Evaluation of Dynamic TCP Congestion Control Scheme in the ENABLE service

by

Mahesh Akarapu

B.E. (Computer Science and Engineering)
University College of Engineering, Osmania University
Hyderabad, India, 2000

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science

_____

Professor in Charge

_____

_____

Committee Members

_____

Date Thesis Accepted

Dedicated to

My parents, brother and sister

Acknowledgements

I would like to express my sincere thanks to Dr. Victor Frost, who is also my committee chair, for his guidance throughout this thesis and for his suggestions and support during my graduate studies here in KU. I would also like to thank Dr. Joseph B. Evans for his valuable suggestions, timely replies to all my doubts during my research work. I would also like to express my thanks to Dr. Jerry James for serving on my committee.

I would also like to thank Larry Sanders for helping me solve some of the major problems during my thesis. I would also like to thank Brett Becker for his help with all the network setup work. He was always there to help me whenever I needed something to setup the local network for my thesis work. I would also like to thank all the ENABLE members for their suggestions and help.

I would like to thank Vijay Jenkel and NT Karthik for their help during the initial stages of my research work.

Last, but not the least, I would like to thank all my friends here in KU for their support and making my stay here memorable.

Abstract

The increase in the transmission speeds of the current day computer networks has increased the interest in the performance issues of TCP on these High Bandwidth Delay Product (HBDP) networks. TCP congestion control algorithms, which were originally implemented to improve the performance of TCP, have some limitations on the HBDP networks. Many of the widely used distributed applications like, FTP do not take total advantage of these high-speed networks. This is not because of the improperly designed applications, but because of the default parameters of TCP, which were designed to sacrifice optimal throughput in exchange for fair sharing of bandwidth on congested networks. To overcome this limitation of TCP, research work is conducted to properly tune the TCP parameters to improve its performance. Current approaches include using the optimal socket buffer sizes and using number of parallel streams. These parameters are different for different networks and vary over time. These techniques have to continuously adapt these parameters to suit the network conditions. This task, which requires network expertise, is difficult. The Enable service makes this task easier. Also to overcome the limitations of TCP congestion control, modifications were made to the TCP stack so that an application can turn off congestion control. Previous research has shown that this improves performance. But it is not ideal to turn off the congestion control at all times. So we need a mechanism, which determines when it is appropriate to change the congestion control state in TCP. In this thesis, we implemented a mechanism to monitor the network state and control the congestion control state. The proposed methodology is tested for a widely used application, FTP. It is shown that the performance, i.e., time to transfer a file is improved for large transfers.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Transmission Control Protocol (TCP) [20] is the most widely used transport protocol in today's computer networks. With the considerable increase in the speed of Internet backbone networks, a lot of attention is being paid to the TCP performance issues to make it better suited for such high-speed networks. Most of the current Internet applications like FTP [22] and HTTP use TCP as their transport protocol. Unfortunately these distributed applications do not take full advantage of the currently available high-speed networks. This is not because of any problems with the design of these applications, but because of the inherent limitations of TCP on these High Bandwidth Delay Product (HBDP) networks. TCP parameters have been designed to sacrifice the throughput to share the network bandwidth fairly in the face of a congested network. This makes the performance of TCP on low latency links good. But on HBDP networks a proper tuning of TCP parameters is required to take maximum advantage of the very high bandwidth available. TCP also uses a set of congestion algorithms to control the rate at which a sender transfers the data. Even though these algorithms are important for preventing the congestion in the network, they have a negative impact on the performance of TCP on the long Round Trip Time (RTT) links [7]. Lot of work is being done on tuning TCP parameters to improve the TCP throughput on such HBDP networks. But application developers require certain level of network expertise to use these mechanisms to achieve better TCP throughputs. This thesis proposes a new mechanism, which distributed application developers can use without much difficulty to maximize their TCP throughput on HBDP networks, especially for long file transfers. It also demonstrates the usefulness of this mechanism.

1.1. Overview of TCP

TCP provides a reliable, connection-oriented, in-order delivery of a stream of bytes. It is a full-duplex protocol, meaning that each TCP connection supports a pair of byte streams, one in each direction. It also includes a flow-control mechanism for each of these byte streams that allows the receiver to limit how much data the sender can transmit at a given time. TCP also supports a demultiplexing mechanism that allows multiple application programs on any given host to simultaneously carry on a conversation with their peers. In addition to the above features, TCP also implements a highly tuned congestion-control mechanism. The idea of this mechanism is to throttle how fast TCP sends data, not for the sake of keeping the sender from overrunning the receiver, but so as to keep the sender from overloading the network.

TCP uses sliding window algorithm to provide reliable, in order delivery of data. It is also used to enforce flow control between the sender and the receiver. TCP on the sending side maintains a send buffer. This buffer is used to store data that has been sent but not yet acknowledged, as well as data that has been written by the sending application, but not transmitted. On the receiving side, TCP maintains a receive buffer. This buffer holds data that arrives out of order, as well as data that is in the correct order, but that the application process has not yet had the chance to read. The way the sliding window algorithm works is as follows. First the sender transmits a segment and waits for the acknowledgement before sending any other data. Once the acknowledgement for the first segment arrives, it sends two segments and when the acknowledgement for these two segments arrive, it sends four segments and this process continues. But there is a limit to the number of segments that a sender can transmit. The receiver advertises a window size to the sender using the Advertised Window field [1] in the TCP header. The sender is then limited to having no more than a value of Advertised Window bytes of unacknowledged data at any given time. The receiver selects a suitable value for Advertised Window based on the amount of

memory allocated to the connection for the purpose of buffering data. The Effective Window used to determine this limit on the maximum number of unacknowledged bytes is calculated as follows [19]:

Effective Window = Advertised Window - (LastByteSent - LastByteAcked)

Thus the flow control is implemented in TCP. There is another window value called Congestion Window (CWND) [1], which is used to implement the congestion control, which is explained in the next section.

1.2. TCP Congestion Control

TCP congestion control [1] was introduced into the Internet in the late 1980s by Van Jacobson. Before there was congestion control in TCP, the Internet used to suffer from Congestion Collapse [5]. Congestion collapse is a network state in which the hosts would send their packets into the Internet as fast as the advertised window would allow, congestion would occur at a router, causing the packets to be dropped, and the hosts would timeout and retransmit the packets, further increasing the congestion in the network. Congestion Collapse occurs when packets arrive at a router in the network at a rate higher than it can handle.

Congestion control is implemented in TCP using four algorithms namely, Slow Start, Congestion Avoidance, Fast Retransmit and Fast Recovery [1]. TCP maintains a new state variable for each connection, called Congestion Window (CWND), which is used by the source to limit how much data is allowed to have in transit at a given time. The congestion window is congestion control's counterpart to the flow control's advertised window. The maximum number of bytes of unacknowledged data would now be the minimum of the congestion window and the advertised window. This is shown by the following equations [19].

Max. Window = MIN (Congestion Window, Advertised Window)

Effective Window = Max. Window - (LastByteSent - LastByteAcked)

Note that Max.Window replaces Advertised Window in the calculation of Effective Window. Hence a TCP source is allowed to send no faster than the slowest component, network or the destination host, could allow.

1.2.1. TCP Slow Start and Congestion Avoidance

TCP uses Slow Start and Congestion Avoidance algorithms [1] to control the amount of outstanding data injected into the network. Apart from the two variables, Congestion Window and Advertised Window defined above, TCP uses another variable called the slow start threshold (*sstresh*) to determine whether to use Slow Start or Congestion Avoidance to control the data transmission.

TCP uses Slow Start mechanism to increase the CWND at the start of a TCP connection and also when a timeout occurs because of a lost packet. During the Slow Start, the source starts out by setting the CWND value to one packet. When the ACK for this packet arrives, it increases the CWND value by 1 and then sends two packets. When the ACKs for these two segments arrive, it increases the CWND value by 2, one for each ACK and sends four segments. This in effect increases the CWND exponentially. The CWND increases this way until a loss is observed or it reaches the value of *sstresh*. TCP will be in the Slow Start state when CWND < *sstresh*. Initially *sstresh* is set to the value of receiver advertised window. Whenever a loss is observed by a timeout, the *sstresh* is set to half the value of the CWND and the CWND is set to 1 packet. The CWND will increase exponentially until it reaches the value of *sstresh* (if there are no drops), at which point it goes into the Congestion Avoidance phase.

Congestion Avoidance is the phase in which TCP increases the CWND linearly instead of exponentially as observed in the Slow Start phase. TCP will be in this phase when CWND > *sstresh*. During this phase TCP increases the CWND by one whenever the ACKs are received for all the packets in the CWND. This is a conservative approach to increase the CWND.

1.2.2. Fast Retransmit and Fast Recovery

Fast Retransmit and Fast Recovery mechanisms [1] are proposed to reduce the long idle periods of time during which the TCP on the sending host waits for a timeout to occur. Fast Retransmit is a mechanism that sometimes triggers the retransmission of a dropped packet sooner than the regular timeout mechanism. The fast retransmit mechanism does not replace regular timeouts; it just enhances that facility.

The idea of fast retransmit is straightforward. Every time a data packet arrives at the receiving side, the receiver responds with an acknowledgement, even if this sequence number has already been acknowledged. Thus, when a packet arrives out of order, TCP resends the same acknowledgement it sent the last time. This second transmission of the same acknowledgement is called a duplicate ACK. When the sending side sees a duplicate ACK, it knows that the other side must have received a packet out of order, which suggests that an earlier packet might have been lost. Since it is also possible that the earlier packet has only been delayed rather than lost, the sender waits until it sees three duplicate ACKs and then retransmits the missing packet.

The Fast Recovery mechanism removes the slow start phase that happens between when fast retransmission detects a lost packet and additive increase begins. That is when the fast retransmit mechanism signals congestion, rather than drop the congestion window all the way back to one packet and run slow start, it simply cuts

the congestion window in half and resumes additive increase. This makes TCP to use slow start only at the beginning of a connection and whenever a timeout occurs.

1.3. Motivation for the Thesis

TCP uses Congestion Window (CWND) to determine the number of packets that it can transmit at any time before it receives an acknowledgment from the receiver. The larger the CWND value, the higher the TCP throughput. TCP Slow Start and the Congestion Avoidance algorithms as described above determine the size of the Congestion Window. The maximum Congestion Window size is related to the amount of buffer space allocated to each socket by the kernel. There is a default buffer size value allocated to each socket which can be changed by the system call, *setsockopt ()* [8].

TCP performs well on the low-latency links, but on high-rate, large roundtrip time links it fails to take advantage of the high bandwidth available [7]. This can be attributed to the improper TCP parameters, including the limitations introduced in the kernel by the sizes of socket buffers. As the network throughput speeds have increased recently, the operating systems have changed the default buffer sizes from the common values of 8 kilobytes to 64 kilobytes. But these socket buffer sizes are still not enough [8] for the current high-speed networks. TCP requires very high buffer sizes to get maximum benefit from these networks. But we can not just use the maximum buffer size values for all the connections, as it wastes the operating system resources and also under certain circumstances overly large TCP buffers can have bad effect on the TCP performance. To solve this problem several approaches have been proposed, tuning of TCP buffer sizes [8,9,12] and use of parallel sockets came into picture [13]. But in order to use these mechanisms application developers need some sort of network expertise. ENABLE (Enhancement of Network Aware Applications and Bottleneck Elimination) project aims to make this task of determining the correct TCP tuning parameters easy to the application developers apart from the large

number of tasks it does. The details of the ENABLE project are given in the next section.

TCP Congestion Control algorithms are very important for the proper functioning of the networks but they can also have a negative impact on the TCP performance on the high latency links [7]. To overcome this limitation, a mechanism has been proposed, implemented and tested, which enables an application to turn off the congestion control in the TCP based on the network state. This mechanism also shows some promising results [2]. But we cannot turn off the congestion control in TCP totally because it causes congestion collapse in the network. Hence we need a mechanism by which we can be able to determine when it is appropriate to turn off CC in TCP. Enable Advisory Server (AS) tries to achieve this task of determining when to change congestion control state in TCP.

This thesis proposes a new mechanism by which an application can turn CC in TCP dynamically during the course of a TCP connection as a function of network state. This mechanism is especially aimed at improving the performance of applications with large file transfers. The advantages and disadvantages of this mechanism are also discussed as well as the results of its implementation with a popular application, FTP.

1.4. ENABLE Overview

ENABLE stands for Enhancing of Network-aware Applications and BottLeneck Elimination. This project is a Department of Energy (DOE) research project to build an adaptive monitoring infrastructure, a monitor data publishing mechanism, and monitor data analysis tools. They are developing a "Grid" service that will provide both of these capabilities. The overall goal of this Enable project is to provide manageability, reliability, and adaptability for high performance applications running over wide-area networks. A main component of Enable project is the Enable network

advice server [3]. An Enable server can be installed on any data server host (e.g.: an FTP server), and configured to monitor the network paths from that host to a set of client hosts. The Enable server monitors the state of the network continuously and can be queried by client applications to get the network tuning parameters to use. These parameters include the optimal TCP buffer size to use for a given path. These network-tuning parameters are different for different network paths and vary over time. The application become aware of the network by constantly contacting the Enable advice server and obtaining the information needed to adapt to the current network conditions.

Presently, the archival tools and the monitoring tools to store per session data in the database are being put together. The Enable service with limited capability is also implemented. The work in this thesis is a sample implementation of the actual application of the Enable service. A new capability of the Enable service is proposed, implemented and evaluated here. This capability helps the Enable service to give input on whether to use the congestion control mechanism in TCP or not based on the network conditions. Once the infrastructure is in place this mechanism can be tested in the real environment.

1.5. Organization of the Thesis

The rest of this thesis is organized into the following chapters. Chapter 2 describes the TCP Extensions for the High Bandwidth Delay Product Networks and also the background work on the TCP tuning and the work done on dynamically adjusting the state of the congestion control in TCP. Chapter 3 gives on overview of the ENABLE Architecture with an emphasis on the implementation of the Advisory Server. Chapter 4 describes the FTP in general along with its implementation and the modifications did to it to implement our mechanism. Chapter 5 shows the various test scenarios used to test our mechanism along with the tests to see its advantages and

disadvantages. It also shows the tests, which are done to see how certain factors influence the mechanism. Chapter 6 gives a summary of the accomplishments of this thesis and the possible future work that can be done in this area.

Chapter 2

Related Work

2.1. TCP Extensions for High Performance

The TCP protocol was designed to operate reliably over almost any transmission medium regardless of transmission rate, delay, corruption, duplication, or reordering of segments. The basic TCP implementation works well for the low latency networks, but it is not suitable for today's high-speed and high-delay networks. Hence several extensions were proposed [6] to the basic implementation to enhance the performance of TCP on such networks. All these extensions are implemented as TCP options so that hosts can still communicate using TCP even if they do not implement these options. Hosts that do implement the optional extensions, however, can take advantage of them.

TCP performance does not depend on the transmission rate alone; it depends on the product of the transmission rate and the round-trip delay. This "Bandwidth Delay Product"(BDP) measures the amount of data that would fill the network path. It is the buffer space required at sender and receiver to obtain maximum throughput on the TCP connection over the path, i.e., the amount of unacknowledged data that TCP must handle in order to keep the network path full. TCP performance problems arise when the bandwidth*delay product is large. We refer to an Internet path operating in this region as a "long, fat pipe" and a network containing this path as an "LFN". The three fundamental problems that arise with TCP over such LFN paths are the Window size limits, recovery from losses and the round-trip time measurements. To over come these problems, Van Jacobson, Bob Braden, Dave Borman suggested the following

extensions to TCP [6], to improve its performance over the high bandwidth-delay product networks.

- TCP Window Scale extension to enable using large windows for accommodating the large BDP values of the current high speed and high delay networks.
- TCP Timestamps for more precise estimation of the round-trip times
- Protect Against Wrapped Sequence Numbers (PAWS), for preventing the accidental reuse of TCP sequence numbers because of the sequence number wrap-around caused by high bandwidths.

We will go over the first proposed extension, which deals with the TCP window sizes.

2.1.1. TCP Large Window extension

The TCP Advertised Window field in the TCP header, which is of 16 bits limits the size of the TCP window to 2^16 = 64KBytes. Without the Large Window extensions, the maximum throughput of a TCP connection is limited by the round trip time as given in the following relation.

