# *Reducing Latency on the Internet using*

# *"Component-Based Download"*

# *and*

# *"File-Segment Transfer Protocol"*

**By Babak S. Noghani**

A Thesis submitted to the Faculty of Graduate Studies
Of the University of Manitoba
In Partial Fulfillment of the Requirements for the Degree of

***MASTER OF SCIENCE***

Department of Electrical & Computer Engineering
University of Manitoba
Winnipeg, Manitoba, Canada

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

**Reducing Latency on the Internet using "Component-Based Download"**

**and "File-Segment Transfer Protocol"**

**BY**

**Babak S. Noghani**

**A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree**

**of**

**Master of Science**

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research.

---

Babak Noghani

I further authorize the University of Manitoba to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

---

Babak Noghani

The University of Manitoba requires the signatures of all persons using or photocopies this thesis. Please sign below, and give address and date.

# ABSTRACT

This thesis examines the Component-Based Download (hereinafter referred to as "CBD") and the Component-Based Download File-Segment Transfer Protocol (hereinafter referred to as "CBD-FSTP") to combat the latency due to the slow performance of the current file transfer protocols. In particular, we focus on accelerating the download of large files (e.g.: video) from the Internet. This can be achieved by the following two methods:

1. Defining a distributed server mechanism for transferring data known as CBD; and
2. Integrating a new distributed file transfer protocol known as CBD-FSTP

More efficient utilization of bandwidth can be obtained by using these two methods. This will lead to a higher throughput and thus a reduced latency. The trade off will be higher processing overhead and network utilization that are no longer major drawbacks because of the emergence of fast computers with increasing processing power and the expansion of fiber optic Gigabit links.

According to the data gathered throughout extensive measurements, CBD can speed up the process of downloading large-size files up to 3 times faster compared to conventional methods. The CBD-FSTP can also improve the latency by additional 30% compared to the FSTP method, which is another fast newly-developed file transfer protocol. Results show that CBD-FSTP can be 7 times faster than the File Transfer Protocol (hereinafter referred as "FTP").

# ACKNOWLEDGMENTS

I would like to extend my thanks to my advisor, Dr. Robert D. McLeod for his support and guidance throughout the entirety of my academic work on the CBD and CBD-FSTP. I also want to thank Dr. David Blight for his support and encouragement when I started Master's program. Finally, I want to thank Steve Kretschmann for his commitment over the course of implementing the FSTP protocol. Of course there are those who go unnamed, and to them I offer my gratitude as well.

## Acronyms

| Acronym | Definition |
|---|---|
| ACK | Acknowledgement |
| ARDP | Asynchronous Reliable Delivery |
| CBD | Component-Based Download |
| DSL | Digital Subscriber Line |
| FSTP | File Segment Transfer Protocol |
| FTP | File Transfer Protocol |
| HTTP | HyperText Transport Protocol |
| IP | Internet Protocol |
| IPTD | Inter-Packet Transmission Delay |
| LAN | Local Area Network |
| MTU | Maximum Transferable Unit |
| NNTP | Network News Transfer Protocol |
| RAM | Random Access Memory |
| RTO | Retransmission Time Out |
| RTT | Round-trip time |
| TCP | Transport Control Protocol |
| TFTP | Trivial File Transfer Protocol |
| TTL | Time To Live |
| UDP | User Datagram Protocol |

**Figure 1-1 Acronyms**

## Table of Contents

# List of Figures



# List of Tables

# Chapter One

# 1 Introduction

## 1.1 Thesis Motives

The Internet has seen tremendous growth within the last few years. The main reason for its popularity is its ability to provide easy user access to a wide variety of data from remote locations. This data can be of any nature and of any size, from small text files to very large movies. Literally, there are no boundaries as to the size of the files that can be accessed through the Internet.

In spite of the Internet's tremendous capabilities, Internet users tend to complain about the time they waste working on their computers, waiting for data to download. This concern has grown larger with the emergence of audio and video files over the Internet. Files with several hundred megabytes are not unusual these days. In other words, for a typical Internet user, latency is a concern, and reducing this delay can be a great help for Internet users. Finding a quicker and more efficient way to download files from the Internet is the primary reason for this thesis.

## 1.2 Challenges

A number of approaches have been taken in minimizing user latency on the Internet. To name a few, increasing the bandwidth on the Internet backbone and to the end user, more efficient routing schemes, streaming data, etc. None of which has effectively solved the latency problem.

Latency is caused by a number of sources. For instance, if a server is overloaded or has a slow disk, it imposes a considerable delay in processing a request. Another such instance is caused when a user's computer does not quickly respond to the packets being received and therefore, adds delay. The latency caused by a server or client can be largely eliminated by using a more powerful computer, more memory, or a faster disk.

The main portion of the latency perceived by an Internet user is caused by the network itself. Some sources of this delay are intrinsic to the network infrastructure, namely propagation and transmission delays. Propagation delay depends on the speed of light and is negligible compared to other delay factors. Transmission delay is not a big concern anymore. The reason for this is that most Internet users nowadays have access to reasonably high-speed Internet connections.

Delay can also be due to network congestion. High performance routing algorithms and a fast and reliable network infrastructure can alleviate this problem to a great extent. Routers, as well, contribute to the delay perceived by the user due to the buffering and processing time involved in routing the Internet packets.

Yet another source of latency can be due to the design and implementation of the Internet protocols themselves. As a common rule, file transfer protocols were

initially designed to match particular network characteristics with the type or size of the data that is transmitted. Therefore, with the evolving nature of the Internet, the protocols are modified occasionally to optimize performance. Many modifications to the existing protocols, namely Hyper Text Transport Protocol HTTP [Ber96], have been proposed in the literature to reduce the latency. Some of them have already been tested and implemented on the Internet to a great extend. The more significant modifications are listed as follows:

- Avoiding the cost of Round Trip Time (RTT) by reducing the number of HTTP connections. This method uses a single, long-lived connection for multiple HTTP transactions (persistent connection). The connection remains open for all the inline images of a single document, and across multiple HTML retrievals [Pad94].
- Utilizing multiple Transport Control Protocol (TCP) connections to the server. This technique is currently used by web browsers that comply with the HTTP1.1 [Fie97]. Instead of opening and closing a connection for each application request, HTTP 1.1 provides a persistent connection that allows for multiple requests to be batched or pipelined to an output buffer. The underlying TCP layer can put multiple requests (and responses to requests) into one TCP segment that is forwarded to the Internet Protocol (IP) layer for packet transmission. Because the number of connection and disconnection requests for a sequence of "get a file" requests is reduced, fewer packets need to flow across the Internet. Since more requests are pipelined, TCP segments become more efficient. The result being less

Internet traffic and faster performance for the user. When a browser supporting HTTP 1.1 indicates it can decompress HTML files, a server will compress them for transport across the Internet, providing a substantial aggregate savings in the amount of data that is being transmitted. (Image files are already in a compressed format so this improvement applies only to HTML and other non-image data types.)

