROM_UNIXTIME(%s", b[1]);
    }

    for(i = 8; i <= 37; i+=1) {
        printf("", 's', a[i]);
    }

    for(i = 38; i <= 41; i+=1) {
        split(a[i], b, "/")
        printf("", 's', 's", b[1], b[2]);
    }

    for(i = 42; i <= 71; i+=1) {
        printf("", 's', a[i]);
    }

    for(i = 72; i <= 75; i+=1) {
        if(a[i] == "NA")
            printf("", '-1');
        else
            printf("", 's', a[i]);
    }
for(i = 76; i ≤ 137; i+=1) {
    printf("", '%s', a[i]);
};

printf("\n");
}

C.2 Database

C.2.1 Create Table

This is the SQL script used to make the database tables.

create table thesis_data (  
conn MediumInt, /* 1 */  
host_a varchar(15), /* 199.218.1.67 */  
host_b varchar(15), /* 64.247.123.200 */  
port_a smallint unsigned, /* 58070 */  
port_b smallint unsigned, /* 80 */  
first_packet datetime, /* 1078988369.192377 */  
last_packet datetime, /* 1078995569.639763 */  
total_packets_a2b int unsigned, /* 1642 */  
total_packets_b2a int unsigned, /* 1910 */  
resets_sent_a2b tinyint, /* 1 */  
resets_sent_b2a tinyint, /* 0 */  
ack_pkts_sent_a2b int unsigned, /* 1640 */  
ack_pkts_sent_b2b int unsigned, /* 1910 */  
pure_acks_sent_a2b int unsigned, /* 1639 */  
pure_acks_sent_b2a int unsigned, /* 0 */  
sack_pkts_sent_a2b int unsigned, /* 0 */  
sack_pkts_sent_b2a int unsigned, /* 0 */  
max_sack_blks_ack_a2b int unsigned, /* 0 */  
max_sack_blks_ack_b2a int unsigned, /* 0 */  
unique_bytes_sent_a2b int unsigned, /* 210 */  
unique_bytes_sent_b2a int unsigned, /* 2452800 */
actual_data_pkts_a2b  int unsigned, /* 1 */
actual_data_pkts_b2a  int unsigned, /* 1907 */
actual_data_bytes_a2b int unsigned, /* 210 */
actual_data_bytes_b2a int unsigned, /* 2784220 */
rexmt_data_pkts_a2b  int unsigned, /* 0 */
rexmt_data_pkts_b2a  int unsigned, /* 229 */
rexmt_data_bytes_a2b int unsigned, /* 0 */
rexmt_data_bytes_b2a int unsigned, /* 331422 */
zwnd_probe_pkts_a2b int unsigned, /* 0 */
zwnd_probe_pkts_b2a int unsigned, /* 0 */
zwnd_probe_bytes_a2b int unsigned, /* 0 */
zwnd_probe_bytes_b2a int unsigned, /* 0 */
outoforder_pkts_a2b  int unsigned, /* 0 */
outoforder_pkts_b2a  int unsigned, /* 454 */
pushed_data_pkts_a2b int unsigned, /* 1 */
pushed_data_pkts_b2a int unsigned, /* 0 */
SYN_pkts_sent_a2b int unsigned, /* 1 -Preprocessed */
FIN_pkts_sent_a2b int unsigned, /* 0 */
SYN_pkts_sent_b2a int unsigned, /* 3 - Preprocessed */
FIN_pkts_sent_b2a int unsigned, /* 0 */
req_1323_ws_a2b  char(1), /* N -Preprocessed */
req_1323_ts_a2b  char(1), /* N */
req_1323_ws_b2a  char(1), /* N -Preprocessed */
req_1323_ts_b2a  char(1), /* N */
adv_wind_scale_a2b int unsigned, /* 0 */
adv_wind_scale_b2a int unsigned, /* 0 */
req_sack_a2b  char(1), /* Y */
req_sack_b2a  char(1), /* N */