Max.TCP Throughput = Receiver Buffer Size/Round Trip time.

On a typical cross-country WAN link with a round trip time of 60ms, the maximum throughput of the TCP connection is limited to

Max.TCP Throughput = 64KBytes/60ms = 8.74Mbps.

This is the limit on the TCP throughput no matter what the transmission rate of the Internet path is. In order to overcome these throughput limitations, the TCP large window extensions were proposed.

Large Window extension is implemented in TCP using the TCP Window Scale option [6]. The window scale extension increases the size of the TCP advertised window to 32 bits and then uses a scale factor to carry this 32-bit value in the 16-bit window field of the TCP header. The scale factor is carried in a new TCP option, Window Scale. This option is sent only in a SYN segment and hence the window scale is fixed in each direction when a new connection is opened. The maximum receive window and hence the scale factor is determined by the size of maximum receiver buffer space. This maximum buffer space is set by default but it can be changed by a system call before a connection is opened. The Window Scale option, sent in a SYN segment, indicates the willingness of the TCP sender to do both the send and receive window scaling. It is also used to send the scale factor to apply to its receive window. Both the sender and receiver must send the Window Scale option in their SYN segments to enable window scaling in either direction. This option is sent in the initial SYN segment. It is also sent in a <SYN, ACK> segment, but only if a Window Scale option was received in the initial SYN segment. A Window Scale option in a segment without a SYN bit is ignored. When the TCP window scaling is enabled, the effective send and receive window sizes are calculated by left shifting the window sizes by scale factor times. The scale factor is limited to a value of 14 to make sure that the maximum window size is 2^30.

2.2. TCP tuning for performance enhancement

TCP uses Congestion Window to determine how many packets can be sent at one time. The larger the congestion window size, the higher the throughput. The TCP slow start and congestion avoidance algorithms [1] determine the size of this congestion window. The maximum congestion window is related to the amount of buffer space the kernel allocates for each socket [8]. For each socket, there is a default value for the buffer size, which can be changed by the program using a system library call just before opening the socket. There is also a kernel enforced maximum

buffer size, which can also be changed. The socket buffer size must be adjusted at both the sender and receiver sides.

To get maximal throughput it is always important to use the optimal TCP send and receive buffers for the link being used [8]. If the buffers are too small, the TCP congestion window will never fully open up and if the buffers are too large, the sender can overrun the receiver and the TCP window will shut down. There exists a large body of work showing that good performance can be achieved using the proper tuning techniques. However, determining the correct tuning parameters can be quite difficult, especially for users or developers who are not networking experts. There are several tools that help determine these values, but none of these include a client API and all require some level of network expertise to use. So we need a mechanism, which is easy to use, to determine the optimal buffer sizes to use for the link we are using.

The optimal buffer size is twice the *bandwidth*delay* product of the link [8].

*buffer size = 2 * bandwidth * delay = bandwidth * RTT*

where bandwidth is the bottleneck bandwidth for a particular path and RTT is the Round Trip Time on that path.

We need to have some network expertise to determine these optimal parameters and most of the distributed application developers find it difficult to deal with this network tuning. Also these optimal buffer sizes are different for different networks and vary over the time. Several research efforts are being conducted to make this task of tuning the network parameters easier for the distributed application developers so that they can concentrate on the application design instead of worrying about the network performance. These research efforts include Net100 [10], Web100 [11] among the others.

The Web100 project aims at providing the software and tools necessary for end-hosts to automatically and transparently achieve high bandwidth data rates over the high performance research networks. This project plans to achieve the TCP performance tuning transparency by embedding the appropriate diagnostics and automatic controls in the end system operating system at a level invisible to the user. Initially the software and tools are being developed for the Linux operating system, but the work is being done in a standard, open manner so that they can easily be ported to other operating systems.

The Net100 project is based on Web100 and NetLogger [32] and it proposes to develop a model for network-aware operating systems using Web100 as the means for incorporating network information and its analysis into host operating systems to improve performance. The modified operating systems will respond dynamically to network conditions and make adjustments in network transfers, sending data as fast as the network will allow.

Recently, the Linux 2.4 kernel [12] also included TCP buffer tuning algorithms. For applications that have not explicitly set the TCP send and receive buffer sizes, the kernel will attempt to grow the window sizes to match the available bandwidth (up to the receiver's default window). Autotuning is controlled by new kernel variables *net.ipv4.tcp_rmem/wmem* and the amount of kernel memory available.

2.3. Experimental modifications to TCP

Congestion Control algorithms of TCP were implemented to prevent the frequently occurring condition of congestion collapse in the Internet. It succeeded in doing that, but it has a performance bottleneck on the high-speed and high delay networks. The most important parameter on which the throughput of TCP for a particular connection depends is the product of the bandwidth of the link and the delay on that link. When

the RTT of a link is very high, the time TCP spends in the slow start phase is high and so effectively TCP doesn't yield good throughput results in that phase. For a protocol like HTTP which uses TCP as the transport protocol and which has very short duration flows, the response times because of TCP's slow start phase is disastrous over a high RTT link. For HTTP flows, the entire duration of the flow is predominated by TCP's slow start behavior and this has bad effects on the response time of the web server. Also whenever there is some random loss of a packet in a connection, the TCP Congestion Control algorithms slow down the sender by entering into the slow start phase, even though there is not much congestion in the network. However the TCP Fast Retransmission and Fast Recovery algorithms [1] minimize this effect to some extent, but still the sender will be slowed down. To overcome these limitations for certain applications, experimental modifications were made in TCP, which enable an application to turn off the congestion control mechanism in TCP from the application level [2]. This modification where there is no congestion control in TCP is referred as NOCC (No Congestion Control).

2.3.1. Implementation of NOCC in TCP

The interface that is provided for the application to turn off the congestion control mechanism in TCP is by means of a *setsockopt( )* system call. The application can turn OFF and turn ON the congestion control in TCP whenever needed. The way this NOCC is implemented in TCP is by controlling the growth of the TCP congestion window, since this is the parameter which the congestion control algorithms use to control the rate of transmission of a host. Once the congestion control is turned off, the TCP control block will be unaware of the Congestion Window parameter and the receiver's advertised window becomes the sender's limit on the number of bytes it can transmit before receiving an acknowledgement. This makes sure that the flow control is still enforced. The study with the experimental modifications to TCP stack [2] gives the details of the implementation of this NOCC mechanism and the advantages and disadvantages of using this mechanism.

The implementation of NOCC mechanism in TCP helps us to control the congestion control state in TCP, but we can not turn off the congestion control in TCP totally because this might again lead to the problem of congestion collapse in the network. To avoid this problem and at the same time take advantage of this provision in the TCP stack, we proposed to develop a new service called Enable service, which dynamically adjusts the state of the congestion control in TCP depending on the network state.

Chapter 3

ENABLE Architecture

3.1. Introduction

Enable service [3] provides its clients with the correct TCP tuning parameters for a given network path so that applications can optimize their use of the network and achieve the highest possible throughput. The Enable service works as follows: An Enable server is co-located on every system that is serving large data files to the wide-area network (e.g.: an FTP or HTTP server). The Enable service is then configured to monitor the network links to a set of client hosts from the perspective of that data server. Network monitoring results are stored in a database, and can be queried by network-aware distributed components at any time. The Enable service runs the network tests on some pre-configured time interval. The Enable service API makes it very easy for application or middleware developers to determine the optimal network parameters. To take advantage of the Enable tuning service, distributed applications must be modified to support network tuning such as the ability to set the TCP buffer size [8] or the ability to create and use multiple data streams to transfer data in parallel [13]. The network tuning parameters that the Enable service is initially concentrating on are those required by large bulk data transfer applications, such as the various "Data Grid" [30] projects.

3.2. The Enable Service

The Enable service [3] has three distinct components. First, there is the Enable Server, which keeps an up-to-date record of network parameters between itself and

other hosts. The second component is a protocol for clients to communicate with the servers. Finally, there is a simple API that makes querying the Enable Servers trivial for application developers. The primary design goal for the Enable service is the ease of installation, configuration, and use. The architecture of Enable is shown in *Figure 2* below. An Enable Server is installed on every data server host, such as an *FTP* or *HTTP* server, and that Enable server is responsible only for determining the optimal network parameters between clients and itself. The following section describes the functionality and implementation of the Enable Service.

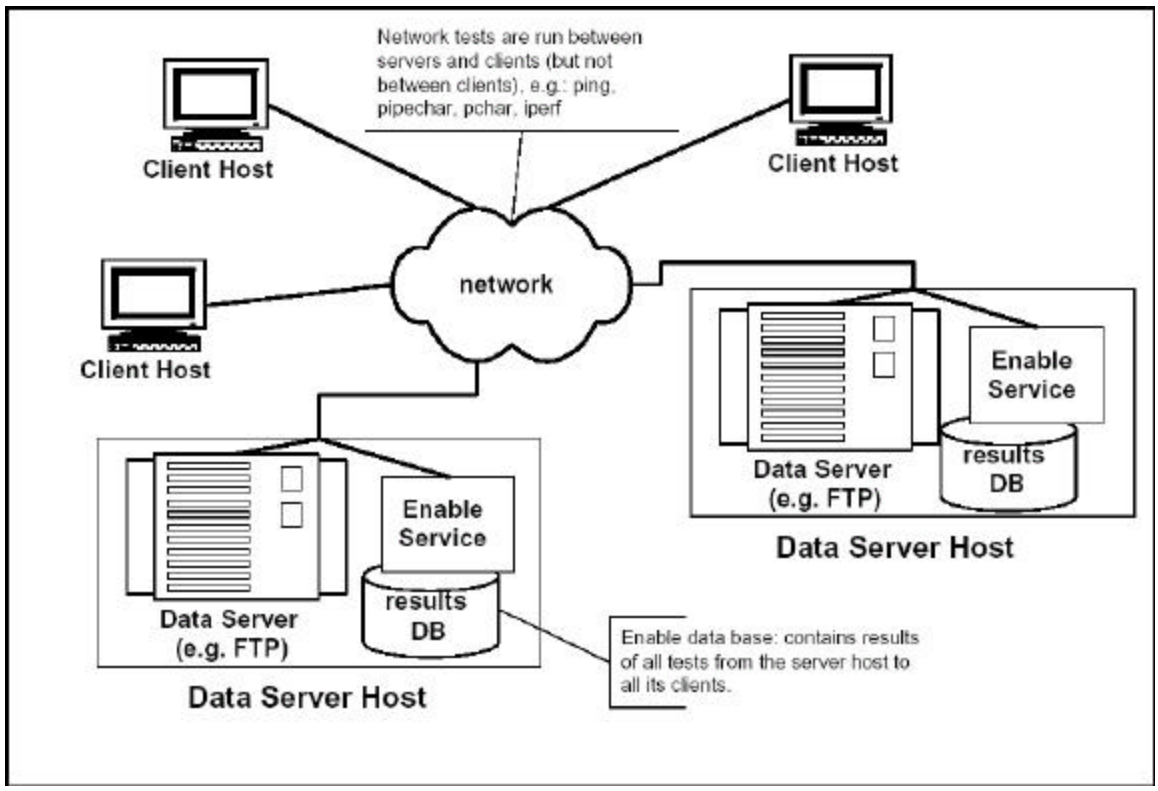

Figure 1: ENABLE Architecture, from [3]

3.2.1. Functionality of the Enable Server

The Enable Server will periodically run network monitoring tests between itself and a number of client hosts. These client hosts could be read at start-up from a configuration file, manually added using an API or command-line utility, or automatically added by monitoring log files from the data server, such as HTTP or FTP logs. The results of the network tests will be stored in a database. The selection and scheduling of tests for each client is dynamically configurable. Clients can query the Enable server, which is listening on a well-known port, for network parameters, also called "*network advice*". The protocol for doing this is XML-RPC [31], a standard XML-based protocol that performs remote procedure calls over HTTP. The standard protocol is used so that interfacing with Enable is made easier without using the Enable API or libraries.

Here is a sample API that clients can use to query the Enable Server. For example:

```
tcp_buffer_size = EnableGetBufferSize(ftp_hostname)
```

returns the optimal buffer size between itself and the *FTP* server host, and:

```
net_info = EnableGetNetInfo(ftp_hostname)
```

returns the result of all network tests for that network path.

Currently the Enable server supported network tests are *ping*, *pipechar* [25], *pchar* [33], and *Iperf* [17], but only *ping* and *pipechar* are run by default.

3.2.2. Implementation of the Enable Service

The ENABLE framework is not completely in place and our work is based on a simple look-alike of the Enable service. For this thesis, the Enable server code is written totally independent to the actual Enable server, but it includes all the concepts of the actual enable server. It maintains a database of the network conditions, which include the available bandwidth on the network path to the client at different times of a day. It uses this data to advise the client about the tuning parameters. The data collected is the available bandwidth variation data on a router interface. The Enable server does not run any network tests to the set of client hosts. It instead bases its advice on the traffic data from the router. Also modifications are made to the ProFTPD daemon [15], which is one of the target applications when the actual ENABLE architecture is in place. Whenever an FTP client tries to download a file from the FTP server, the FTP server contacts the Advisory Server (AS, also called the Enable Server) for advice on the network parameters (TCP congestion control state (CC State) input in this case) to use. The advisory server gives input on the CC state of TCP and also the time at which it has to be contacted again. Based on the input it receives from the AS, the ProFTPD daemon turns the CC State in TCP and begins transferring the requested file to the client host. When the next advise time, as suggested by the AS nears, the FTP server contacts the AS again to see if there is any change in the state of network and hence in the network parameters to use for the client host. If so it changes the network parameters (TCP CC State). This process continues till the transfer is complete.

The way our work differs from the actual ENABLE framework is that the Enable server does not run any network tests to determine the state of the network. Instead it determines the current state of the network based on the traffic data that is collected before hand. The other change is, since the data is collected only for a single router, it can give useful input only to hosts on the router path. Also the inputs that the Enable

server gives and the way those inputs are derived are different from that of the actual Enable server. But with little changes we can make this very similar to the actual ENABLE architecture.

3.3. Algorithm used to implement the Advisory Server

The purpose of the Advisory Server (AS) is to give the applications the inputs on the network parameters to use. To achieve this, we model the changes in available bandwidth on a network path and use the result to give inputs on the network tuning parameters. The following description is based on [4]. We use FTP as the target application, which uses this Advisory Server (AS) to reduce the transfer time of large files. One of the parameters the AS gives is the optimal TCP buffer size to use for a connection. The optimal TCP buffer size can be calculated by the following formula [8]:

Optimal TCP buffer = RTT * (bandwidth of bottleneck link)

Another parameter that we deal with is, if there is no congestion in the link, we can turn off congestion control in TCP. So the AS gives the input of whether to use CC in TCP or not. Turning CC OFF can improve the performance of a TCP flow at the expense of the background flows [2]. For relatively short file transfers, bandwidth can be assumed to be constant; but for very large files, we need to build a model to predict when to adjust these network parameters. We also monitor the Congestion State of the link from server to client. The architecture of the system considered is shown in *Figure 2*.

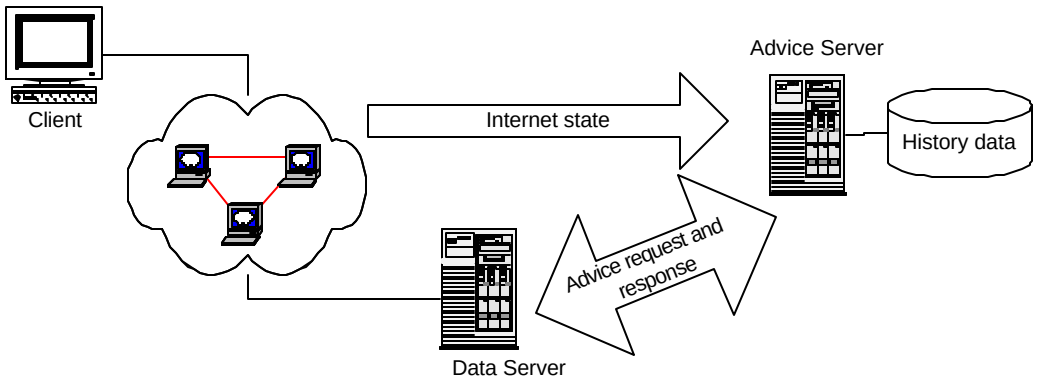

Figure 2: System architecture, from [4]

The following is from [4]. Here we model the changes in available bandwidth on a network path. In this model, we regard the available bandwidth as being composed of two parts: one is deterministic; another is random to reflect the burstiness of network traffic, it can be taken as noise. For the first part, a linear regression method is used to model the traffic for short time intervals; that is, at any time we assume a linear relationship between time and available bandwidth over a short period of time. For example, when the advice server receives request at 2:00 am, it calculates the mean available bandwidth and bandwidth rate of change at 2:00 am according to historical data.