- Pre-fetching techniques attempt to predict future requests of a user, based on the history of observed Web pages. Pre-fetching can reduce network delays considerably [Cro98]. There are many solutions toward making the pre-fetching techniques more efficient. They are described throughout literature. Most of these solutions try to deploy a learning algorithm by which proxies would be able to pre-fetch files that are most likely accessed next by the Internet user.
- Using mirror/replicated servers, i.e.; spreading the workload among a cluster of servers rather than a single machine handling the HTTP requests. Server replication is an approach often used to improve the ability of a service to handle a large number of clients. The most important factor in efficient utilization of replicated servers is the ability to direct client requests to the best server, according to some optimal criteria. This issue is discussed further in [Fei98].

As mentioned earlier, the Internet has rapidly evolved more so in recent years. One of its biggest improvements involves increased bandwidth availability. This

possibility has resulted due to the use of fiber networks, DSL, cable connections, and other new technologies. A typical Internet user now has abundant bandwidth available to their computers compared to a few years ago. However, in most cases, the bandwidth cannot be utilized efficiently. Experience has shown that when using a typical file transfer protocol to download a file, only a portion of the available bandwidth is utilized. Current file transfer protocols use one of the versions of sliding-window mechanism for providing reliable connection. In the sliding-window mechanism [1] the sender should wait for an acknowledgment from the receiver before sending the next segment of data. The amount of time before the sender receives the acknowledgment is referred to as Round-Trip Time (RTT). In high-speed network, RTT can be even larger than the transmission time. Therefore, the actual throughput of a connection is limited by the RTT. This degrades the efficient utilization of the bandwidth by current file transfer protocols.

The objective of this thesis is to expand downloading capabilities of large files over the Internet as quick as possible. This can be achieved by defining a distributed server mechanism for transferring data. And secondly, by designing a new file transfer protocol, which is compatible with our distributed mechanism.

---

[1] Refer to Appendix 7.1 for more information

The idea behind these two solutions is to utilize the bandwidth more efficiently and eliminate the negative effect of RTT. This would lead to higher throughput and shorter latency perceived by the Internet user. Of course, this enhancement is achieved as a trade off with additional processing overhead and network costs. Fortunately, this is not a significant drawback due to the emergence of new computers with ever-increasing processing power and fiber optic gigabit links.

## 1.3 Structure of Thesis

In the following chapter a new distributed server mechanism (CBD) will be introduced. In chapter three, a new file transfer protocol (FSTP) will be introduced and explained in greater detail. Following that, in chapter four, the CBD paradigm would be integrated into the FSTP and a distributed version of this new protocol (CBD-FSTP) would be designed. The prototype implementation of CBD-FSTP will also be covered in that chapter. Chapter five focuses on the conclusion and discusses future research and direction.

# Chapter Two

# 2 Component-Based Download

## 2.1 Motivation for CBD

As mentioned earlier, bandwidth is not a bottleneck in today's networks any longer. TCP, as the de facto file transfer protocol in the Internet, has not properly adapted itself to this improvement. Experiments have shown that TCP's sliding window mechanism for flow control and slow start algorithm for congestion control[2], causes limited throughput particularly for a high-speed network connection.

Although numerous changes have been applied to TCP to allow for transferring data quicker, this protocol is still considered quite slow. This is due to its initial design, which is based on the assumption that in every given network, packet loss ratio should be considerable and bandwidth relatively low. To be more specific, overly conservative Retransmission Timeout (RTO), and inability to measure the available bandwidth accurately, degrades the TCP service throughput [Hoe96][Bra95].

The initial concept of CBD is closely related to what already has been utilized in the implementation of HTTP1.1 [Fie97]. HTTP1.1 takes advantage of the concept of long-lived TCP connections. That is, several logical data streams are

---

[2] Refer to appendix 7-1 for more information

multiplexed by the application into one TCP socket. This concept has been extensively addressed in the literature over the last few years. Examples include persistent-control HTTP and Session Control Protocol (SCP) [Pad94].

In CBD, quite similar to HTTP1.1, the user establishes multiple concurrent connections. The only difference is that these connections are linked to different servers, not just one. The distributed approach (i.e., using several servers) is intentionally chosen for the CBD because it increases fault tolerance of the downloading process. (This issue will be discussed in greater details in the following sections.) Another concept that deserves consideration is that in HTTP1.1 the sizes of the transferred files are generally quite small and connections are mediated through the web server. Whereas, CBD deals with substantially larger files.

The concept of CBD is very simple and straightforward. For a user to be able to download a file via CBD, the file should first be divided into several components and put on different servers. When the user attempts to download the file, four components are identified and they are as follows:

1. Multiple concurrent connections will be established between the user's application and the servers maintaining the file components.
2. As soon as each connection has been set up, the file component will start downloading independently from the others. In other words, all the components will be downloaded simultaneously.

3. On the user's end, each connection will be terminated after the respective component has been completely downloaded.
4. Finally, the components on the user's machine will get appended to each other and reconstitute the original file.

The following illustrates the connections between the user's machine and the multiple servers.

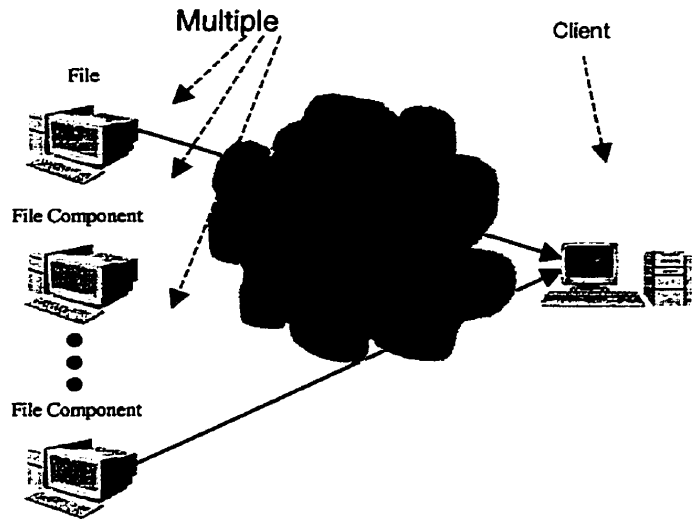


**Figure 2-1 CBD Distributed Server Mechanism**

### 2.1.1 Proof of Concept

As the first step in verifying the feasibility of the CBD mechanism, some experiments were performed. During these experiments files were downloaded once via the CBD, and once using the conventional method. After each trial, the latency was measured and a comparison was drawn. The CBD was simulated as follows:

- Since the FTP is the de-facto application protocol for downloading large files over the Internet, we used it in our tests for transferring files.
- Several files located on remote FTP servers were selected. These files were downloaded first, by establishing concurrent (all simultaneously) FTP sessions between an FTP client and the servers, and secondly, by establishing sequential (one after another) FTP sessions between the same FTP client and the servers.
- The total delays for downloading the files were measured in both cases. In concurrent download, total perceived latency is equal to the greatest value of the latencies measured for each individual download. In sequential download, the perceived latency is equal to the sum of the latencies measured for each individual download.