sacks_sent_a2b int unsigned, /* 0 */
sacks_sent_b2a int unsigned, /* 0 */
urgent_data_pkts_a2b int unsigned, /* 0 */
urgent_data_pkts_b2a int unsigned, /* 0 */
urgent_data_bytes_a2b int unsigned, /* 0 */
urgent_data_bytes_b2a int unsigned, /* 0 */
mss_requested_a2b  smallint unsigned, /* 1460 */
mss_requested_b2a  smallint unsigned, /* 1460 */
max_segm_size_a2b  smallint unsigned, /* 210 */
max_segm_size_b2a  smallint unsigned, /* 1460 */
min_segm_size_a2b  smallint unsigned, /* 210 */
min_segm_size_b2a  smallint unsigned,  /*  1460 */
avg_segm_size_a2b  smallint unsigned,  /*  209 */
avg_segm_size_b2a  smallint unsigned,  /*  1459 */
max_win_adv_a2b   int unsigned,       /*  24820 */
max_win_adv_b2a   int unsigned,       /*  61440 */
min_win_adv_a2b   int unsigned,       /*  24820 */
min_win_adv_b2a   int unsigned,       /*  61440 */
zero_win_adv_a2b  int unsigned,       /*  0 */
zero_win_adv_b2a  int unsigned,       /*  0 */
avg_win_adv_a2b   int unsigned,       /*  24804 */
avg_win_adv_b2a   int unsigned,       /*  61440 */
initial_window_bytes_a2b  int unsigned, /*  210 */
initial_window_bytes_b2a  int unsigned, /*  2920 */
initial_window_pkts_a2b  int unsigned, /*  1 */
initial_window_pkts_b2a  int unsigned, /*  2 */
ttl_stream_length_a2b  int unsigned,  /* NA -Post Process */
ttl_stream_length_b2a  int unsigned,  /* NA */
missed_data_a2b  int unsigned,       /* NA */
missed_data_b2a  int unsigned,       /* NA */
truncated_data_a2b  int unsigned,     /*  196 */
truncated_data_b2a  int unsigned,     /* 2757522 */
truncated_packets_a2b  int unsigned, /*  1 */
truncated_packets_b2a  int unsigned, /* 1907 */
data_xmit_time_a2b  float,            /*  0.000 */
data_xmit_time_b2a  float,            /* 7177.306 */
idletime_max_a2b  float,              /* 29882.7 */
idletime_max_b2a  float,              /* 29882.5 */
hardware_dups_a2b  int unsigned,      /*  0 */
hardware_dups_b2a  int unsigned,      /*  0 */
throughput_a2b   int unsigned,       /*  0 */
throughput_b2a   int unsigned,       /* 341 */
RTT_samples_a2b int unsigned,        /*  2 */
RTT_samples_b2a int unsigned,        /* 951 */
RTT_min_a2b     float,               /*  3.9 */
RTT_min_b2a     float,               /*  0.1 */
RTT_max_a2b     float,               /*  4.8 */
RTT_max_b2a     float,               /* 120.0 */
RTT_avg_a2b     float,               /*  4.3 */
RTT_avg_b2a     float,               /*  27.7 */
<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTT_stdev_a2b</td>
<td>float,</td>
<td>/ 0.0 */</td>
</tr>
<tr>
<td>RTT_stdev_b2a</td>
<td>float,</td>
<td>/ 46.8 */</td>
</tr>
<tr>
<td>RTT_from_3WHS_a2b</td>
<td>float,</td>
<td>/ 3.9 */</td>
</tr>
<tr>
<td>RTT_from_3WHS_b2a</td>
<td>float,</td>