Assume the relationship between available bandwidth and time is linear,

$$b = ?_0 + ?_1 ?$$

b represents available bandwidth

$?$ represents time relative to the request time

$?_0$ will be the available bandwidth at the time of the request, $?_1$ will be rate of change of the available bandwidth, it may be negative.

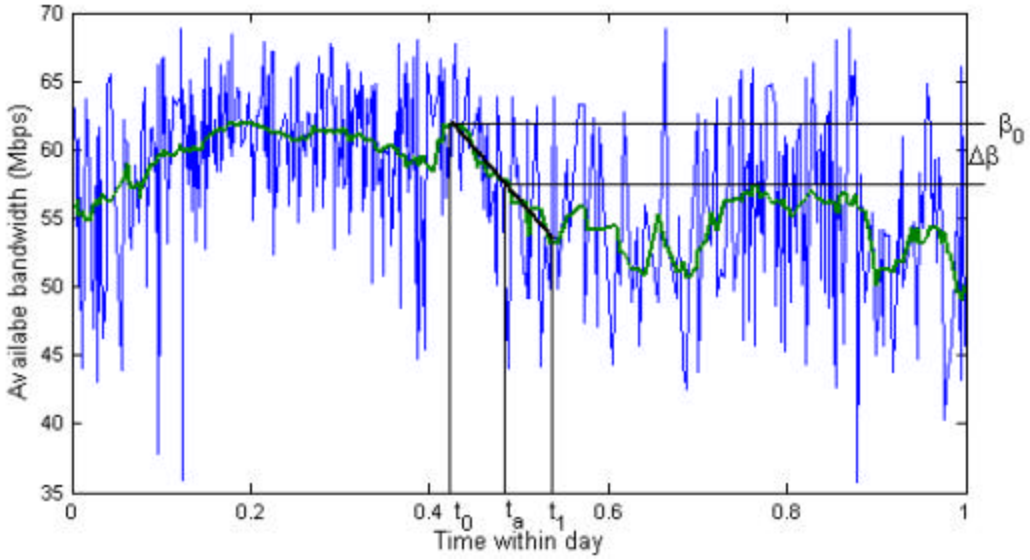

Figure 3: Illustration of linear model, from [4]

In *Figure 3*, the advice request arrives at time t_0. The linear regression is done for data between t_0 and t_1 ($t_1 - t_0$ is a configured constant). β_0 will be the available bandwidth at t_0, β_1 is the slope, representing the rate of change in available bandwidth. If we require that the change in available bandwidth will not exceed $\Delta\beta$, then the next advice time will be t_a.

The database in the advice server will contain a series of observed pairs $(b_i, \beta_i), i = 1, \ldots, n$, from which the estimators of the coefficients β_0, β_1 can be calculated, that is [4],

$$\hat{\beta}_1 = \frac{n \sum b_i \beta_i - (\sum b_i)(\sum \beta_i)}{n \sum \beta_i^2 - (\sum \beta_i)^2}$$

$$\hat{\beta}_0 = \frac{\sum b_i - \hat{\beta}_1 \sum \beta_i}{n}$$

The measurement contains noise so that the model becomes

$$b_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

We assume that:

- ε_i is a Gaussian random variable
- Mean value of ε_i is 0, i.e., $E(\varepsilon_i) = 0$
- Variance of ε_i is constant. (Therefore, $\varepsilon_i \sim N(0, \sigma^2)$)
- Correlation of ε_i and ε_j ($i \neq j$) is 0
- Correlation of ε_i and b_i is 0

Under these assumptions:

- $\hat{\beta}_1$ is an unbiased estimator of β_1
- $\hat{\beta}_0$ is an unbiased estimator of β_0
- The goodness of fitness is measured by

$$R^2 = 1 - \frac{\sum (b_i - (\hat{\beta}_0 + \hat{\beta}_1 x_i))^2}{\sum b_i^2 - (\sum b_i)^2 / n}$$

3.3.1. Implementation of Advice Server

The network data is present at the Advice Server host in a database as a sequence of records of (ip, t, b, c). Where ip is the IP address of each hop along the path to the client (only one hop for our case), t is test time, b is measured available bandwidth at the time t, c is a Boolean that indicates whether congestion is occurring (TRUE) or not (FALSE) at this time t. This value is determined based on the available bandwidth at this time. If the available bandwidth is less than the threshold (defined

based on the % of used traffic), we turn the CC ON and if the available bandwidth is more than the threshold, we turn the CC OFF.

When the data server receives a file transfer request from client, it sends a request to the advice server. The request includes the following parameters:

- IP address of client
- file length or remaining file length to be transferred

When the advice server receives the request, it determines the following parameters to send back to the data server:

- Flag to indicate whether to use congestion control or not in TCP
- RTT from server to client.
- Current available bandwidth
- Next advice time

RTT can be tested when the advice request arrives at the advice server, either ICMP programming or ping can be used to obtain the result. To get the congestion control flag, the advice server searches for the most recent observation of this destination from database. The result is a series of stored information in the form of

$$(ip_1, ?, b_1, c_1)$$

……

$$(ip_n, ?, b_n, c_n)$$

Here, ip_is are IP addresses of hops from the server to the destination, b_is are available bandwidth of corresponding hops, c_is are flags of congestion states. If the most recent test time t is not close enough to current time, the congestion control flag is set to inform the data server use congestion control during transmission. If the nearest test time t is close enough to current time, the congestion control flag is determined by congestion flags. If anyone of them is TRUE, which means congestion

happened in this hop, the congestion control flag will be turned on; if none of them is true, the congestion control flag will be turned off.

The available bandwidth and next advice time will be calculated according to historical data of the minimum bandwidth hop. When an advice request comes, the first step is to calculate the available bandwidth and rate of change of available bandwidth for each previous day with test data.

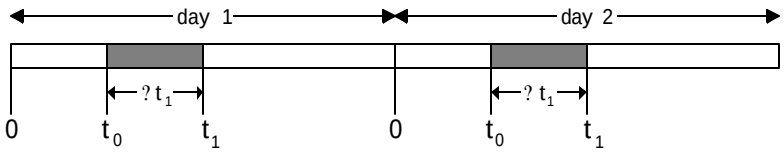

Figure 4: Illustration of calculation, from [4]

The advice request comes at time t_0 in current day. For each previous day, we do linear regression for history data from t_0 to $t_1 ? t_0 ? ? t_1$; then we get available bandwidth $?_0(i)$ and rate of change in available bandwidth $?_1(i)$, i is the index of the previous test days.

Current available bandwidth will be estimated as the mean value of series $?_0(i)$

$$?_0 ? \frac{1}{N} \sum_{i?1}^{N} ?_0(i)$$

This value will be sent back to the data server.

For the rate of change in available bandwidth $?_1(i)$, we calculate the mean value $?_1$ and standard deviation $?_{?_1}$, then the rate of change in available bandwidth will be bounded by $?_1 ? t_z ?_{?_1}$ and $?_1 ? t_z ?_{?_1}$ (t_z is determined by confident level requirement of normal distribution. According to theory, $?_1(i)$ is student-t distribution, hence

$?_1(i)$ can be approximated by normal distribution. For example, t_z ? 2 is for 95% confidence).

Denote S ? max($\left|?_1 ? t_z ?_{?_1}\right|, \left|?_1 ? t_z ?_{?_1}\right|$), S will be the maximum rate of change in available bandwidth. That is, from t_0, after a certain time interval $?$, the change in available bandwidth will be $S ? ?$.

The upper bound of change in available can be determined either by absolute value or percent of estimated bandwidth $?_0$, denote it as $? ?$. Then

$$S ? ? ? ? ? ? \quad ? ? ? ? / S$$

which gives the next advice time t_a ? t_0 ? ?.

The Advisory Server (AS) that is implemented in this manner will be co-located with the data server, for example an FTP server. The FTP server interacts with this AS and uses the network tuning parameters as suggested by the AS for all the data transfers to the FTP dients. Thus the AS helps in reducing the transfer times of large file transfers in FTP. The next chapter describes the modifications that are made to an FTP server in order to interact with the AS.

Chapter 4

Modifications to the FTP server

This chapter gives an overview of the File Transfer Protocol (FTP), the details of the implementation of ProFTPD, an FTP server daemon, and the changes that are made to ProFTPD so that it interacts with the Enable server while transferring large files to FTP clients.

4.1. Overview of File Transfer Protocol

File Transfer Protocol (FTP) [22] is an application protocol with TCP as its transport protocol. As the name suggests, its main purpose is to transfer files between the computers. FTP is implemented based on two connections, namely, *control connection* and *data connection*. The control connection is the communication path between the FTP client and the server to exchange the commands and replies. It follows the Telnet protocol [23]. The data connection is a full duplex connection over which data is transferred, in a specified mode and type. The FTP server listens on the standard port, 20. This is the port to which the FTP client connects in order to establish a control connection with the FTP server. The port number used on the FTP client side for the data connection is 21 and this is also standard. FTP client sends this data port among the other parameters to the FTP server in the initial set of commands. The FTP server establishes a connection based on the connection parameters sent by the client and starts the data transfer.

The communication channel from the client to the server process is established as a TCP connection from the client to the standard server port (20). The client sends the

FTP commands and interprets the replies received. The server interprets the commands, sends the replies to the client, establishes the data connection and transfers the data. There are different kinds of commands that a client sends to the server.

- *Access control commands*: These commands are used to control the access of the FTP server by the client.
- *Transfer parameter commands*: These commands specify to the server, the parameters to use for the current data transfer. These commands must precede the FTP service request.
- *FTP service commands*: These are the actual service requests from the clients. They define the file transfer or the file system function requested by the client.

4.2. Details of modifications to ProFTPD

For this thesis, ProFTPD [15] is selected as the target FTP server since it is a highly secure and configurable FTP server. Modifications are made to the ProFTPD daemon to make it interact with the Enable service. The design of ProFTPD daemon is derived from that of Apache web server and it can be run as a standalone server or it can run under *"inetd"*.

4.2.1. Implementation of ProFTPD

ProFTPD is an FTP server modeled around the Apache HTTP server, with a similar configuration file syntax and modular structure. The implementation details given in this section are based on [24]. ProFTPD handles the commands in a series of simple steps as follows:

- Preprocessing the command
- Processing the command
- Postprocessing the command
- Logging the command

These phases are handled by looking at each of the module, looking to see if it has a handler for the phase, and attempting to invoke it if there is one. The handler does one of the three things

- Handle the command and let the processing engine know that the command has been handled and it can proceed with its processing.
- Decline to handle the command and let the processing engine know that it should proceed its processing as if it has never called that handler.
- Signal an error by returning one of the FTP error codes [22]. This terminates the normal handling of the request; the command may be logged.

Most phases are terminated by the first module that handles them. The handlers are functions of one argument (a cmd_rec structure), which returns a `MODRET` (a modret_struc t`ypdef`ed to `MODRET`) [Appendix C].

4.2.2. Module structure

The details of all the modules and data structures are based on [24]. Each module declares the command handlers for the commands issued by the client, that it is interested in handling. The modules can also contain the code to handle the configuration commands. To handle these configuration directives the modules have the configuration directive handlers. These configuration directive handlers perform such checks as whether the configuration directive is in an appropriate context, whether the arguments are correct, etc. Each module has a *command handler table*, which links the client-issued commands with the interested handlers and a

configuration command hander table, which declares the configuration directives, and the corresponding configuration directive handlers.

Some of the data structures, which are used very often in these command handlers, are as follows: a ***`pool`*** is a pointer to a *resource pool* structure. These are used by the server to keep track of the memory which has been allocated, files opened, etc., either to service a particular request, or to handle the process of configuration itself. This is maintained so that when the request is over, the memory can be freed, and the files closed, *en masse*, instead of tracking them all down and disposing them.

The sole argument to handlers is a *cmd_rec* structure. This structure describes a particular command, which has been made to the server, on behalf of a client. Each connection by a client generates multiple *cmd_rec* structures, starting with the USER command. The ***`cmd_rec`*** contains pointers to a resource pool, which will be cleared when the server is finished handling the command, to structures containing per-server information, and most importantly, information on the command itself. There are also pointers to private data a handler has built in the course of servicing the command, and to a *server_rec*, which contains per (virtual) server configuration data. When the processing engine reads an FTP command from a client, it builds the corresponding *cmd_rec* structure by filling its fields. The filled-in *cmd_rec* is then handed off to the command handlers that have registered an interest in handling that particular FTP command.

4.2.3. Command Responses

Each handler, when invoked to handle a particular ***`cmd_rec`***, returns a `MODRET` to indicate what happened. That can be one of:

?? `HANDLED` -- the command was handled successfully. This may or may not terminate the phase.

?? `DECLINED` -- no erroneous condition exists, but the module declines to handle the phase; the server tries to find another.

?? `ERROR` -- an error has occurred while processing the command, which aborts its handling.

Each module handles the configuration directives by looking in its *configuration table*. As stated previously, this table contains information on what directives the module handles and the corresponding configuration handler. It takes only one argument, a *cmd_rec* pointer. That structure contains a bunch of arguments, including a resource pool, and the (virtual) server being configured, from which the module's per-server configuration data can be obtained if required. The module's configuration table has entries for all the directives it handles.

The entries in these tables are:

?? the name of the configuration directive

?? the function which handles it

?? a pointer which is set to the "owning" module when the module code is compiled; It is always set to `NULL`

Once all the configuration directives are handled by the appropriate handlers the module goes on to execute the command issued by the client. This it does by looking into the *command handler table* to see the command handler to call for the particular command issued by the client. Even the command handlers take a single argument of type *cmd_rec*.

ProFTPD daemon was modified so that all the download requests from the clients are handled by interacting with the Advisory Server. The modifications were made to the command handler, *cmd_retr,* which handles the download commands from the clients. Just before the server starts transferring the file, it contacts the Advisory

Server by sending it the client's IP address, the length of the file it has to still transfer in a ADVISING_REQUEST structure. Now the AS determines the correct network parameters (the CC State to use in TCP) to use for that particular client and sends them to the FTP server in an ADVISING_REPLY structure. In this structure, it also sends the time after which it has to contact it again to check if there is any change in the network path. Based on the response from the AS, we turn the CC State in TCP by using the setsockopt( ) system call interface provided to control the CC State in TCP [2] as shown below.

```
if (REP.ccstate == CONGEST_CTRL_OFF)
{
 param = 1;
setsockopt(session.d->outf->fd,IPPROTO_TCP,15,(char*)&param,sizeof(param));
}
```

Based on the next advice time given by the AS the FTP server contacts the AS again and it changes the CC State in TCP if necessary. Thus a mechanism is provided to dynamically change the CC State of TCP to accommodate the changes in the network for large FTP transfers.

The next chapter describes the tests that are done to evaluate the usefulness of the AS in reducing the reducing the FTP transfer times. It also describes the test environment used to do our tests and how that test environment is created.

Chapter 5

Evaluation of the Dynamic TCP Congestion Control Scheme

5.1. Introduction

In this chapter we evaluate how the Dynamic TCP Congestion Control Scheme in the Enable service effects the performance of the FTP server (ProFTPD in our case). Primarily we need to find out if this scheme is functioning properly and then to see if it really provides performance gains in the FTP transfers. Next we would like to determine how the Dynamic TCP Congestion Control scheme effects background traffic. Finally we would also like to see how the history data used by the Enable service effects its decision making process. To achieve all the above-mentioned tasks we performed the following tests.

?? Tests to evaluate the performance of FTP with the Enable service

For these tests we identified different times of a day, which represent the network states with different load in a day, and performed the large file transfers with different FTP implementations at these times. The different FTP implementations that we have tested here are the standard FTP, FTP with NOCC for the entire file transfer and FTP interacting with AS during the file transfer. We have done the tests with different FTP implementations for networks with different levels of congestion to see how the performance of FTP with Advisory Server (AS) varies with different levels of congestion in the network.