Needless to say that in a real implementation of CBD, components of a file should be downloaded instead of standalone files. On the client's end the components should be appended to each other to reconstitute the original file.

In this chapter the main concern is to verify that the CBD is quicker than conventional download methods. Therefore, the focus will be merely on measurement of latencies. In chapter five a file transfer protocol will be introduced, which CBD can be practically incorporated into.

#### 2.1.1.1 CBD Results

Our tests clearly illustrated the great potential to improve the performance of a network by using the CBD for large files. Depending on what network connection, what level of distribution, and what component size were used, the downloading time was 1 and a half to 3 times faster.

To repeat the experiment a reasonable number of times, a Java program was written. This program automates the process of opening the FTP sessions, measuring and collecting the latencies.[3] Over the course of experiments, different contingencies that might occur in real-life situations were taken into consideration. To compensate for the effect of variations of the traffic load on the network and to get a reliable result, experiments were repeated at various times throughout the day on certain days of the week. To investigate the performance of CBD and find out its optimum state, various environments and different numbers and sizes of files were tested.

The experiments were performed over campus type LANs, modems, and wireless LANs supporting 802.11. The following machines were used in the experiments:

- For connecting to 100BaseT: Sun Sparc Ultra 10, 768 Meg of RAM, running Solaris 5.6

[3] Refer to appendix 7.4.1 for its source code

- For connecting to 10BaseT: Sun Sparc Ultra 2, running Solaris
- For connecting to wireless LAN: Intel Pentium II, running Windows 98 with 32 Meg of RAM

The results obtained under each environment are provided in the following sub-sections.

#### 2.1.1.1.1 100BaseT LAN

Experiments were performed on three different sizes and numbers of files:

- Three files, with sizes 1.1, 1.3 and 2 Meg, were downloaded. Measurements were repeated one hundred times. The mean latency while utilizing the CBD was 85 seconds as compared to 176 seconds for the conventional download. (Figure 2-2)

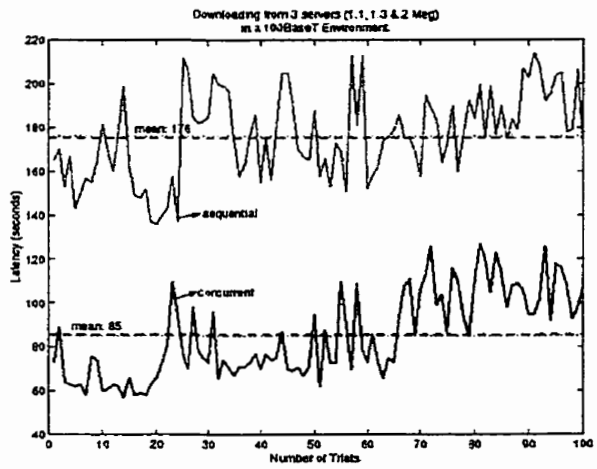


**Figure 2-2 Downloading Three Files (1, 1.3 & 2 Meg) in a 100BaseT LAN Environment**

- Three files, with sizes 7.9, 8 and 9 Meg, were downloaded. Measurements were repeated ninety times. The mean latency while utilizing the CBD was

317 seconds as compared to 697 seconds for the conventional download (Figure 2-3).

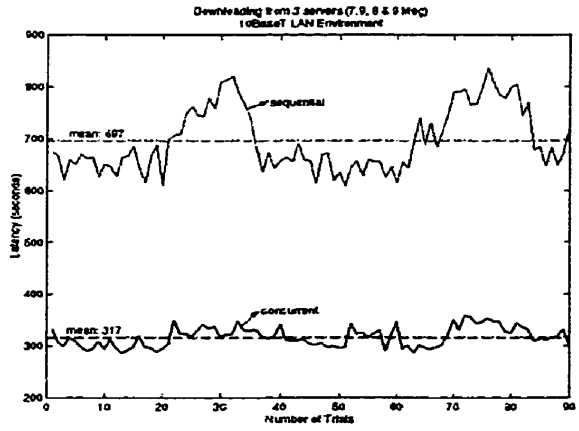


**Figure 2-3 Downloading Three Files (7.9, 8 & 9 Meg) in a 100BaseT LAN Environment**

- Three files, with sizes 50.8, 33.7 and 34.9 Meg, were downloaded. Measurements were repeated twenty times. The mean latency while utilizing the CBD was 1,035 seconds as compared to 2438 seconds for the conventional download (Figure 2-4).

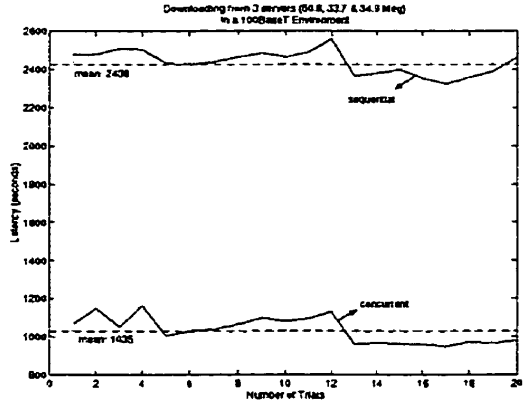


**Figure 2-4 Downloading Three Files (50.8, 33.7 & 34.9 Meg) in a 100BaseT LAN Environment**

- Five files, with sizes 1.1, 1.3, 1.8, 2 and 2.8 Meg, were downloaded. Measurements were repeated one hundred and twenty times. The mean latency while utilizing the CBD was 91 seconds as compared to 267 seconds for the conventional download. (Figure 2-5)

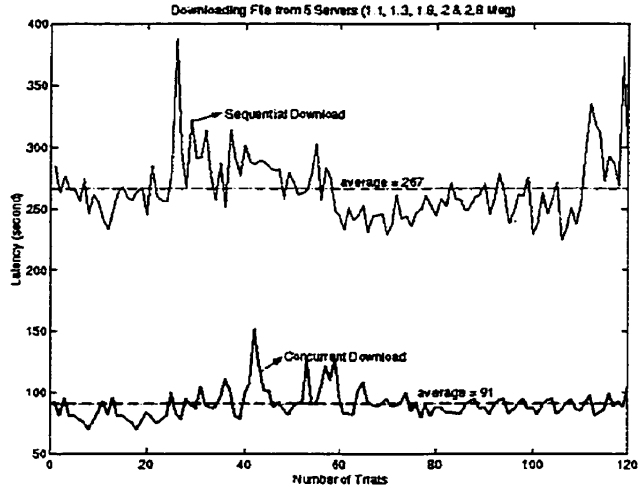


**Figure 2-5 Downloading Five Files (1.1, 1.3, 1.8, 2 & 2.8 Meg) in a 100BaseT LAN Environment**

The results show higher improvements in latency, when larger files get downloaded, and the number of participating servers is increased (from 3 to 5).