<td>/ 0.1 */</td>
</tr>
<tr>
<td>RTT_full_sz_smpls_a2b</td>
<td>int unsigned,</td>
<td>/ 1 */</td>
</tr>
<tr>
<td>RTT_full_sz_smpls_b2a</td>
<td>int unsigned,</td>
<td>/ 950 */</td>
</tr>
<tr>
<td>RTT_full_sz_min_a2b</td>
<td>float,</td>
<td>/ 4.8 */</td>
</tr>
<tr>
<td>RTT_full_sz_min_b2a</td>
<td>float,</td>
<td>/ 0.1 */</td>
</tr>
<tr>
<td>RTT_full_sz_max_a2b</td>
<td>float,</td>
<td>/ 120.0 */</td>
</tr>
<tr>
<td>RTT_full_sz_max_b2a</td>
<td>float,</td>
<td>/ 4.8 */</td>
</tr>
<tr>
<td>RTT_full_sz_avg_a2b</td>
<td>float,</td>
<td>/ 27.7 */</td>
</tr>
<tr>
<td>RTT_full_sz_avg_b2a</td>
<td>float,</td>
<td>/ 0.0 */</td>
</tr>
<tr>
<td>RTT_full_sz_stdev_a2b</td>
<td>float,</td>
<td>/ 46.8 */</td>
</tr>
<tr>
<td>post_loss_acks_a2b</td>
<td>int unsigned,</td>
<td>/ 0 */</td>
</tr>
<tr>
<td>post_loss_acks_b2a</td>
<td>int unsigned,</td>
<td>/ 227 */</td>
</tr>
<tr>
<td>ambiguous_acks_a2b</td>
<td>int unsigned,</td>
<td>/ 0 */</td>
</tr>
<tr>
<td>ambiguous_acks_b2a</td>
<td>int unsigned,</td>
<td>/ 0 */</td>
</tr>
<tr>
<td>RTT_min_last_a2b</td>
<td>float,</td>
<td>/ 0.0 */</td>
</tr>
<tr>
<td>RTT_min_last_b2a</td>
<td>float,</td>
<td>/ 0.0 */</td>
</tr>
<tr>
<td>RTT_max_last_a2b</td>
<td>float,</td>
<td>/ 0.0 */</td>
</tr>
<tr>
<td>RTT_max_last_b2a</td>
<td>float,</td>
<td>/ 0.0 */</td>
</tr>
<tr>
<td>RTT_avg_last_a2b</td>
<td>float,</td>
<td>/ 0.0 */</td>
</tr>
<tr>
<td>RTT_avg_last_b2a</td>
<td>float,</td>
<td>/ 0.0 */</td>
</tr>
<tr>
<td>RTT_sdv_last_a2b</td>
<td>float,</td>
<td>/ 0.0 */</td>
</tr>
<tr>
<td>RTT_sdv_last_b2a</td>
<td>float,</td>
<td>/ 0.0 */</td>
</tr>
<tr>
<td>segs_cum_acked_a2b</td>
<td>int unsigned,</td>
<td>/ 0 */</td>
</tr>
<tr>
<td>segs_cum_acked_b2a</td>
<td>int unsigned,</td>
<td>/ 502 */</td>
</tr>
<tr>
<td>duplicate_acks_a2b</td>
<td>int unsigned,</td>
<td>/ 2 */</td>
</tr>
<tr>
<td>duplicate_acks_b2a</td>
<td>int unsigned,</td>
<td>/ 459 */</td>
</tr>
<tr>
<td>triple_dupacks_a2b</td>
<td>int unsigned,</td>
<td>/ 0 */</td>
</tr>
<tr>
<td>triple_dupacks_b2a</td>
<td>int unsigned,</td>
<td>/ 0 */</td>
</tr>
<tr>
<td>max_retrans_a2b</td>
<td>int unsigned,</td>
<td>/ 0 */</td>
</tr>
<tr>
<td>max_retrans_b2a</td>
<td>int unsigned,</td>
<td>/ 1 */</td>
</tr>
<tr>
<td>min_retr_time_a2b</td>
<td>float,</td>
<td>/ 0.0 */</td>
</tr>
<tr>
<td>min_retr_time_b2a</td>
<td>float,</td>
<td>/ 29881.9 */</td>
</tr>
<tr>
<td>max_retr_time_a2b</td>
<td>float,</td>
<td>/ 0.0 */</td>
</tr>
<tr>
<td>max_retr_time_b2a</td>
<td>float,</td>
<td>/ 29882.7 */</td>
</tr>
<tr>
<td>avg_retr_time_a2b</td>
<td>float,</td>
<td>/ 0.0 */</td>
</tr>
</tbody>
</table>
avg_retr_time_b2a float, /* 29881.7 */
sdv_retr_time_a2b float, /* 0.0 */
sdv_retr_time_b2a float /* 0.4 */;