?? Tests to see the effect on the background flows

To determine the effect on the background traffic, we performed large file transfers by running FTP and a background flow simultaneously. From these tests we observed how the throughput of the background traffic is effected. We also performed tests with multiple background flows to have a better understanding of the effect on the background traffic.

?? Tests with different history databases

We did tests with different history data sets to determine how the history data effects the decision process of AS. For this we used three different history data sets. First set is the whole database of network data. Second set is network data of one week, immediately preceding the test day. This is chosen to see if using only the recent trends in the network behavior has any better effect on the decision process of the AS. The final set is the network data of all the test days (e.g., Fridays) as the history data.

To do all our tests, mentioned above, we have considered using the following three network environments.

Real Network

We can test the mechanism on a real High Bandwidth Delay Product network. This is the ideal case since it allows us to see the performance gain of the FTP transfers involving the real protocols in the target environment. For this we need to have access to an FTP server to which we can make the required modifications. In this case we do not have much control on how the network behaves. Also the tests in a real environment are not reproducible and make it difficult to identify any problems that occur. We have considered using this approach for our tests. But considering the problems mentioned above we did not use this approach.

Simulated Network

We could use a simulated network to do our tests. This involves rewriting the FTP code and the TCP code with necessary modifications for use in a simulation. This implementation for the simulation may differ from the real implementation. Also the simulated environment may not represent the real environment exactly. Also in all our tests, we used large files (16GBytes), which will take a long time to transfer using a simulator. Doing these tests with a software simulator is not practical, and hence did not use a simulated environment.

Emulated Network

An emulated network is an environment in which we emulate the conditions of a WAN in a lab-environment network. It is a controlled, reproducible environment for running real code. By using an emulated environment we will be dealing with the real protocols and will produce valid estimation of the performance of the transfer protocols. An emulated environment does not increase the test times and the tests are also reproducible in an emulated environment. Also emulated environments tend to be much nearer to the real environment than the simulated environments.

Considering the advantages mentioned above of using the emulated network over the simulated network and the real network, and the feasibly in creating an emulated environment, we decided to use an emulated network environment for all our experiments. The next subsection describes how we have created this emulated environment and what test scenarios we have used to perform our tests.

5.2. Creation of the Emulated Network Environment

To create an emulated WAN locally, we need to emulate the WAN conditions in a local environment. The WAN conditions we need are the large RTTs (e.g. 50ms), typical of WANs and the variation of the available bandwidth along a path, which

represents the congestion along a network path. NISTNet [14], a network emulation tool, is used to emulate these WAN effects. The overview of NISTNet is given below.

5.2.1. Overview of NISTNet

NISTNet is a network emulation package that runs on Linux. It is a general-purpose tool for emulating performance dynamics in IP networks. NISTNet allows a single Linux PC set up as a router to emulate a wide variety of network conditions. The tool allows controlled, reproducible experiments with network performance sensitive/adaptive applications and control protocols in a simple laboratory setting. It operates at the IP level.

NISTNet can emulate the critical end-to-end performance characteristics imposed by various wide area network situations (e.g., congestion loss) or by various underlying sub network technologies. NISTNet is implemented as a kernel module extension to the Linux operating system and an X Window System-based user interface application. *Appendix B.1* gives the details of the NISTNet usage.

5.2.2. Test Environment

A local network is set up with hosts running Linux operating system. NISTNet was installed on a Linux host. This host acts as the router and it applies the WAN conditions to all the packets that traverse through it. Three test configurations are used for all the tests. A 3-host configuration is used for tests with out any background flows and 6-host, 8-host configurations are used for tests with the background flows. These configurations are shown below.

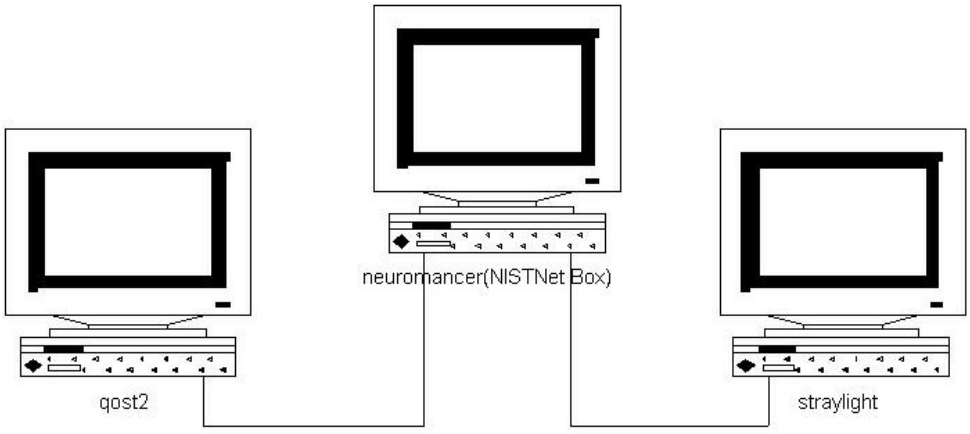

Figure 5: 3-host network configuration

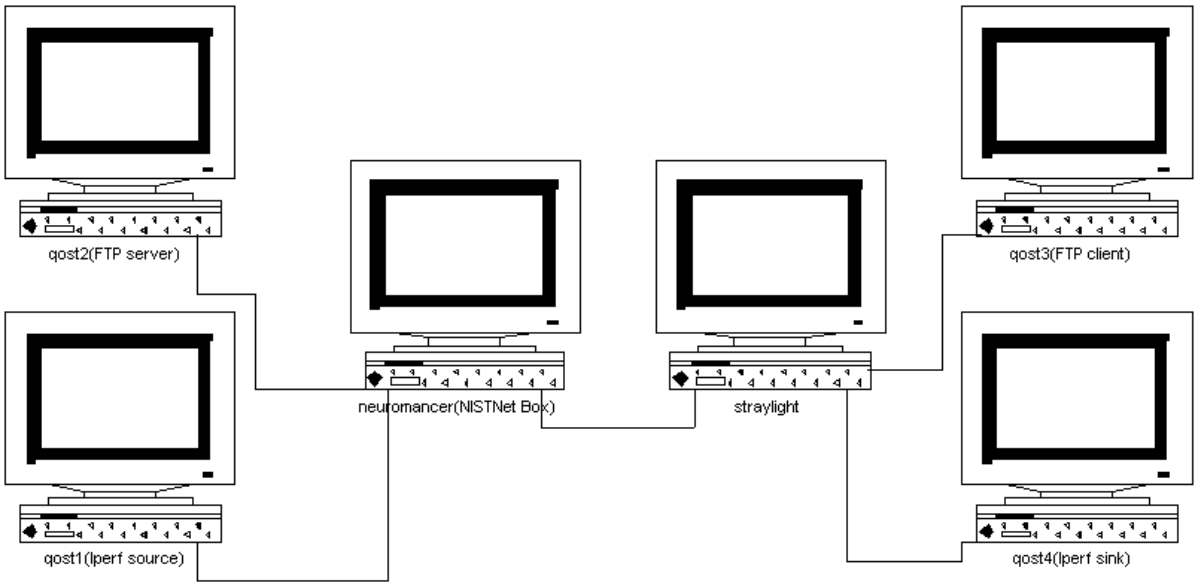

Figure 6: 6-host network configuration

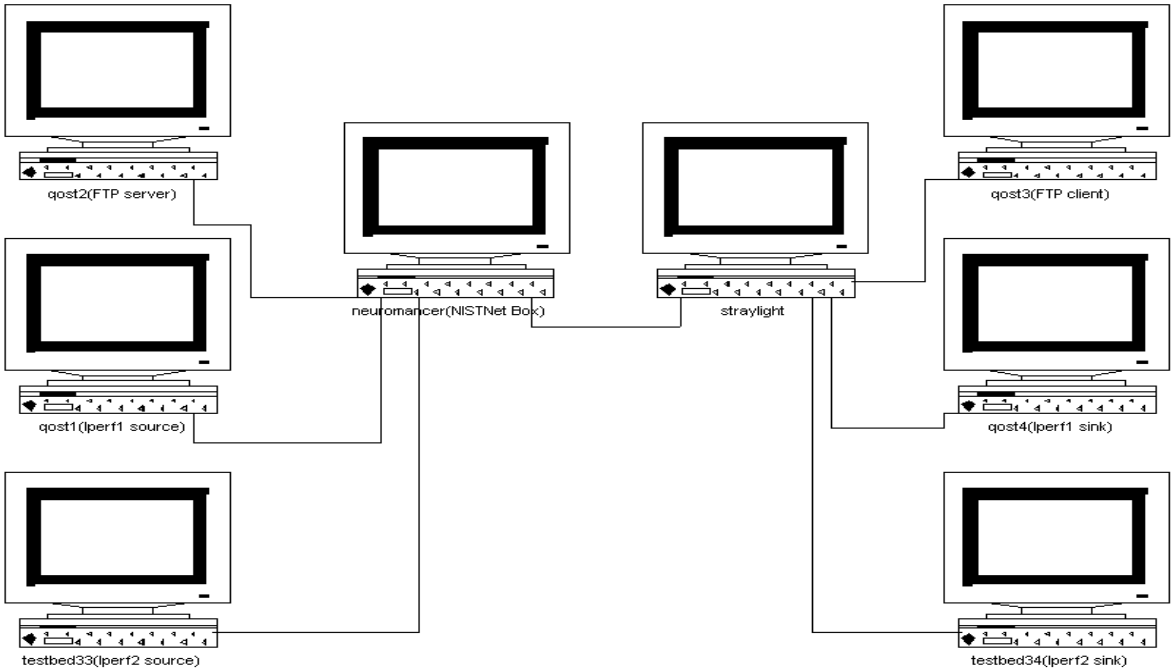

Figure 7: 8-host configuration

The specifications of all the machines used in the above configurations are shown in the *Appendix A*. All the connections between the machines are 100Mbps Ethernet.

The FTP server used for testing is the ProFTPD [15] daemon. Modifications are made to it, as described in C*hapter 4*, to interact with the Advisory Server during the transfer of large files to the clients. To make sure that the congestion control is changed properly we used tools such as tcpdump [26], tcptrace [27] and xplot [29]. During all the FTP transfers, large (1Mbytes) TCP buffers are used taking the Bandwidth Delay Product (BDP) into consideration. We have used the NcFTP client [16] since it allows us to set large receive windows as a configuration parameter. Iperf version 1.2.1 [17] is used to generate the background traffic for the tests.

We obtained the traffic data, specifically available bandwidth as a function of time, on the interface of the router, ks-2-a10-52.r.greatplains.net (IP address: 64.113.234.206) and used this data to emulate a WAN environment. This router interface connects the KU (University of Kansas) network to Internet2 [34]. Hence the data collected on this interface models the variation of the traffic in a real WAN. We stored this data in a MySQL [28] database as a set of records with the fields of the timestamp, and available bandwidth.

In order to create the WAN environment to do our tests, we have to emulate the WAN conditions, i.e., delay and the congestion state of the network. NISTNet is used to emulate both of these WAN conditions. To represent the continuous change in the congestion state of the network, we change the available bandwidth of the network link continuously by means of a NISTNet script. We limit the available bandwidth of the network link by introducing packet drops in the TCP traffic flowing through this link. There is a well-established equation [21], which relates the TCP throughput to the drop rate as shall be explained in the next section. We use this equation to determine the drop rate for a particular bandwidth value and use this in the NISTNet script to vary the available bandwidth on the link. As stated in the introduction, we do the tests at different times of a day. For each test time, we obtain the available bandwidth variation data starting at that time from the database of network data. We use this bandwidth data to generate the corresponding packet drop values, according to the equation relating the TCP throughput to the packet drop rate. Now we use this packet drop data in a NISTNet script to continuously vary the bandwidth available to the FTP traffic flowing through this network link. We also apply the WAN delays to the FTP traffic using the same NISTNet script. This way we run all our tests in a local network with NISTNet emulating the WAN conditions.

5.3. Equation relating the TCP throughput to the drop rate

TCP uses Additive Increase Multiplicative Decrease algorithm [1] for congestion control. On detecting a loss it decreases the size of its congestion window by a factor of two and attempts to get extra bandwidth by increasing the window linearly when there is no congestion. The long-term throughput of a TCP flow and the packet drop rate is approximated by the following equation [21].

$$P = [(C * S) / (RTT * T_{TCP})]^2$$

Where P is the packet loss rate, C is a constant, S is the packet size, RTT is the round trip time including queuing delay, and T_{TCP} is the long-term TCP throughput. Here the current available bandwidth is obtained from the database of network data and a corresponding packet drop rate is calculated using the above equation. This drop rate is then applied to the traffic flow using a NISTNet script.

5.4. Tests to show the validity of the throughput-drop relation

Equation showing the relationship between the long-term throughput of the TCP flow and the packet loss rate is shown in Section 5.3. In order to estimate the constant factor, C, we applied different packet drop rates to an Iperf flow and compared the observed throughputs with the theoretical value for two different values of C. The results of these tests are shown in *Table 1*. Iperf is used to generate the traffic flow for these tests.

Drop %	Throughput (Mbps) With Delay = 25ms			Throughput (Mbps) With Delay = 50ms		
	From eqn. C=9.76	From eqn. C=12.2	Observed	From eqn. C=9.76	From eqn. C=12.2	Observed
1	4.68	5.86	4.1	2.34	2.93	2.3
2	3.31	4.14	2.9	1.66	2.07	1.6
3	2.70	3.38	2.3	1.35	1.69	1.3
4	2.34	2.93	2.1	1.17	1.46	1.1

Table 1: Results of tests done to determine the constant factor in TCP throughput drop relation

From *Table 1* we can observe that, the observed throughput is near to the value calculated from the equation relating the TCP throughput to the packet drop rate, mentioned in Section 5.3, when the constant factor used is equal to 9.76. Hence for all our experiments we used C = 9.76.

5.5. Tests to evaluate the performance of FTP with the Enable service

5.5.1. Performance of FTP as a function of load in the network

We used the 3-host configuration, shown in *Figure 5*, for all the tests of this section. The data we collected is from a router whose output link capacity is 55Mbps. Hence the maximum available bandwidth that can be observed in the collected data is 55Mbps. Since the drops, that we introduce to limit the throughput, are based on the collected data, the throughput we can observe with standard TCP in our tests is no more than 55Mbps. But all the links in our local test environment are of 100Mbps. So

we can assume that the remaining 45Mbps of bandwidth available on the link is used by other traffic in all the experiments. Hence we can think of the experiments are done with minimum of 45% used traffic. We also converted all the collected data such that the maximum available bandwidth values ranged from 25Mbps to 40Mbps. This is done to see how the performance of our mechanism varies with different network congestion states or available bandwidth values in the network. Hence the tests are done with networks of different congestion levels. In our case a slightly congested network is a network with minimum of 45% used bandwidth, a moderately congested network is a network with a minimum of 60% used bandwidth and a highly congested network is a network with a minimum of 75% used bandwidth.

5.5.1.1. Tests in a slightly congested network

For all the tests in this section a 16GByte file is transferred with different FTP implementations at different times of the test day. The test timings chosen are 12:15am, when the network load, as observed from the network data collected, is high, 7:25pm, when the network load is low and 3:25am, 2:25pm, 9:10pm, when there is a large variation in the network load. Advisory Server (AS) uses the whole network data in the database as the history data.

TCP buffer size	1Mbyte
NISTNet delays	50ms
FTP transfer size	16Gbyte
NISTNet drops	YES
Available Bandwidth (ABW) Threshold	42Mbps (58% used bandwidth)
History data used	Whole database

Table 2: Parameters for tests to estimate the performance of FTP as a function of network load in a slightly congested network

The results are shown in the *Table 3* and *Figure 8* below.