### 2.1.1.1.2 10BaseT LAN

Experiments were performed on two different sizes and numbers of files:

- Three files, with sizes 1.1, 1.3 and 2 Meg, were downloaded. Measurements were repeated one hundred times. The mean latency while utilizing the CBD was 47 seconds as compared to 99 seconds for the conventional download (Figure 2-6).

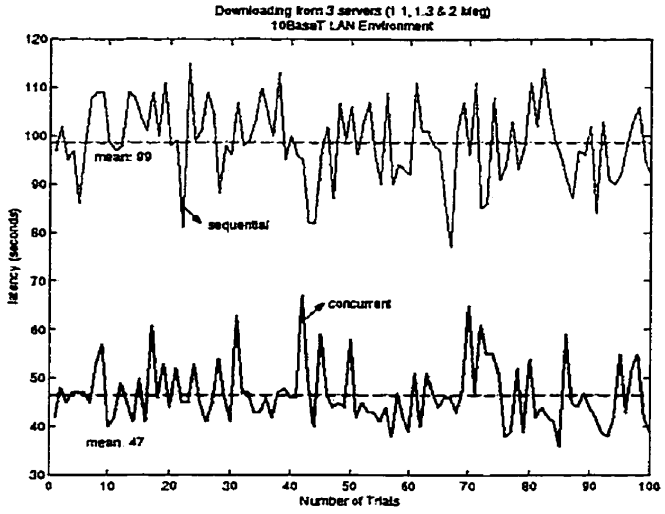


**Figure 2-6 Downloading Three Files (1.1, 1.3 & 2 Meg) in a 10BaseT LAN Environment**

- Five files, with sizes 1.3, 1.8, 2.2, 2.3 and 2.5 Meg, were downloaded. Measurements were repeated fifty times. The mean latency while utilizing the CBD was 59 seconds as compared to 151 seconds for the conventional download (Figure 2-7).

As the results imply, once again, increased improvement of latency was achieved when the number of participating servers was increased.

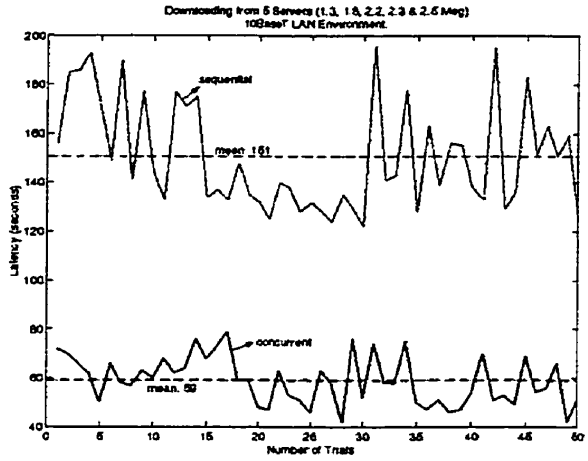


**Figure 2-7 Downloading Five Files (1.3, 1.8, 2.2, 2.3 & 2.5 Meg) in a 10BaseT LAN Environment**

### 2.1.1.1.3 Wireless LAN

Experiments were performed on 3 different sizes and numbers of files:

- Three files, with sizes 1.1, 1.3 and 2 Meg, were downloaded. Measurements were repeated ninety times. The mean latency while utilizing the CBD was 89 seconds as compared to 194 seconds for the conventional download (Figure 2-8).
- Three files, with sizes 6.6, 8.1 and 11.1 Meg, were downloaded. Measurements were repeated thirty times. The mean latency while utilizing the CBD was 396 seconds as compared to 534 seconds for the conventional download. (Figure 2-9).
- Five files, with sizes 1.1, 1.3, 1.8, 2 and 2.8 Meg, were downloaded. Measurements were repeated one hundred twenty times. The mean

latency while utilizing the CBD was 103 seconds as compared to 317 seconds for the conventional download. (Figure 2-10).

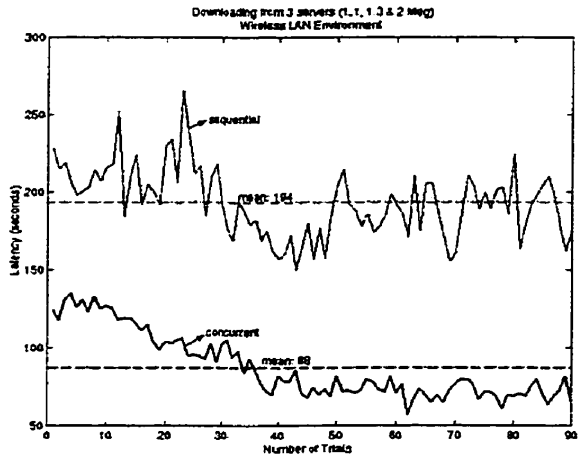


**Figure 2-8** **Downloading Three Files (1.1, 1.3 & 2 Meg) in a 10BaseT Wireless Environment**

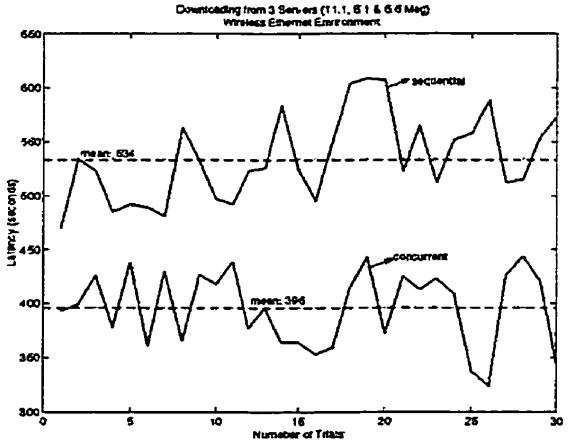


**Figure 2-9** **Downloading Three Files (6.6, 8.1 & 11.1 Meg) in a 10BaseT Wireless Environment**

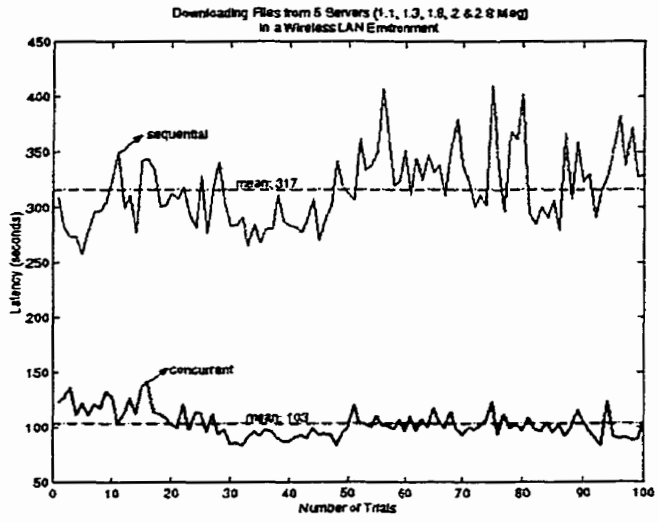


**Figure 2-10 Downloading Five Files (1.1, 1.3, 1.8, 2 & 2.8 Meg) in a 10BaseT Wireless Environment**

The same as the last two environments, results indicate an increase in the number of concurrent connections improves the relative latency. When the sizes of the files were increased, the performance of CBD deteriorates. This result is due to the lack of enough Random Access Memory (RAM) in the machine used in a wireless environment. When FTP downloads a file, it is temporarily saved on the RAM, before the Operating System allocates permanent storage space on the hard disk. The machine utilized in the wireless environment, has only 32 MB of RAM, which is not enough to handle the process of saving large files. By utilizing a machine with more RAM this problem would be alleviated.