C.2.2 Key Update

This SQL statement is used to compute the uniq_key field.
update outgoing_client
  set uniq_key = unix_timestamp(first_packet)
    - (unix_timestamp(first_packet) \% 3600);

C.2.3 Move Incomplete Connections

The following queries are used to move the incomplete connections to a holding table.
insert into incomplete
  select *
  from outgoing_client
  where SYN_pkts_sent_a2b=0
    or FIN_pkts_sent_a2b=0
    or SYN_pkts_sent_b2a=0
    or FIN_pkts_sent_b2a=0;

delete from outgoing_client
  where SYN_pkts_sent_a2b=0
    or FIN_pkts_sent_a2b=0
    or SYN_pkts_sent_b2a=0
    or FIN_pkts_sent_b2a=0;

insert into incomplete
  select * from outgoing_server
  where SYN_pkts_sent_a2b=0
    or FIN_pkts_sent_a2b=0
    or SYN_pkts_sent_b2a=0
    or FIN_pkts_sent_b2a=0;
delete from outgoing_server
  where SYN_pkts_sent_a2b=0
  or FIN_pkts_sent_a2b=0
  or SYN_pkts_sent_b2a=0
  or FIN_pkts_sent_b2a=0;

C.2.4 Class Breakouts

The following queries are used to sort the connections into tables separated by
traffic class.
source ~/qos-test/dbscripts/qos-table.sql;
alter table thesis_data add uniq_key int unsigned;
alter table thesis_data rename to outgoing_staff_server;
insert into outgoing_staff_server
  select * from outgoing_server
  where port_b=80 and host_a='199.218.1.66';
delete from outgoing_server where port_b=80 and host_a='199.218.1.66';

source ~/qos-test/dbscripts/qos-table.sql;
alter table thesis_data add uniq_key int unsigned;
alter table thesis_data rename to outgoing_staff_client;
insert into outgoing_staff_client
  select * from outgoing_client
  where port_b=80 and host_a='199.218.1.66';
delete from outgoing_client where port_b=80 and host_a='199.218.1.66';

C.2.5 Client and Server Data Merge

insert into outgoing_staff_all select * from outgoing_staff_server;
insert into outgoing_staff_all
  select * from outgoing_staff_client
  where uniq_key NOT IN
  (SELECT distinct uniq_key from outgoing_staff_server);

```sql
```
C.3 Data Queries

These are example on the staff tables. There are corresponding queries for the other data sets.

C.3.1 Throughput

\[
\text{tee } /\text{root/qos-test/output/staff_throughput.txt}; \\
\text{select uniq_key, throughput_{a2b}, throughput_{b2a}} \\
\text{from outgoing_staff_all} \\
\text{order by uniq_key;}
\]

notee;

C.3.2 Packet Loss

\[
\text{tee } /\text{root/qos-test/output/staff_packet_loss.txt}; \\
\text{select uniq_key, rexmt_data_pkts_{a2b}, rexmt_data_pkts_{b2a},} \\
\text{outoforder_pkts_{a2b}, outoforder_pkts_{b2a},} \\
\text{total_packets_{a2b}, total_packets_{b2a},} \\
\text{((rexmt_data_pkts_{a2b} + outoforder_pkts_{a2b}) / total_packets_{a2b})} \\
\text{+ 0.0 \ast 100.0 as loss_perc_{a2b},} \\
\text{((rexmt_data_pkts_{b2a} + outoforder_pkts_{b2a}) / total_packets_{b2a})} \\
\text{+ 0.0 \ast 100.0 as loss_perc_{b2a}} \\
\text{from outgoing_staff_all} \\
\text{order by uniq_key;}
\]

notee;

C.3.3 Round Trip Time

\[
\text{tee } /\text{root/qos-test/output/staff_all_rtt.txt}; \\
\text{select uniq_key, RTT_{avg_{a2b}}, RTT_{avg_{b2a}}} \\
\text{from outgoing_staff_all} \\
\text{order by uniq_key;}
\]

notee;
C.3.4 Differential Round Trip Time

```sql
tee /root/qos-test/output/staff_diff_rtt.txt;
select server.uniq_key,
    server.RTT_avg_a2b,
    client.RTT_avg_a2b,
    (client.RTT_avg_a2b - server.RTT_avg_a2b) as rtt_avg_a2b,
    server.RTT_avg_b2a,
    client.RTT_avg_b2a,
    (server.RTT_avg_b2a - client.RTT_avg_b2a) as rtt_avg_b2a
from outgoing_staff_server as server, outgoing_staff_client as client
where server.uniq_key = client.uniq_key
order by uniq_key;
```