Time of Day	FTP with CC		FTP with NOCC		FTP with Adv.Server		
	Throughput (Mbps)	Transfer Time (h:mm:ss)	Throughput (Mbps)	Transfer Time (h:mm:ss)	Throughput (Mbps)	Transfer Time (h:mm:ss)	Percentage improvement over FTP with CC
12:15am	45.68	0:47:48	70.08	0:31:09	45.44	0:48:03	00.00
03:25am	45.68	0:47:48	70.00	0:31:12	57.60	0:37:54	26.09
07:25am	45.60	0:47:54	70.08	0:31:09	70.08	0:31:11	53.68
02:25pm	45.92	0:47:34	70.08	0:31:10	53.76	0:40:36	17.07
09:10pm	45.84	0:47:39	70.08	0:31:10	68.08	0:32:05	48.52

Table 3: Throughput of different FTP implementations in a network with a minimum of 45% used bandwidth and no background flows

Average percentage improvement in the FTP throughput with Advisory Server = 29.072%

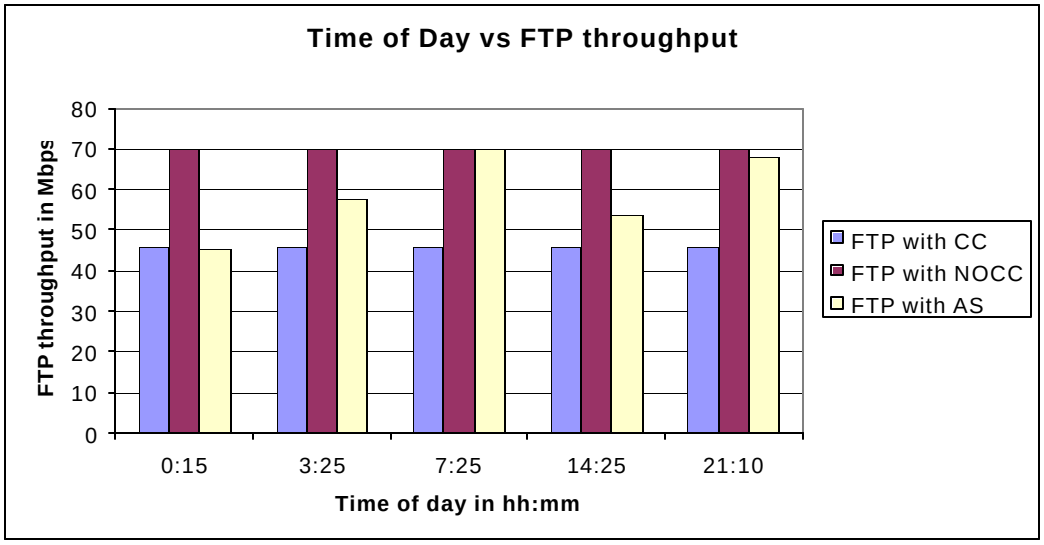

Figure 8: Time of day vs FTP throughput for tests in a network with minimum of 45% used bandwidth and no background flows

From *Figure 8* we can see that the throughput of FTP with AS is between that of FTP with CC and FTP with NOCC depending upon the CC inputs received from the AS. At 12:15am, CC state inputs received from the AS are ON and hence the throughput of FTP with AS is same as that of FTP with CC. At 7:25am, CC state inputs received from the AS are OFF and hence the throughput of FTP with AS is same as that of FTP with NOCC. At other times the throughput of FTP with AS varies between that of FTP with CC and FTP with NOCC based on the CC state inputs received from the AS during the FTP transfer. The congestion control state inputs received from the Advisory Server during the tests are shown in the *Table 4* below.

Time of Day	CC State inputs
12:15am	1(4)
03:25am	1,0(3)
07:25am	0(2)
02:25pm	0,1(2),0
09:10pm	0(3),1

Table 4: Congestion control state inputs received for tests in a network with a minimum of 45% used bandwidth and no background flows

In the CC State inputs column of *Table 4* a value of '0' indicates the CC state input of OFF and a value of '1' indicates the CC state input of ON. 0(x) indicates that there are x number of inputs of 0 continuously. 0(x), 1(y) indicates that there are "x" number of continuous inputs of 0 followed by "y" number of continuous inputs of 1. From *Table 4* we see that at 12:15am we receive the inputs as ON for the whole transfer. This is because, the available bandwidth during this transfer is below the threshold of 42Mbps. Also at 7:25am, all the inputs are OFF. This is because the available bandwidth at this time is above the threshold for the entire transfer. At 3:25am, the available bandwidth starts with a value below the threshold and then

increases above the threshold value. Hence the CC State inputs received from the AS are ON at the start of the transfer and then OFF afterwards. Similarly at 2:25pm and 9:10pm the available bandwidth starts with a value above the threshold and then decreases. Hence the AS starts with an input of OFF and then gives the inputs of ON. The plot showing the CC State inputs from the AS and the available bandwidth variation for the test time 2:25pm is shown in the Figure 9 below.

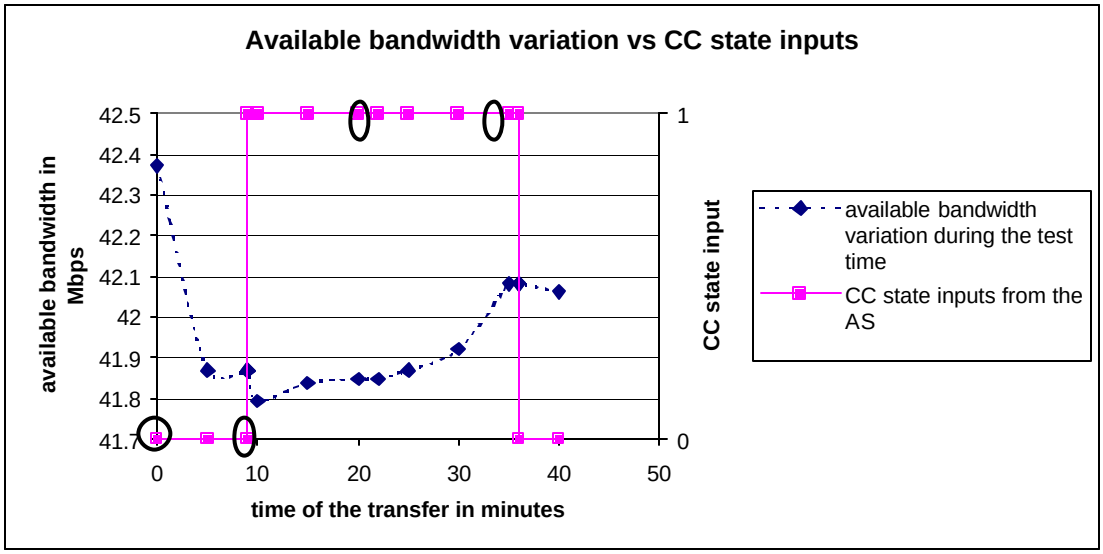

Figure 9: Available bandwidth variation vs CC state inputs at 2:25pm for tests in a network with minimum of 45% used bandwidth and no background flows

In *Figure 9* the dotted line shows the available bandwidth variation during the FTP transfer and the solid line shows the times at which the AS is contacted and the CC state inputs received from the AS during the FTP transfer. From *Figure 9* we can see that when the available bandwidth is more than the threshold (42Mpbs in this case), the CC state input from the AS is OFF (value = 0 in the figure). Also when the available bandwidth is less than the threshold, the CC state input fom the AS is ON (value = 1 in the figure). We can also see from the figure that there are two CC state transitions during the transfer, one at 9 minutes into the FTP transfer and the other at around 36 minutes into the transfer. The circled dots on the solid line in the *Figure 9*

indicate the times at which the enable server is contacted and CC state inputs received at those times.

5.5.1.2. Tests in a moderately congested network

For our tests we have considered a network with a minimum of 60% used bandwidth as a moderately congested network. In this section we perform large file transfers with different FTP implementations. *Table 5* below lists the parameters used for our tests.

TCP buffer size	1Mbyte
NISTNet delays	50ms
FTP transfer size	16Gbyte
NISTNet drops	YES
Available Bandwidth (ABW) Threshold	27Mbps (73% used bandwidth)
History data used	Whole database

Table 5: Parameters for tests to estimate the performance of FTP as a function of network load in a moderately congested network

Time of Day	FTP with CC		FTP with NOCC		FTP with Adv.Server		
	Throughput (Mbps)	Transfer Time (h:mm:ss)	Throughput (Mbps)	Transfer Time (h:mm:ss)	Throughput (Mbps)	Transfer Time (h:mm:ss)	Percentage Improvement over FTP with CC
12:15am	25.68	1:24:59	69.68	0:31:21	25.60	1:25:15	00.00
03:25am	31.84	1:08:41	69.84	0:31:15	57.68	0:37:53	81.16
07:25am	44.40	0:49:11	70.08	0:31:10	70.00	0:31:10	57.66
02:25pm	28.80	1:15:54	69.76	0:31:18	40.24	0:54:18	39.72
09:10pm	27.92	1:18:07	69.76	0:31:17	43.84	0:49:48	57.02

Table 6: Throughput of different FTP implementations when run in a network with a minimum of 60% used bandwidth and no background flows

Average percentage improvement in the throughput time for all the tests = **47.112%**

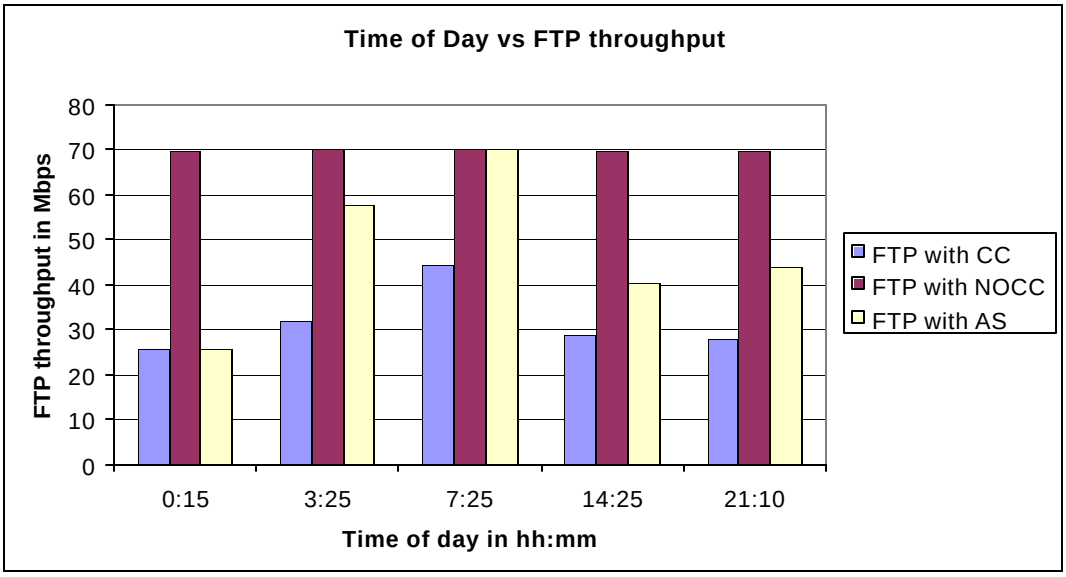

Figure 10: Time of day vs FTP throughput for tests in a network with a minimum of 60% used bandwidth and no background flows

Figure 10 shows that the throughput of FTP with AS varies between that of FTP with CC and FTP with NOCC. This behavior is similar to the one in the previous case shown in *Figure 8*.

Time of Day	CC State inputs
12:15am	1(12)
03:25am	1,0(9)
07:25am	0(5)
02:25pm	0(2),1(4),0(3)
09:10pm	0(5),1(4)

Table 7: Congestion Control State inputs received for tests in a network with minimum of 60% used bandwidth and no background flows

Table 7 shows the CC State inputs received from the AS at different test times. This is similar to the previous case. The only difference is that, the number of inputs received from the AS during the transfers in this case is more. This increase in the number of inputs is because of two reasons. First, the transfer time is longer in this case because the maximum available bandwidth for the standard TCP in this case is only 40Mbps. Second, the Next Advice Times (NAT) given by the AS in this case are less than that of previous case. This is because the amount of bandwidth change to trigger the next advice from the AS is lower. The plot showing the CC State inputs from the AS and the available bandwidth variation for the test time 2:25pm is shown in the *Figure 11* below.

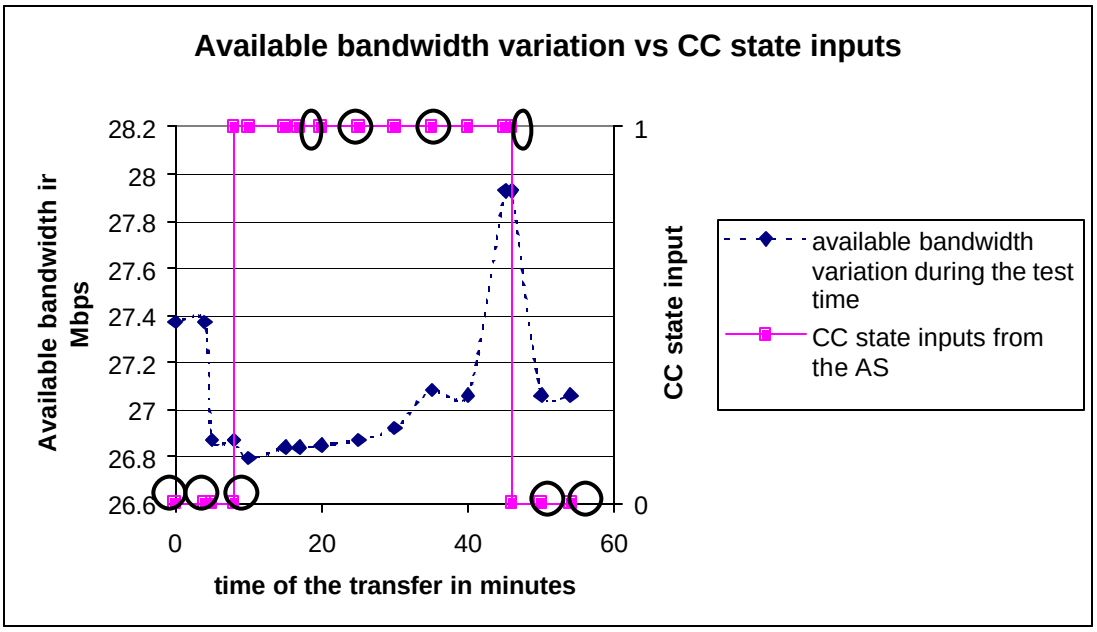

Figure 11: Available bandwidth variation vs CC state inputs at 2:25pm for tests in a network with a minimum of 60% used bandwidth and no background flows

In *Figure 11* the dotted line represents the available bandwidth variation during the FTP transfer and the solid line shows the times at which the AS is contacted and the CC state inputs received from the AS during the FTP transfer. From *Figure 11* we can see that at the beginning of the transfer, when the available bandwidth is more than the threshold (27Mpbs in this case), the CC state input from the AS is OFF (value = 0 in the figure). Also when the available bandwidth is less than the threshold, the CC state input from the AS is ON (value = 1 in the figure). We can also see from this figure that there are two CC state transitions during the transfer, one at around 8 minutes into the FTP transfer and the other at around 46 minutes into the transfer. The circled dots on the solid line in the *Figure 11* indicate the times at which the enable server is contacted and CC state inputs received at those times.

5.5.1.3. Tests in a highly congested network

For our tests we have considered a network with a minimum of 75% used bandwidth as a highly congested network. In this section we perform large file transfers with different FTP implementations and observe how the use of Enable service effects the performance of FTP transfers. The parameters used for the tests in this section are listed in *Table 8* below.

TCP buffer size	1Mbyte
NISTNet delays	50ms
FTP transfer size	16Gbyte
NISTNet drops	YES
Available Bandwidth (ABW) Threshold	12Mbps (88% used bandwidth)
History data used	Whole database

Table 8: Parameters for tests to estimate the performance of FTP as a function of network load in a highly congested network

Time of Day	FTP with CC		FTP with NOCC		FTP with Adv.Server		
	Throughput (Mbps)	Transfer Time (h:mm:ss)	Throughput (Mbps)	Transfer Time (h:mm:ss)	Throughput (Mbps)	Transfer Time (h:mm:ss)	Percentage Improvement Over FTP with CC
12:15am	11.68	3:07:13	64.96	0:35:15	16.88	2:00:35	44.52
03:25am	15.68	2:19:18	68.32	0:31:54	58.40	0:37:23	272.45
07:25am	22.24	1:38:04	69.20	0:31:35	68.56	0:31:50	208.27
02:25pm	14.24	2:33:03	69.12	0:31:36	34.72	1:02:51	143.82
09:10pm	13.04	2:47:32	68.56	0:31:52	29.28	1:14:31	124.54

Table 9: Throughput of different FTP implementations when run in a network with a minimum of 75% used bandwidth and no background flows

Average percentage improvement in the transfer time for all the tests = **158.72%**

Figure 12 below shows the throughput of different implementations of FTP, i.e. FTP with CC, FTP with NOCC and FTP with AS.