#### 2.1.1.1.4 Dialup Modem

Three files, with sizes 1.1, 1.3 and 2 Meg, were downloaded. Measurements were repeated thirty times. The mean latency while utilizing the CBD was 1578 seconds as compared to 1493 seconds for the conventional download. The results imply that no improvement was achieved through utilization of the CBD in this case (Figure 2-11).

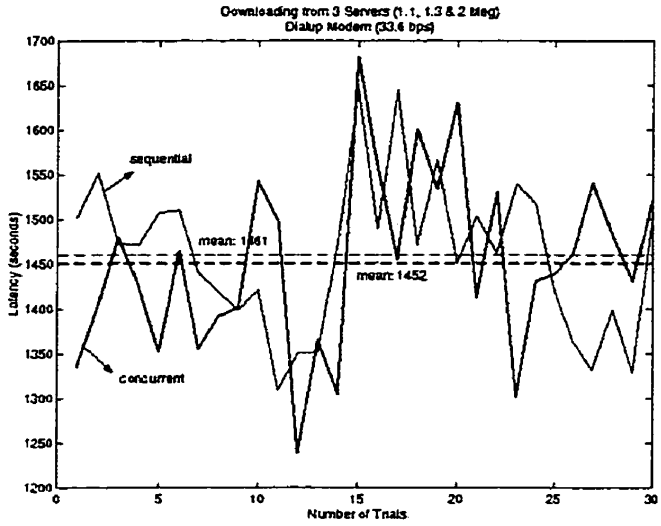


**Figure 2-11 Downloading 3 Files (1.1, 1.3 & 2 Meg) in a Modem Environment (33.6 bps)**

Even before performing the experiments on dialup modem connection, one could predict the same results (i.e., no improvement). One of the basic assumptions made in the design of CBD, is that the user machine should be connected through a high-speed link. Having this wealth of bandwidth allows the user to spare some of the bandwidth to the overhead of creating concurrent network connections. Whereas in a dialup modem environment, the bandwidth is scarce

in the first place and we cannot afford loosing a portion of it to the overhead in establishing multiple connections. In other words, the whole bandwidth is already consumed by one network connection and there is no room for added connections.

### 2.1.2 Discussion of the Results

To have a better representation of the results, the relative latencies for each environment and file-component size are outlined in Table 2-1. Generally speaking, when the number of concurrent connections is increased, CBD shows more effectiveness in decreasing the latency. The number of parallel connections for optimum performance depends on the available bandwidth at hand and the processing power and specifications of the machine being utilized. Therefore, there is no pre-defined optimum number of parallel connections. Also, when larger files are downloaded, CBD shows better performance in decreasing the latency. The reason for this may be due to the fact that when the duration of an FTP session is prolonged, the percentage of the network overhead (due to connection setup and tear down) as compared to the total network cost will be reduced. Another proven result based on the experiments determined that the limitation in a computer RAM can degrade the performance of the CBD. This is due to the fact that CBD needs more RAM for buffering its incoming data and maintaining its concurrent TCP connections.

| File Configuration | Environment | | | |
|---|---|---|---|---|
| | *100BaseT* | *10BaseT* | *Wireless 10BaseT* | *Dial up modem* |
| *3 Files (4.5 Meg)* | 0.48 | 0.47 | 0.46 | 1.06 |
| *3 Files (26 Meg)* | 0.45 | 0.39 | 0.74 | - |
| *3 Files (119.4 Meg)* | 0.42 | - | - | - |
| *5 Files (9 Meg)* | 0.34 | - | 0.32 | - |

**Table 2-1 Relative Latency (CBD/Conventional)**

## 2.2 Fault Tolerance

As with all distributed mechanisms, a higher degree of fault tolerance and load balancing can be obtained in CBD as compared to non-distributed approaches. Loosing connection to a server on the Internet due to the congestion or the server itself going down, is very likely to occur. In such a situation, the CBD client cannot access all the file components, and the downloading process will inevitably fail.

This problem can be addressed by resorting to a replication scheme, i.e., instead of having only one copy of a file component on one server, multiple file copies can be maintained on several servers. (The same approach used in mirror servers.) Depending on the nature of the file cluster, and the network's environment, various replication schemes can be implemented. Two practical simple schemes are illustrated in the following figures. Figure 2-12 illustrates the scenario in which a complete backup of the file cluster is put on an additional

server. Figure 2-13 illustrates the scenario in which a second copy of the file cluster is divided into several components and spread among the servers.

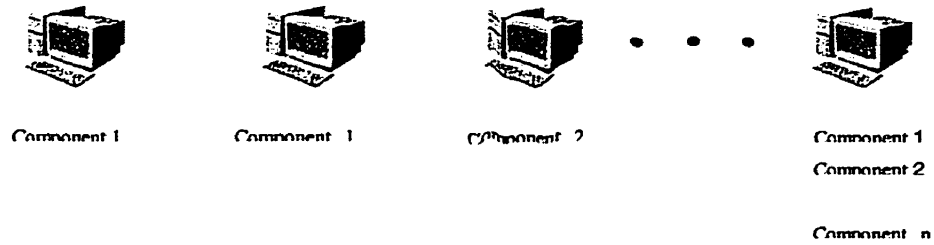


**Figure 2-12 Replication Scheme A**

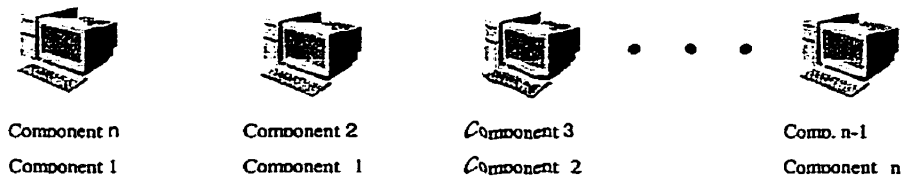


**Figure 2-13 Replication Scheme B**

## 2.2.1 Reliability Analysis

In this section the reliability that can be achieved while replicating the data over multiple servers will be explained and analyzed. Replication Scheme B is taken as an example and the number of replicated components needed to achieve a certain level of reliability will be analytically found. Before that the methodology on how to estimate the reliability will be described.

### 2.2.1.1 Reliability Concept

The reliability of a system is its ability to maintain operation over a period of time *t*. Formally, the reliability, *R(t)*, of a system is:

*R(t) = Pr (the system is operational in [0, t])* *(Equation 2-1)*

If *X* defined to be a random variable representing the lifetime of the system and also letting *F* be the cumulative distribution function (CDF) of *X*, then reliability of the system at time t is: (It is assumed that a system is working properly at $t = 0$; therefore, $R(0) = 1$)

$$R(t) = \Pr(x > t) = 1 - F(t) \qquad \textit{(Equation 2-2)}$$

When modeling a system, it is often assumed that the failure rate is constant. The importance of this assumption is when the failure rate, λ, is constant, the resulting CDF of the lifetime of the components is exponential. That is:

$$F(t) = 1 - e^{-\lambda t} \qquad \textit{(Equation 2-3)}$$

And the reliability:

$$R(t) = e^{-\lambda t} \qquad \textit{(Equation 2-4)}$$

Another measure often used for the analysis of systems is availability. The availability of a system is often expressed as the instantaneous availability, *A(t)*, and/or steady-state availability (i.e., $\lim_{t\to\infty} A(t)$ ). The instantaneous availability, *A(t)*, is defined as the probability that a system is operational at time *t*. It allows for one or more failures to have occurred during the interval *(0, t)*. If a system is not repairable, the definition of *A(t)* is equivalent to *R(t)*. Dependability

is used as the catch-all phrase for various measures such as reliability, availability, etc.

"Series-Parallel Block Diagram" will be used to as a modeling technique to analyze the reliability of the CBD. Next section describes this technique and it can be adapted into the CBD paradigm.

#### 2.2.1.2 Series-Parallel Reliability

The series-parallel reliability block diagram is a technique used for determining a system's dependability. In a block diagram model, components are represented as blocks and are combined with other blocks in *series, parallel, and/or k-out-of-n* configuration. A diagram that has components connected, as "series structure" requires that each component must be functioning for the overall system to be operational. A diagram that has components connected, as "parallel structure" requires only one component to be functional for the overall system to be operational. A "k-out-of-n structure" is superset of the series and parallel structures and requires *k* of the *n* total components to be functional for an operational system. Therefore, parallel and series structures are represented with "k-out-of-n structures" that are "1-out-of-n" and "n-out-of-n", respectively. The equations for the distribution function of these structures are: (The upper line represents a series and the lower line a parallel structure)

$$F(t) = \{^{1-\prod_{i=1}^{N}(1-F_i(t))}_{\prod_{i=1}^{N}F_i(t)} \qquad \textit{(Equation 2-5)}$$

### 2.2.1.3 Reliability of the Replication Scheme B

In Replication Scheme B, if there were no replicated data on the servers, "Series Reliability Block Diagram" would apply to the system. And if there were a complete set of file components on each participating servers, "Parallel Reliability Block Diagram" would apply.

Now, the question is how many copies of the data must be replicated and put on the distributed server system to achieve a certain level of reliability. Equation 2-6 gives the reliability of the CBD in which *P(t)* is the probability of having a connection to each of the servers. In other words, this equation calculates the distribution function for the *k-th* order statistic on *n* independent, identically distributed random variables.

$$P_{k|n}(t) = \sum_{i=0}^{k} \binom{n}{i} (1-P(t))^i P(t)^{n-i} \qquad \textit{(Equation 2-6)}$$

To make the computation of the above equation manageable, *P(t)* will be assumed to be a constant and identical value for all the participating servers. In a real-life scenario, to offset the effect of this unrealistic assumption in analyzing the reliability of CBD, a minimum value can be considered for *P(t)* (the worst-case scenario). Therefore, the Equation 2-6 can be rewritten as follows.

$$P_{k|n} = \sum_{i=0}^{k} \binom{n}{i} (1-P)^i P^{n-i} \qquad \textit{(Equation 2-7)}$$

To better illustrate how to use Equation 2-7 for calculating the number of replication needed to achieve a desired reliability, an example is provided here. The following assumptions are considered in this example.

- *5* servers are participating in the replication scheme
- The minimum probability of successfully downloading a file component from each server is *90%*.
- The desirable overall reliability for the replication scheme must be at least *99%*.

Finding the optimum number of data replication comes down to simply plugging in the given numbers into the Equation 2-7 and looking for the values of *k* that corresponds to the reliabilities more than *99%*

$$99\% < \sum_{i=0}^{k} \binom{5}{i} (1-0.9)^{i} (0.9)^{5-i} \qquad \textit{(Equation 2-8)}$$

By solving the above inequality, the optimum number of data replication (represented by the minimum value for *k*) to achieve reliabilities more than 99% can be found. The values of the right-hand side of above inequality for different k's are given in Table 2-2.

| K (number of replication) | R (Reliability of the system) |
|---|---|
| 0 | 0.59049 |
| 1 | 0.91854 |
| 2 | 0.99144 |
| 3 | 0.99954 |
| 4 | 0.99999 |

**Table 2-2 Reliability and Number of Replicated Data**

By referring to the above table, the optimum number for having an overall reliability more than 99% will be for *k=2*. (I.e., having two copies of the data, in addition to the original one). Therefore, there should be at least *3* copies of the data on the distributed server system to achieve an overall reliability of more than *99%*.

Although Series-Parallel Reliability Block Diagram is a common technique for modeling the network reliability, it fails to be a precise model for a replication scheme. To explain how and why it cannot be an accurate model, a scenario in which there are 5 servers with 3 file components on each of them (i.e.: k=2) is illustrated in Figure 2-14.

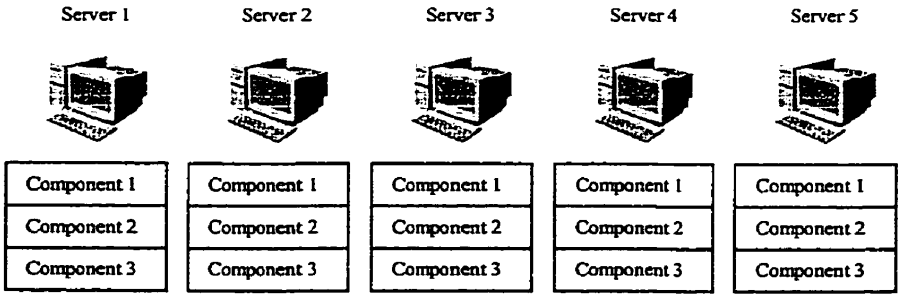


**Figure 2-14 Five Distributed Servers with Replication Degree of three**

According to the equation 2-9, the reliability of this system is:

$$\sum_{i=0}^{2} \binom{5}{i} (1-0.9)^{i} (0.9)^{5-i} \qquad \textit{(Equation 2-9)}$$

The above equation is the sum of three probabilities.