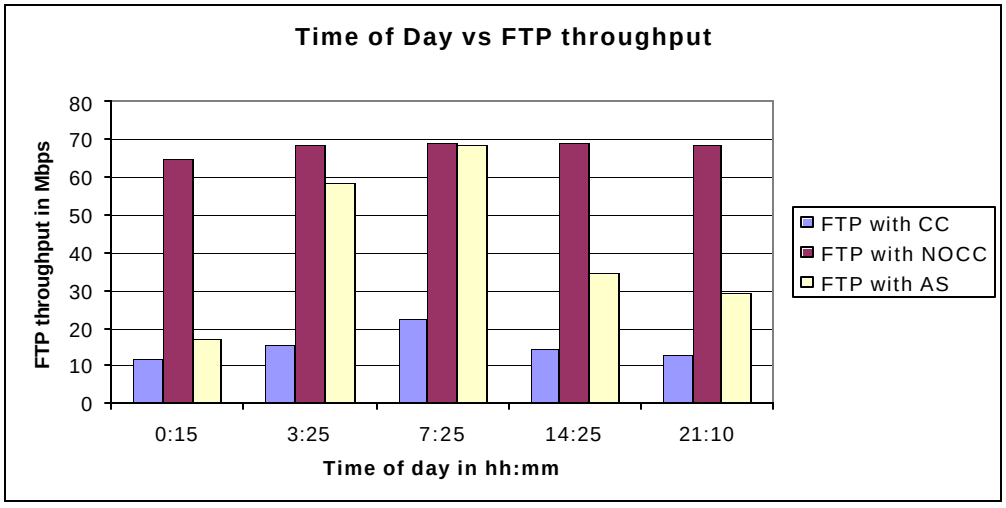

Figure 12: Time of day vs FTP throughput for tests in a network with a minimum of 75% used bandwidth and no background flows

Figure 12 shows that the throughput of FTP with AS varies between that of FTP with CC and FTP with NOCC. This behavior is similar to the one in the previous cases as shown in *Figure 8* and *Figure 10*.

Table10 shows the CC State inputs received from the AS at different test times.

Time of Day	CC State inputs
12:15am	1(46),0(23)
03:25am	1,0(48)
07:25am	0(18)
02:25pm	0(7),1(9),0(21),1(3)
09:10pm	0(24),1(6),0(4),1(9)

Table 10: Congestion Control State inputs received for tests in a network with a minimum of 75% used bandwidth and no background flows

From *Table 10* we see that the number of inputs received from the AS in this case is larger than the previous case. Again the reasons for this is same as explained in the previous case. The only difference in this case is that the number of state changes has increased at some test times, 12:15am, 2:25pm and 9:10pm. This is because of the increase in the transfer times. The plot showing the CC State inputs from the AS and the available bandwidth variation for the test time 2:25pm is shown in the Figure 13 below.

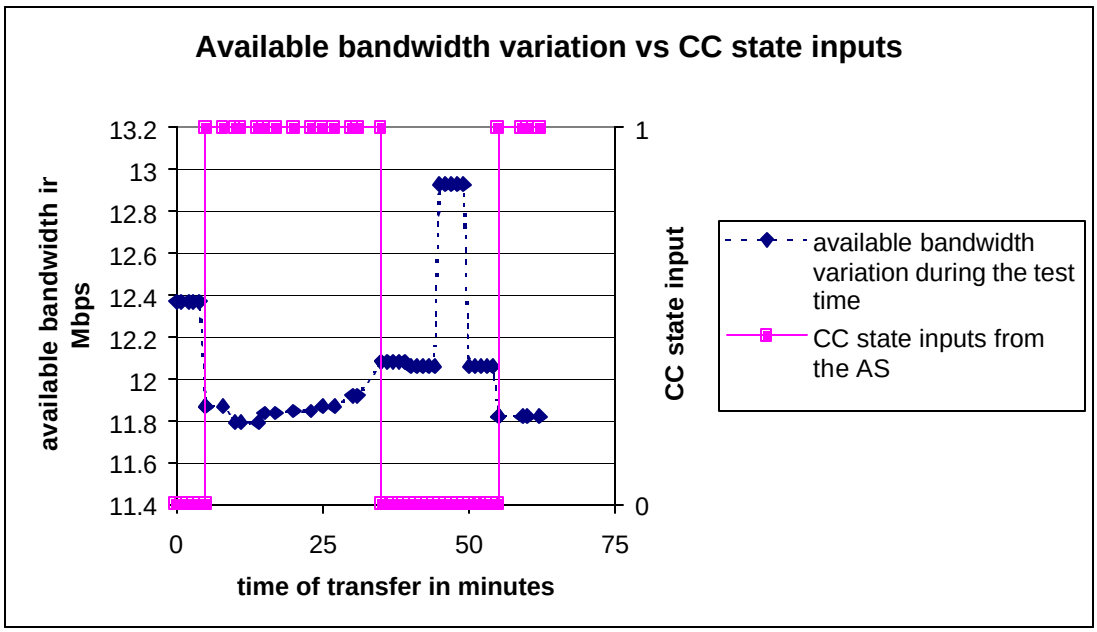

Figure 13: Available bandwidth variation vs CC state inputs at 2:25pm for tests in a network with a minimum of 75% used bandwidth and no background flows

In *Figure 13* the dotted line represents the available bandwidth variation during the FTP transfer and the solid line shows the times at which the AS is contacted and the CC state inputs received from the AS during the FTP transfer. From *Figure 13* we can see that at the beginning of the transfer, when the available bandwidth is more than the threshold (12Mpbs in this case), the CC state input received from the AS is OFF (value = 0 in the figure). The when the available bandwidth is less than the threshold, the CC state input from the AS is ON (value = 1 in the figure). At around 45 minutes there is an increase of around 1Mbps in the available bandwidth and since the available bandwidth was above the threshold value, the CC state remained as OFF. We can also observe that there are three CC state transitions in this case.

Figure 14 below shows the comparison of the average throughput values of different FTP implementations.

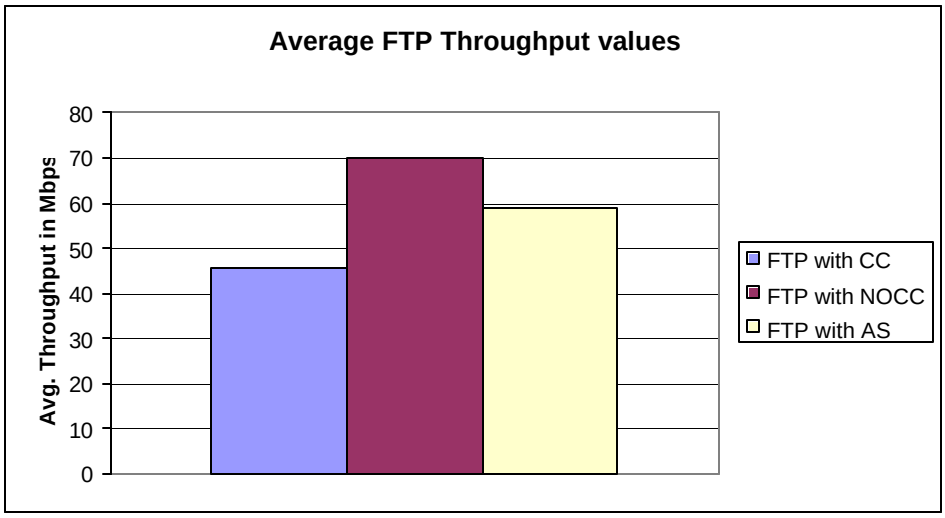

Figure 14: Average throughput of different FTP implementations in a slightly congested network

Figure 14 shows the average throughput of different FTP implementations. This is for the case where the minimum used traffic is 45%. From *Figure 14* we see that the throughput of FTP with AS is in between that of FTP with CC and FTP with NOCC. This is because it can take advantage of the available bandwidth when there is not much congestion in the network and at the same time it is not as aggressive as FTP with NOCC. The same behavior is observed for the other two cases.

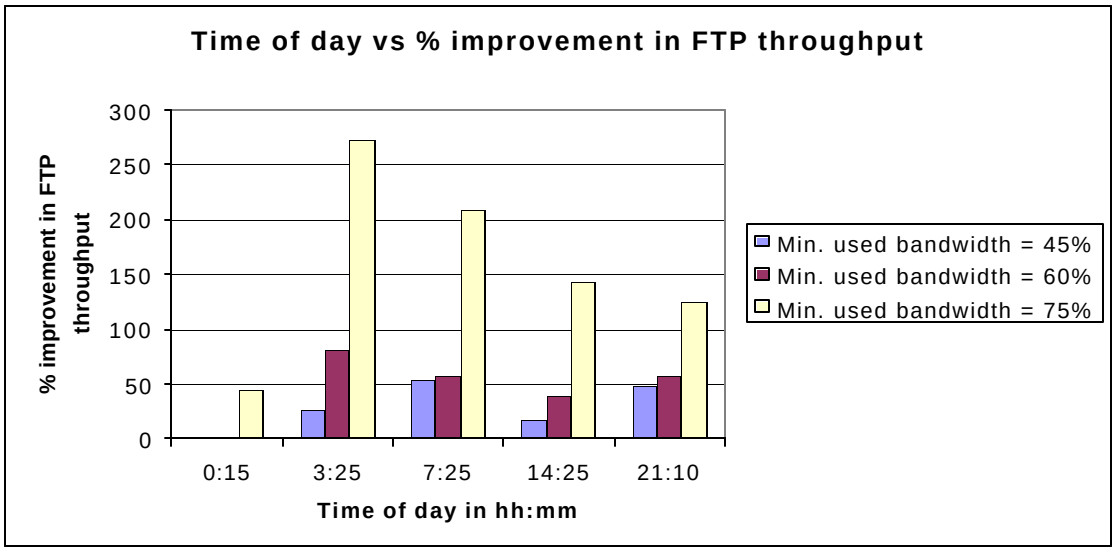

Figure 15: Time of day vs % improvement in the throughput of FTP in networks with different percentages of used bandwidth

Figure 15 shows the percentage improvement in the throughput of FTP with AS over standard FTP for different cases of used traffic. From *Figure 15* we see that the percentage improvement in the FTP throughput is greater when the used bandwidth is high. This is because for this case the throughput of standard FTP is lower than that of other cases because of the high percentage of used traffic and FTP with AS remains aggressive on the background traffic. But the background traffic will be effected more for this case. We can also observe that at 00:15, there is improvement in the throughput of FTP only for the case where the minimum used bandwidth is 75%. This is because for this case (minimum used bandwidth = 75%) there is a CC state transition from CC to NOCC during the file transfer as observed from *Table 10* and hence there was some improvement in the throughput. But for the other cases there is no CC state transition during the file transfer, as observed from *Table 4* and *Table 7*, and the throughput of FTP with AS is same as that of the standard FTP. Hence there was no throughput improvement for these cases.

5.5.2. Performance of the background flows

From the test results of previous section we observed that there is improvement in the performance of FTP when it is interacting with the AS. But we would like to determine how much the background traffic is effected. In order to determine the effect on the background traffic, during all the FTP transfers we ran a background flow and observed the throughput of the background traffic. Here we also did the tests with multiple background flows.

5.5.2.1. Tests with a single background flow

The 6-host configuration, shown in *Figure 6*, is used for these tests. The logical network topology that we create with the configuration is shown below in *Figure 16*. It shows that two flows travelling through two different network paths pass through the same router, *straylight*.

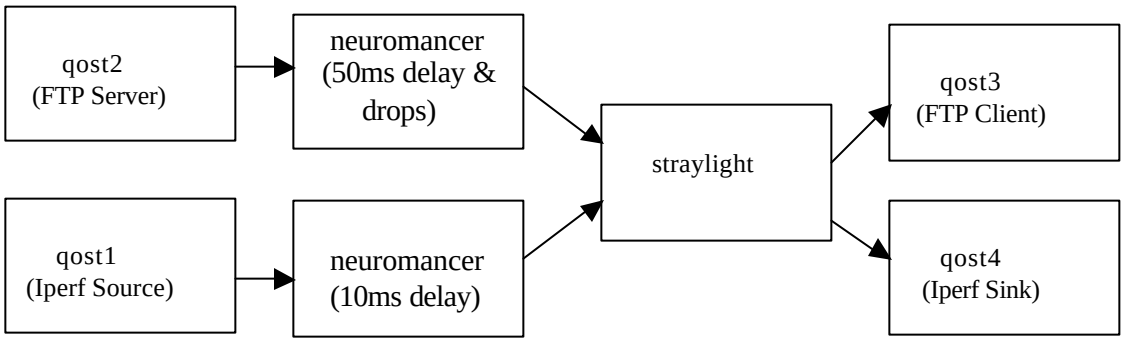

Figure 16: Logical network topology of the 6-host configuration

The FTP parameters and the background flow parameters used for the tests in this section are shown below in *Table 11* and *Table 12* respectively.

TCP buffer size	1Mbyte
NISTNet delays	50ms
FTP transfer size	16Gbyte
NISTNet drops	YES
Minimum used bandwidth	45%
Available Bandwidth (ABW) Threshold	42Mbps (58% used bandwidth)
History data used	Whole database

Table 11: FTP parameters for tests to see the effect on a single background flow

TCP buffer size	128Kbyte
NISTNet delays	10ms
NISTNet drops	NO

Table 12: Background flow parameters for tests to see the effect on a single background flow

The throughput of background flow (Iperf) when run with no background traffic is 83.4Mbps.

The results of the tests with a single background flow are shown in *Table 13* and *Table 14* below.

Time of Day	Throughput in Mbps					
	FTP with CC		FTP with AS		FTP with NOCC	
	FTP	Background flow	FTP	Background flow	FTP	Background flow
12:15am	39.76	52.10	39.20	52.82	69.52	20.61
3:25am	39.60	52.22	53.52	37.97	69.60	20.63
7:25am	39.60	52.21	69.68	20.60	69.68	20.58
2:25pm	39.68	52.17	48.56	42.49	69.76	20.58
9:10pm	39.68	52.12	67.20	22.73	69.76	20.59

Table 13: Throughput of different FTP implementations when run with a single background flow

Time of Day	% decrease in the throughput of background flow when run with FTP		
	FTP with CC	FTP with AS	FTP with NOCC
12:15am	37.53	36.67	75.29
3:25am	37.39	54.47	75.26
7:25am	37.40	75.30	75.32
2:25pm	37.45	49.05	75.32
9:10pm	37.51	72.75	75.31

Table 14: Percentage decrease in the throughput of background flow when run with different implementations of FTP

Average decrease, with respect to no interfering traffic, in the throughput of the background flow when run with FTP with CC = **37.46%**

Average decrease, with respect to no interfering traffic, in the throughput of the background flow when run with FTP with AS = **57.65%**

Average decrease, with respect to no interfering traffic, in the throughput of the background flow when run with FTP with NOCC = **75.30%**

The throughput of the background flow when run with different implementations of FTP is shown in the *Figure 17* below.