1. Probability of all connections to servers being up and running and
2. Probability of one connection being down
3. Probability of two connections being down

However, in the replication scheme shown in Figure 2-14 if three servers were down, as long as the missing servers were not adjacent, all the file components will be accessible. This fact has not been accounted for in the "Series-Parallel Reliability Block Diagram. Therefore, the actual reliability of the replication scheme is more than what calculated before. To present a precise probability, Equation 2-7 thus should be modified as follows.

$$P_{k|n} = \sum_{i=0}^{k+1} \binom{n}{i} (1-P)^i P^{n-i} - n(1-P)^{k+1} P^{n-k-1} \qquad \textit{(Equation 2-10)}$$

Figure 2-15 better illustrates the difference between Equations 2-7 and 2-10. In this figure, the reliabilities of a distributed server of five are plotted for different number of replicated data. Therefore, replication degree can vary between zero and four. The X-axis represents the number of participating servers. For the sake of discussion, a reliability of 90% is assumed for each individual server. Two graphs are shown in this figure, the dashed graph represents the Equation 2-7, and the solid graph the Equation 2-10. As anticipated, reliability of the system when using the Equation 2-10 is slightly higher.

Similar methodology can be applied to other replication schemes in order to verify the degree of replication required for certain dependability.

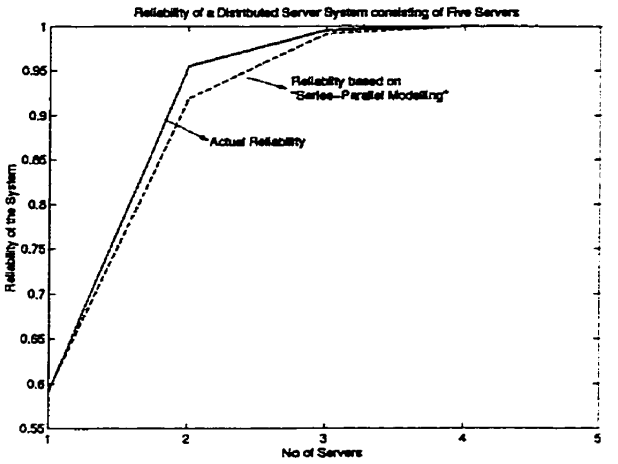


**Figure 2-15 Reliability of a System of Distributed Server Based on Number of Replicated Data**

## 2.2.2 Server Selection

When the file components are replicated, the issue becomes questionable as to how clients shall select the replicated components and what policies they shall follow to regulate their connections to the servers. Depending on the type of data (read-only or not) and network connections, different methods have been proposed in the literature. These different methods can be divided into two main classes: static and dynamic replication techniques [MCr95]. The following two sub-sections describe each of these mechanisms in more details. The replication schemes introduced above can be easily implemented using these methods.

### 2.2.2.1 Static Methods

In this type of server selection, clients must have *a priori* knowledge of the server location and network topology, i.e., they pre-determine which server has the quickest response to their request. Such a static server selection scheme is used

in distribution of network news utilizing NNTP. The file transfer application counts the number of hops between the client and each of the servers that contains the file component (the original and replicated copies) and chooses the one with the least number of hops, as the source of that file component. Since this metric is less likely to change over long period of time, it is used in several static server selection methods such as [Guy95].

#### 2.2.2.2 Dynamic Methods

In this type of server selection the file transfer application probes the servers that contain the replicated file component and chooses the first to reply (using the RTT) as the source of that file component. The extra cost at runtime incurred by dynamic methods, as compared to prior static knowledge of hop distances that can be justified based on the improved latency [ACh95].

# Chapter Three

# 3 File Segment Transfer Protocol (FSTP)

## 3.1 Motivation for Designing a New File Transfer Protocol

Slow performance of the TCP and its overly redundant mechanism to manage the network congestion and data flow, has motivated us to envisage a new protocol for transferring files over the Internet.

Earlier in this thesis, the FTP was utilized to test the CBD mechanism. The FTP works on top of the TCP that provides a reliable network connection. As a matter of fact, all the commercial network applications that must preserve the integrity of data use TCP as their transport level protocol. The most popular one, apart from the FTP, is Telnet.

The TCP is the most commonly used transport level protocol on the Internet. This was defined in the early eighties when the transmission medium was the bottleneck of communication. Today, the emerging use of high-speed networks, fiber optic links and powerful routers, has dramatically reduced the number of corrupted or lost packets over the Internet. This has caused the TCP's Sliding Window Mechanism to appear slow. Another drawback of the TCP is the fact that relatively large round trip propagation delay reduces its throughput. This causes a serious limitation in the TCP design by further causing a hindrance in achieving higher throughput. In addition, TCP also adds a 20-byte header for ensuring a reliable virtual connection. Some of the fields in this header appear to be

redundant and adds some unnecessary overhead. The processing and network costs of this lengthy header (especially when TCP packets are defragmented on their path) cause yet another hindrance on the TCP.

There have been numerous modifications to enhance the performance of TCP. Some of them are as follows.

- Selective Acknowledgment [MMR96]
- Window Scale option
- Round-Trip Time Measurement
- Protect Against Wrapped Sequence Numbers" [JBB92]

Latency of the TCP has been the primary motivation for designing faster application protocols. This has been achieved through a trade off between speed and loss of data integrity with no guarantee that the user will receive all the packets.

Similar techniques have been utilized in other application-level protocols. For example, Trivial File Transfer Protocol[4] (TFTP) [Sol92] takes advantage of the User Datagram Protocol (UDP) as its transport-level service to manipulate files with no reliability guarantee. Another example worth mentioning is the Asynchronous Reliable Delivery Protocol (ARDP) [ISI], which is a protocol

[4] Refer to appendix 7-2

developed by the University of Southern California for reliable transmission of data over UDP.

In the following two sections a new protocol, called File Segment Transfer Protocol (FSTP), will be introduced. This protocol is designed to transfer data much faster than the FTP, while maintaining the integrity of the data.

## 3.2 File Segment Transfer Protocol (FSTP)

FSTP is an application level protocol, which uses a UDP socket for transferring data and a TCP socket for sending control commands as compared to the FTP that uses two TCP connections for both the control commands and the data.

Basically, FTP protocol is not concerned about retrieving the missing or corrupted packets. The duty of providing a reliable network connection is delegated to TCP. The TCP ensures the integrity of the data by monitoring the incoming packets, and asking the sender for retransmission of the erroneous and/or missing packets. The FSTP operates significantly different from the FTP. Although we still have a TCP network connections for sending control commands between the client and the server, the data itself is transferred via UDP packets. We chose to utilize the UDP in this case as it affords simple access to the Internet Protocol (IP). Sending data over the UDP does not bind us to the restrained performance of the TCP.