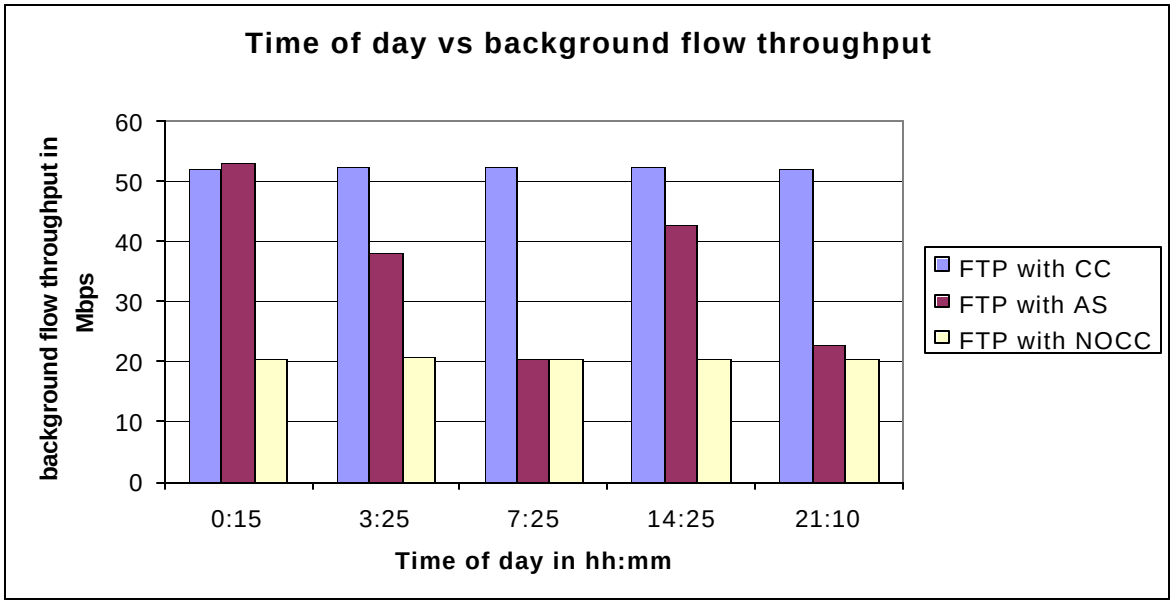

Figure 17: Time of day vs background flow throughput when run with different FTP implementations in a network with the a minimum used bandwidth of 45%

From *Table 14* we observe that the percentage decrease in the throughput of the background flow is high when FTP is run with NOCC. This is slightly reduced when FTP is run with AS. *Figure 17* shows the variation of the background flow throughput when run with different implementations of FTP. From the *Figure 17* we observe that the throughput of the background flow is the least when FTP is run with NOCC and highest when FTP is run with CC. When FTP is run with AS, the background flow throughput varies between these two values depending upon the CC state inputs obtained from the AS. This shows that FTP with AS is less aggressive on the background traffic than the FTP with NOCC.

5.5.2.2. Tests with multiple background flows

The 8-host configuration, shown in *Figure 7*, is used for these tests. The logical network topology that we create with the configuration is shown below in *Figure 18*.

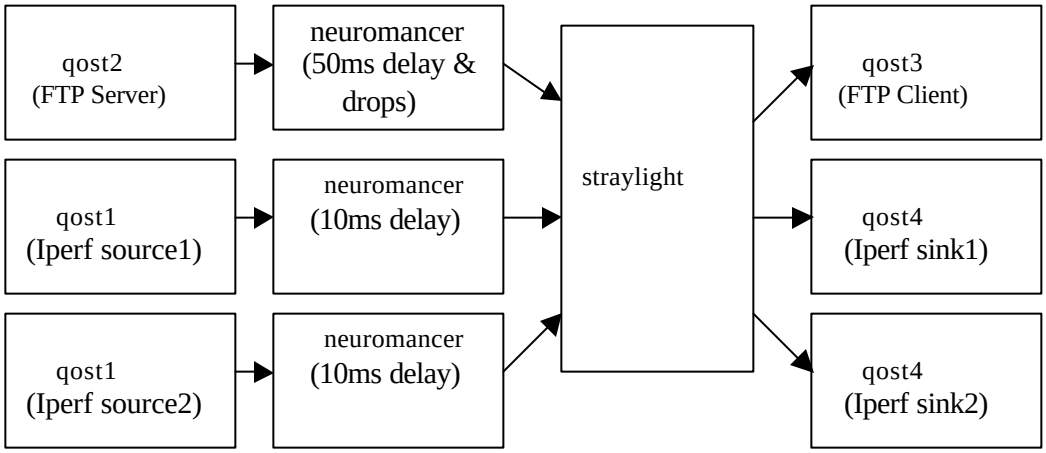

Figure 18: Logical network topology of 8-host configuration

The FTP parameters and the Iperf parameters used for the tests in this section are shown below in *Table 15* and *Table 16* respectively.

TCP buffer size	1Mbyte
NISTNet delays	50ms
FTP transfer size	16Gbyte
NISTNet drops	YES
Minimum used bandwidth	45%
Available Bandwidth (ABW) Threshold	42Mbps (58% used bandwidth)

Table 15: FTP parameters for tests to see the effect on multiple background flows

TCP buffer size	128Kbyte
NISTNet delays	10ms
NISTNet drops	NO

Table 16: Background flow parameters for tests to see the effect on multiple background flows

The results of the tests with multiple background (Iperf) flows are shown in *Table 17* and *Table 18* below.

Time of Day	Throughput in Mbps					
	FTP with CC			FTP with NOCC		
	FTP	Background flow1	Background flow2	FTP	Background flow1	Background flow2
12:15am	38.80	25.79	27.53	69.28	10.28	10.61
3:25am	39.04	25.78	27.25	69.36	10.24	10.64
7:25am	38.96	25.85	27.31	69.28	10.32	10.60
2:25pm	38.48	26.02	27.64	69.28	10.45	10.42
9:10pm	39.04	25.86	27.25	69.28	10.26	10.60

Table 17: Throughput of FTP with CC and FTP with NOCC when run with multiple background flows

Time of day	Throughput when FTP is run with AS		
	FTP	Background flow1	Background flow2
12:15am	38.64	25.83	27.60
3:25am	52.24	23.04	24.17
7:25am	69.20	10.74	10.28
2:25pm	47.68	21.24	22.96
9:10pm	66.48	12.03	11.29

Table 18: Throughput of FTP with AS when run with multiple background flows

The throughput variations of the background flows when run with different FTP implementations are shown in *Figure* 19 and *Figure 20* shown below.

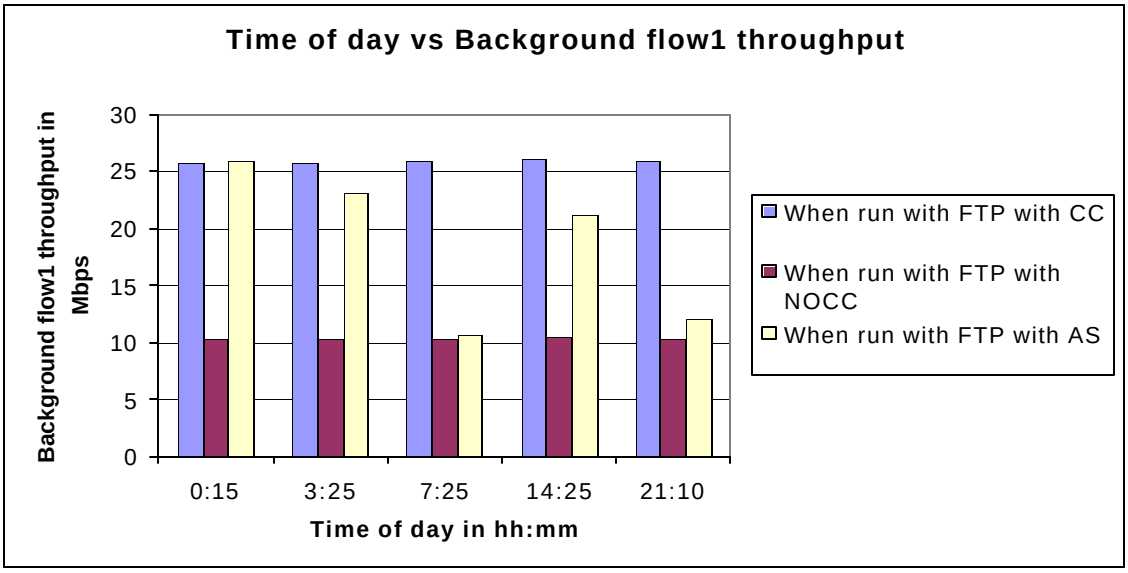

Figure 19: Time of day vs Background flow1 throughput when run with different FTP implementations in a network with minimum of 45% used bandwidth

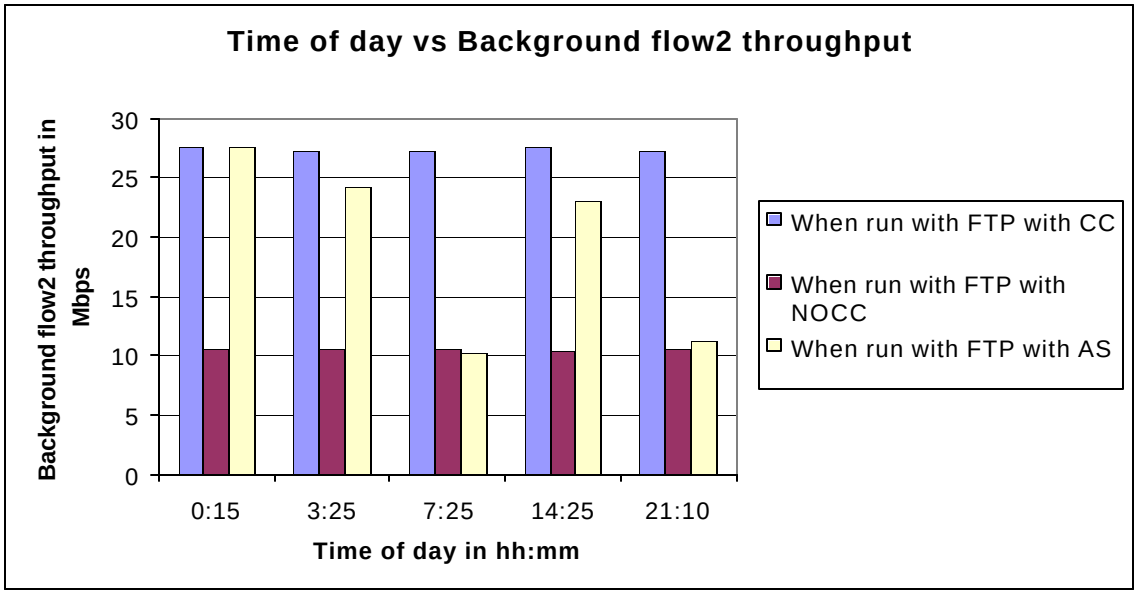

Figure 20: Time of day vs Background flow2 throughput when run with different FTP implementations in a network with minimum of 45% used bandwidth

From *Figure 19* and *Figure 20* we observe that throughputs of the background flows are the least when FTP is run with NOCC and highest when FTP is run with CC. When FTP is run with AS, the background flow throughput varies between these two values depending upon the CC state inputs obtained from the AS. This is same as the case with a single background flow. We also observe from *Table 18* that the throughput is divided equally between the two background flows. These results show that FTP with AS has similar effects on the multiple background flows.

5.5.3. Tests with different history data sets

In order to determine the effect of the history data sets on the way the AS gives the inputs to the FTP server, we tested by using three different history data sets, which the AS uses to make its decisions. These three history data sets are listed below. These tests are done with networks with different levels of congestion to have a better idea of the effect of history data sets.

- The available bandwidth data of all the days in the database of network data. See *Figure 21* below.

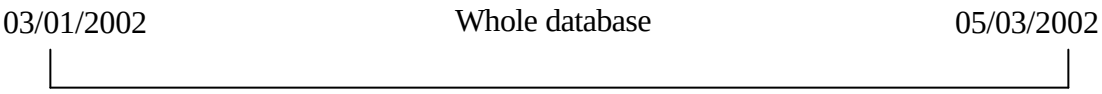

Figure 21: History data set comprising of the whole database

- The available bandwidth data of only one-week prior to the test day. See *Figure 22* below.

Figure 22: History data set comprising of one-week's data

✍ The available bandwidth data of all the test days (e.g. Fridays). See *Figure 23* below.

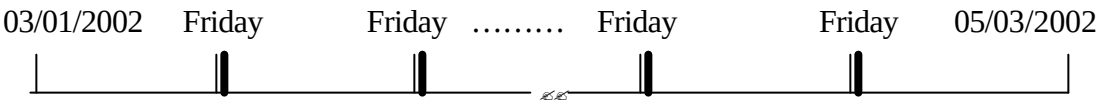

Figure 23: History data set comprising the data of test days

5.5.3.1. Tests in a slightly congested network

The parameters used for the tests in this section are listed in *Table 19* below.

TCP buffer size	1Mbyte
NISTNet delays	50ms
FTP transfer size	16Gbyte
NISTNet drops	YES
Available Bandwidth (ABW) Threshold	42Mbps (58% used bandwidth)

Table 19: Parameters for tests with different history data sets in a slightly congested network

The results of the tests with different history data sets are shown in *Table 20* and *Table 21* below.

Time of Day	Throughput of FTP when run with AS in Mbps		
	History = whole dB	History = Previous Fridays	History = Previous week
12:15am	45.44	45.52	45.36
3:25am	57.60	52.96	51.36
7:25am	70.08	70.00	70.08
2:25pm	53.76	57.84	70.08
9:10pm	68.08	70.08	70.08

Table 20: Throughput of FTP with AS when run in a network with minimum of 45% used bandwidth and no background flows for different history data sets,

Time of day	CC states received when FTP is run with AS		
	History = whole dB	History = Previous Fridays	History = Previous week
12:15am	1(4)	1(4)	1(3)
3:25am	1,0(3)	1,0	1,0
7:25am	0(2)	0(2)	0
2:25pm	0,1(2),0	0,1	0
9:10pm	0(3),1	0(2)	0(2)

Table 21: CC State inputs received when FTP with AS is run in a network with a minimum of 45% used bandwidth and no background flows for different history data

From *Table 20* we observe that there is not much difference in the throughput of FTP at most of the test times with different history data sets. The major difference we see will be in the number of times that we contact the AS and the number of CC state changes that we observe during a transfer. This is because by changing the history data sets we are changing the data that the AS uses to determine the Next Advice Time (NAT) that it gives to the FTP server. From *Table 21* we observe that the number of inputs received from the AS is highest when we use the whole database as the history. The TCP CC state changes are also the highest when we use the whole database as the history data. We can thus observe that there are more redundant inputs from AS when we use the whole database as the history data and using the previous week's data as the history data decreases this number. By redundant inputs we mean the inputs from the AS, which do not change the congestion control state.

5.5.3.2. Tests in a moderately congested network

The parameters used for the tests in this section are listed in *Table 22* below.

TCP buffer size	1Mbyte
NISTNet delays	50ms
FTP transfer size	16Gbyte
NISTNet drops	YES
Available Bandwidth (ABW) Threshold	27Mbps (73% used bandwidth)

Table 22: Parameters for tests with different history data sets in a moderately congested network

The results of the tests with different history data sets for this case are shown in *Table 23* and *Table 24* below.

Time of Day	Throughput of FTP when run with AS		
	History = whole dB	History = Previous Fridays	History = Previous week
12:15am	25.60	25.68	25.68
3:25am	57.68	53.60	51.76
7:25am	70.00	69.92	70.00
2:25pm	40.24	42.00	43.92
9:10pm	43.84	45.84	55.28

Table 23: Throughput of FTP with AS when run in a network with a minimum of 60% used bandwidth and no background flows for different history data sets

Time of day	CC states received when FTP is run with AS		
	History = whole dB	History = Previous Fridays	History = Previous week
12:15am	1(12)	1(8)	1(5)
3:25am	1,0(9)	1,0(5)	1,0(4)
7:25am	0(5)	0(4)	0(2)
2:25pm	0(2),1(4),0(3)	0,1(2),0(2)	0,1,0
9:10pm	0(5),1(4)	0(3),1(2)	0(3),1

Table 24: CC State inputs received when FTP with AS is run in a network with a minimum of 60% used bandwidth and no background flows for different history data

In this case we observe from *Table 24* that the number of times the AS is contacted is highest when we use the whole database as the history data. But the number of TCP CC State changes is same for all the three history data sets. Also from *Table 23* we can see that there is not much difference in the observed throughput by changing the history data. Here again the number of redundant inputs is more when we use the whole database as the history data.

5.5.3.3. Tests in a highly congested network

The parameters used for the tests in this section are listed in *Table 25* below.

TCP buffer size	1Mbyte
NISTNet delays	50ms
FTP transfer size	16Gbyte
NISTNet drops	YES
Available Bandwidth (ABW) Threshold	12Mbps (78% used bandwidth)

Table 25: Parameters for tests with different history data sets in a highly congested network

The test results of the tests with different history data sets are shown in *Table 26* and *Table 27* below.

Time of Day	Throughput of FTP when run with AS		
	History = whole dB	History = Previous Fridays	History = Previous week
12:15am	16.88	17.68	17.52
3:25am	58.40	57.04	56.88
7:25am	68.56	69.44	69.52
2:25pm	34.72	30.72	28.08
9:10pm	29.28	34.16	38.08

Table 26: Throughput of FTP with AS when run in a network with a minimum of 75% used bandwidth and no background flows for different history data sets

Time of day	CC states received when FTP is run with AS		
	History = whole dB	History = Previous Fridays	History = Previous week
12:15am	1(46),0(23)	1(27),0(10)	1(22),0(15)
3:25am	1,0(48)	1,0(26)	1,0(24)
7:25am	0(18)	0(10)	0(6)
2:25pm	0(7),1(9),0(21),1(3)	0(3),1(6),0(7),1,0(3)	0(2),1(3),0(5),1(2),0(3)
9:10pm	0(24),1(6),0(4),1(9)	0(14),1(5),0(4),1(4)	0(12),1(3),0(2),1(2)

Table 27: CC State inputs received when FTP with AS is run in a network with a minimum of 75% used bandwidth and no background flows for different history data

From *Table 26* we observe that in this case also there is not much difference in the throughput of FTP with different history data sets. The number of inputs from the AS is again highest when we use the whole database as the history. The number of TCP CC state changes is same for all the three history data sets except at 2:25pm at which, with the history data sets of all Fridays and last one week prior to the test day, there is one additional TCP CC state change. Contacting the AS more frequently can effect its performance, especially when there are a lot of clients to be monitored. Hence using the most recent network data as the history data can serve the AS well.

From the results of tests to evaluate the performance of the Dynamic TCP Congestion Control Scheme in the Enable service, we observe that, the FTP transfer times are reduced by an average of *78.30%* when FTP is run with AS. This is because we use TCP with NOCC when there is not much congestion in the network and this increases its throughput. We can also observe that, by using AS we are making sure that Congestion Control (CC) in TCP is turned off only when required instead of turning it off totally. Also we observe that as the percentage of used bandwidth increases, the percentage improvement in the FTP throughput increases. This is because, as the percentage of used bandwidth increases the bandwidth available for the standard FTP decreases. But FTP with AS will not slow down in this case and its throughput remains the same and hence the percentage increase in the throughput of FTP is higher.

From the results of tests with background flows we observe that, when FTP is run with AS, the effect on the throughput of the background traffic is less when compared to running FTP with NOCC. This shows that by not turning off the Congestion Control in TCP totally, we are trying to be less aggressive on the background traffic. From the tests with multiple background flows we observe that the effect on both the background flows is similar.

From the results of tests with different history data sets, we observe that the AS is contacted more frequently when we use the whole database as the history database. Since there is not much difference in the throughput of FTP by changing the history data, it is better to use whole database as the history data.

All the test results show that the Dynamic TCP Congestion Control Scheme is implemented correctly in the Enable service and by properly using the AS with the FTP server we can reduce the transfer times of large files on High Bandwidth Delay Product networks. Also the mechanism is not totally TCP friendly and hence the background traffic is effected to some extent.

Chapter 6

Conclusions and Future work

6.1. Conclusions

TCP Congestion Control algorithms have been designed to avoid congestion collapse in the networks. It performs well on the low delay links but on high delay links it has a bad performance. Experimental modifications were done to the TCP stack such that applications can turn off the congestion control in TCP. But turning off the congestion control totally is not advisable. Hence a mechanism has been designed so that congestion control is turned off only when required depending on the network congestion and transfer file size.

We have successfully implemented the Enable service. This Enable service decides when to change the congestion control state of TCP based on the network conditions. ProFTPD, a widely used FTP server daemon, was modified to interact with the Enable service during large file transfers. It dynamically changes the TCP Congestion Control State based on the inputs from the Enable service. By not totally turning off the congestion control in TCP we were less aggressive and thus the impact on the background traffic was reduced. Also we were able to emulate the WAN conditions in a local environment by using NISTNet, a network emulation tool.

From the results of the tests conducted here with this dynamic congestion control mechanism, it was found that we could reduce the transfer times for large file transfers. Also this mechanism is shown to behave in a less aggressive manner on the background traffic than when the congestion control in TCP is turned off totally.

6.2. Future Work

The first enhancement that can be made is testing the mechanism on a real WAN environment instead of on an emulated network. This can determine the performance benefit of the mechanism. The next enhancement can be with the way the Advisory Server decides about the TCP congestion control state to use. Currently it bases its decision only on the current available bandwidth on the network path. But other parameters can be used in addition, to make a better decision. Instead of using the data collected from a router, we could use *pipechar* [25] to collect the network state and use it to determine if there is congestion in the network path or not.

References

[1] M.Allman, V.Paxon, W.Stevens. TCP Congestion Control, RFC 2581.

[2] Anupama Sundaresan. Application Level Congestion Control Enhancements for High Bandwidth Delay Product Networks, Master of Science Thesis, University of Kansas, June 2000, http://www.ittc.ukans.edu/projects/enable/anu_thesis.pdf

[3] B. Tierney, D. Gunter, J. Lee, M. Stoufer, J. B. Evans, "Enabling Network-Aware Applications", 10 th IEEE Symposium on High Performance Distributed Computing, August 2001.

[4] XueJun Mao, Victor Frost, Joseph Evans, and Mahesh Akarapu, "Reducing the Transfer Time for Large Files in High Performance Networks", Sept. 2002, ITTC-FY2003-20430-01.

[5] Sally Floyd. Congestion Control Principles. Internet-Draft draft-floyd-cong-02.txt, April 2000.

[6] Van Jacobson, Robert Braden, David Borman. TCP Extensions for High Performance, May 1992. RFC 1323.

[7] Hans Kruse. Performance of Common Data Communications Protocols Over Long Delay Links: An Experimental Examination. In 3rd International Conference on telecommunication Systems Modeling and Design, 1995.

[8] TCP Tuning Guide for Distributed Application on Wide Area Networks, February 2001. http://www-didc.lbl.gov/tcp-wan.html

[9] PSC: Enabling High Performance Data Transfers on Hosts, September 1999. http://www.psc.edu/networking/perf_tune.html

[10] Net100: Development of Network-Aware Operating systems, December 2000. http://www.net100.org

[11] "The WEB100 Project, Facilitating Effective and Transparent Network Use", http://www.web100.org/

[12] Auto-tuning in Linux 2.4 kernels http://www.linuxhq.com/kernel/v2.4/doc/networking/ip-sysctl.txt.html

[13] Sivakumar, H, S. Bailey, R. L. Grossman. "PSockets: TheCase for Application-level Network Striping for Data Intensive Applications using High Speed Wide Area Networks", Proceedings of IEEE Supercomputing 2000, Nov. 2000. http://www.ncdm.uic.edu/html/psockets.html

[14] NISTNet: A network emulation tool developed by the NIST, September 1997. http://snad.ncsl.nist.gov/itg/nistnet/

[15] ProFTPD: Highly configurable FTP server software http://www.proftpd.org/

[16] NcFTP - Browser program for the File Transfer Protocol http://www.ncftpd.com/ncftp/

[17] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson. Iperf – The TCP/UDP Bandwidth measurement tool. http://dast.nlanr.net/Projects/Iperf/

[18] Douglas E. Comer. Internetworking with TCP/IP, Volume I, Principles, Protocols, and Architecture. Prentice Hall, 3rd edition, 1995.

[19] Larry L.Peterson & Bruce S. Davie. Computer Networks: A Systems Approach, 2nd edition, 2000, Morgan Kaufmann Publishers, Inc.

[20] Jon Postel. Transmission Control Protocol, September 1981. RFC 793.

[21] Victor O.K. Li, Zaichen Zhang. "Internet Multicast Routing and Transport Control Protocols", Proceedings of the IEEE, Vol. 90, No.3, March 2002

[22] J.Postel, J.Reynolds. File Transfer Protocol, October 1985. RFC 959.

[23] J.Postel, J.Reynolds. Telnet Protocol Specification, May 1983. RFC 854.

[24] ProFTPD Developer's Guide by TJ Saunders http://www.castaglia.org/proftpd/doc/devel-guide/

[25] Jin, G., Yang, G., Crowley, B., Agarwal, D., "Network Characterization Service", Proceedings of the IEEE High Performance Distributed Computing conference, August 2001, http://www-didc.lbl.gov/NCS/

[26] Van Jacobson. Packet Sniffing tool. http://www.tcpdump.org

[27] Shawn Ostermann. tcptrace - TCP dump file analysis tool. http://jarok.cs.ohiou.edu/software/tcptrace/tcptrace.html

[28] MySQL: Open source database
http://www.mysql.com

[29] Tim Shepard. xplot - A Plotting Tool, February 1991.
ftp://mercury.lcs.mit.edu/pub/shep

[30] Chervenak, A., Foster, I., Kesselman, C., Salisbury, C. and Tuecke, S. "The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Data Sets". Journal of Network and Computer Applications, 2000.

[31] XML-RPC: http://www.xmlrpc.org/

[32] D. Gunter, B. Tierney, B. Crowley, M. Holding, J. Lee NetLogger: A Toolkit for Distributed System performance Analysis, Proceedings of the IEEE Mascots 2000 Conference, August 2000. http://www-didc.lbl.gov/NetLogger/

[33] Bruce A. Mah. Pchar: A Tool for Measuring Internet Path Characteristics, February 1999. http://www.employees.org/~bmah/Software/pchar/

[34] Internet2: www.internet2.edu

Appendix

Appendix A

A.1. Specifications of the machines used in the test scenarios

Machine Name	Specification
qost1	Pentium II 400MHz
qost2	Pentium II 400MHz
qost3	Pentium II 400MHz
qost4	Pentium II 400MHz
Neuromancer	Celeron 434 MHz(Dual processor)
Straylight	Celeron 467 MHz(Dual processor)
testbed33	Pentium III 1GHz
testbed34	Pentium III 1GHz

Table 28: Specifications of the machines used in the test scenarios

Appendix B

B.1. Using NISTNet

NISTNet has a graphical user interface, which allows the user to select and monitor specific traffic streams passing through the router and to apply selected performance effects to the IP packets of the stream. It also provides a command line interface to be able to generate scripts, so that it can be driven by traces produced from measurements of actual network conditions. The command-line interface used to generate scripts used in the tests is ***cnistnet.*** Its usage is shown below:

cnistnet –u turns the NISTNet emulator ON and *cnistnet –d* turns the emulator OFF.

Once the emulator is turned ON, we can identify the flow to which to apply the WAN effects using various filters such as the source and destination addresses, source and destination ports. To all such identified flows we can apply the WAN effects using the following command [14].

cnistnet -a src[:port[.protocol]] dest[:port[.prot]] [cos]
[--delay delay [delsigma[/delcorr]]]
[--drop drop_percentage[/drop_correlation]]
[--dup dup_percentage[/dup_correlation]]
[--bandwidth bandwidth]
[--drd drdmin drdmax [drdcongest]]

The WAN effects that we can add using the above command are the delay, packet drop rate, packet duplication rate, and the bandwidth to which to limit the flow.

To remove the NISTNet filter we can identify the flow and remove it using the following command.

cnistnet -r src[:port[.prot]] dest[:port[.prot]] [cos]

To view the statistics of the packets of the flow identified by the NISTNet filters, we use the following commands.

cnistnet -s src[:port[.prot]] dest[:port[.prot]] [cos]

cnistnet -S src[:port[.prot]] dest[:port[.prot]] [cos]

B.2. NISTNet script to dynamically change the drop rate of the traffic

```
/* Program to run a script which dynamically adjusts the drop rates along
   a path using the cnistnet command. */


#include<stdio.h>

main(int argc,char *argv[])
{
 char sec[10],drop[10],temp_drop[10];
 char *str =(char *)malloc(100);
 char *str1 =(char *)malloc(50);
 char *str2 =(char *)malloc(100);
 FILE *fin;

 fin = fopen(argv[1],"r");

 strcpy(str2,"cnistnet -r 192.168.124.2 192.168.126.6 --drop ");

 system("cnistnet -u");

 //Delays in the reverse direction for the FTP flow
 system("cnistnet -a 192.168.126.6 192.168.124.2 --delay 25");

 //Delays in the forward and reverse directions for the first Iperf flow
 system("cnistnet -a 192.168.125.1 192.168.122.4 --delay 5");
 system("cnistnet -a 192.168.122.4 192.168.125.1 --delay 5");

 //Delays in the forward and reverse directions for the second Iperf flow
 system("cnistnet -a 192.168.128.33 192.168.127.34 --delay 5");
 system("cnistnet -a 192.168.127.34 192.168.128.33 --delay 5");

 while (fscanf(fin,"%s",drop)!= EOF)
 {
   strcpy(temp_drop,drop);

   if(fscanf(fin,"%s",sec)== EOF)
   {
     printf("Error Reading from the input file\n");
     exit(1);
   }
```

```
    strcpy(str1,"sleep ");
    strcat(str1,sec);

    strcpy(str,"cnistnet -a 192.168.124.2 192.168.126.6 --delay 25 --drop ");
    strcat(str,drop);

    system(str);
    system(str1);

  }
 strcat(str2,temp_drop);

 system(str2);
 system("cnistnet -d");

  fclose(fin);
}
```

B.3. Script used to generate the drop rates data from the available bandwidth data

```
/* Program to convert the input bandwidth values into the corresponding drop rates */

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

#define C      9.76            //1.22*8
#define S      1076
#define RTT    50             //milliseconds
#define FACTOR 1000   //mbps * msec

main(int argc,char **argv)
{
 FILE *fin,*fout;
 float bw,temp,drop_rate;
 int sec;

 fin  = fopen(argv[1],"r");
 fout = fopen(argv[2],"w");

 while (fscanf(fin,"%f",&bw)!= EOF)
 {
```

```
  if(fscanf(fin,"%d",&sec)== EOF)
  {
    printf("Error Reading from the input file\n");
    exit(1);
  }

  temp = (C*S*8)/(RTT*bw*FACTOR);
  drop_rate = pow(temp,2);

  fprintf(fout,"%f %d\n",drop_rate,sec);

 }

 close(fin);
 close(fout);

}
```

Appendix C

C.1. Commonly used data structures of the ProFTPD

a) module_struc

Declaration:

```
struct module_struc {
  module *next, *prev;

  /* module API version  */
  int ver;

  /* module name */
  char *name;

  /* configuration directive table */
  conftable *conftable;

  /* command handler table */
  cmdtable *cmdtable;
```

```
  /* authentication handler table */
  authtable *authtable;

  /* module initialization */
  int (*module_init)();

   /* post-fork initialization */
   int (*module_init_child)();

   /* internal use, greater number == higher priority */
   int priority;
};
```

Source File: include/modules.h

b) cmd_rec

Declaration:

```
typedef struct cmd_struc {

     /* memory pool for this object */
  pool *pool;

  server_rec *server;
  config_rec *config;

  /* temporary pool which only exists while the cmd's handler is running*/
  pool *tmp_pool;

  int argc;
  char **argv;

  /* entire argument (excluding command) */
  char *arg;

  /* command group */
  char *group;

  /* command class */
  int class;
```

```
  /* hack to speed up symbol hashing in modules.c */
  int symtable_index;

  /* private data for passing/retaining among handlers */
  privdata_t *private;

  /* internal use */
  array_header *privarr;

 } cmd_rec;
```
SourceFile:include/dirtree.h

c) server_rec

Declaration:

```
 typedef struct server_struc {

  struct server_struc *next, *prev;

     /* memory pool for this object */
  pool *pool;

  /* set holding all the servers */
  xaset_t *set;

  /* this server's name */
  char *ServerName;

  /* this server's address */
  char *ServerAddress;

  /* this server's fully qualified domain name */
  char *ServerFQDN;

  /* this server administrator's name */
  char *ServerAdmin;

  /* this server's welcome message */
  char *ServerMessage;

  /* port number for this server */
  int ServerPort;
```

```
    /* receive/send windows */
    int tcp_rwin, tcp_swin;

    /* specifically override the TCP rwin */
    int tcp_rwin_override;

    /* specifically override the TCP swin */
    int tcp_swin_override;

    /* do not greet until after the user's logged in */
    int AnonymousGreeting;

    /* internal address of this server */
    p_in_addr_t *ipaddr;

    /* our listening connection */
    struct conn_struc *listen;

    /* configuration details */
    xaset_t *conf;

  } server_rec;
```

SourceFile:include/dirtree.h