Another benefit in utilizing the UDP packets is their ability to lessen processing time and network overhead in comparison to a TCP connection. However, since UDP only provides a datagram service, the necessary functionality for data consistency is provided in another level. This task is performed in the application level (i.e., by FSTP itself). To do this, on the server end, FSTP attaches a unique sequence number to each packet. On the client end, once FSTP receives all of the data packets in a stream, it looks for missing and/or corrupted packets. If FSTP client found any missing and/or corrupted packets, it sends a request for retransmission of the missing packets back to the FSTP server. The server then retrieves the missing parts of the file from its local disk and sends them once again to the client. This process continues until the data is completely transferred to the user.

Clearly, transmitting these packets at maximum speed would result in much of the packets getting lost in transit due to a smaller maintainable bandwidth over the Internet. To maintain a reasonably small percentage of packet loss, an inter-packet transmission delay is added. Adding this delay results in a more successful packet reception (with respect to the number of packets transmitted) and will result in much fewer required retransmissions.

Relatively speaking, the examination of received packets, the generation of a retransmission request, and the processing of a request by the server, takes a fair amount of time. Therefore, the perceived latency to the user will actually be

lowered due to the use of appropriate inter-packet transmission delay for a pre-chosen packet size. This inter-packet transmission delay will also work to minimize excessive network traffic and avoid network congestion. Also, to keep the fragmentation overhead as low as possible, we have to choose a UDP packet size that is less than the Maximum Transfer Unit (MTU) size for the network through which the data is traveling through.

Steps involved in FSTP process is summarized as follows:

- FSTP client opens up a TCP connection with the server for exchanging commands.
- The client receives a list of files and their respective sizes.
- Packet Size is set by the client and forwarded to the server. The server begins the transmission-timing test by submitting the UDP packets of the specified size to the client. Once the test is completed, the server transmits a message to the client indicating the total transmission time and the number of packets transmitted during the test.
- The client calculates an appropriate inter-packet delay time and transmits it to the server. The server then reads this value.
- The client forwards a "SEND" command by specifying the name of the file to be retrieved and taking note of the file's size from the previously acquired file listing. The size of the "sequence number" field in FSTP header is set dynamically. (This will be described in greater details later in this section). Both the client and the server calculate the number of bytes

necessary for the packets' sequence numbers. It is necessary for the client to calculate the number of bytes, so that it will properly handle the format of received packets without the server having to explicitly send a description of the format. This is a requirement as the client and server access the same program library to handle tag numbers. The server sends the file as a steam of UDP packets whose headers contain the file name and sequence number to indicate the position of the data in the file.

- After the stream of UDP packets has ended, the client generates a list of missing or corrupted packets and submits them in the form of a retransmission request to the server. To further locate the corrupt packets, the checksum capability of the UDP protocol is utilized.
- The server retransmits the requested packets in the same format as the original transmission.
- This retransmission process continues until all packets are received.

The current experimental system attempts to establish an average maintainable data rate in packets per second (for a chosen packet size) and transmit the data from a single server to the client.

## 3.3 FSTP flow control

TCP performs its flow control mechanism on the server end. The server adjusts its window size based on the client's and the network buffer sizes. In a simplified

scenario, the server transmits a new packet after receiving acknowledgement from the client; or retransmits the previous one, if timeout occurs. This allows for the data to be transmitted at a slower pace in passing through the network.

FSTP uses a totally different mechanism for managing the flow of data. The method presented in our first version of FSTP attempts to measure the throughput between the client and the server by conducting a brief test. This test takes place before the server attempts to flood its connection with UDP packets destined for the client over a short time interval (less than 1 second). The client then calculates the packets it received in total together, with the noted transmission time from the server, and calculates an appropriate delay. For example, if the server sends 1,000 packets in one second and the clients receives 100, we can speculate that if we transmit a packet every 10mS, we should be able to maintain a high packet reception rate of success. By using this simple scheme, a very reasonable data loss rate can be achieved.[5]

[5] For complete covering of the FSTP and its prototype implementation along with the results achieved from testing this new protocol, please refer to [SKr99]/

# Chapter Four

# 4 CBD-FSTP

## 4.1 Motivation for CBD-FSTP

According to the results discussed In chapter two, CBD can make downloading up to three times faster. Also, preliminary tests carried out on FSTP performance indicate a significant improvement to the latency [SKr99]. Intuitively, if we incorporate the CBD mechanism into the FSTP, we should be able to reach even a better performance compared to what has been achieved through the use of the two methods by themselves. Based on this reasoning, we designed a distributed version of the FSTP. We denoted this protocol as the CBD-FSTP. A prototype of this protocol was also implemented in Java. This prototype was used for our testing and comparison to the performance of CBD-FSTP along with two other file transfer protocols already covered earlier in this thesis (i.e.: FSTP & FTP).

This chapter starts off by introducing the CBD-FSTP and explaining its differences with the FSTP. Some code snippets from CBD-FSTP will be presented to clarify the design. Following that, the test environment (i.e.: methodology and specifications of the participating computers) will be described. Lastly, the data gathered through these measurements will be presented and discussed.

## 4.2 CBD-FSTP Description

As mentioned earlier, CBD-FSTP is a modified version of FSTP. Therefore, explaining the design details of this protocol seems to be redundant and will not be discussed in greater

details. Our focus here will only be on parts that have been added or modified to make possible the incorporation of CBD mechanism into FSTP.

A CBD-FSTP client should be able to establish multiple concurrent connections with several servers, instead of only one, before the requested file is downloaded. This is because the file to be downloaded by the CBD-FSTP has been split into components and put on different servers. Keeping this in mind, when the CBD-FSTP client established its TCP and UDP connections with the servers, packets are streamed through UDP connections down to the client. The client allocates a temporary file for each component (or each connection to the server). When a packet arrives, the client verifies the origin of the packet and sends it to the corresponding temporary location. For each of its connections, if the client does not receive more packets after a certain amount of time, it assumes that the server on the other side of UDP connection has finished submitting its packets. The client will then start looking for missing packets in each file component. It does not wait for other file components to get downloaded. (This feature is important for making this phase of download faster). The client finds the missing packets by using their *sequential number* field. (The same as the FSTP.) For each file component, the client sends a *retransmission* request, along with the sequence numbers of missing packets, back to its server (through its TCP connection). CBD-FSTP servers retransmit the requested packets to the client (through their UDP connections). This process continues until all the packets from different file components get downloaded. After all the packets belonging to a file component are downloaded, the client starts sorting them. Once again, it does not wait for other file components to be downloaded completely. The sorted file

components are stored into temporary locations. Upon finishing the sorting phase for all the file components, they get appended to reconstitute the original file. This task is done through the allocation of permanent memory space and writing the file components, based on the ascending order of its component numbers, into it.

The algorithm explained above is implemented in a client-server environment. Therefore, the tasks performed by each party will be described separately in the following two sections.

## 4.3 CBD-FSTP Server

The FSTP server runs on a known port and spawns a thread for every client wanting to set up a TCP connection. Upon establishing the TCP connection with a client, it sends a message back to the client announcing that it is ready to serve the client's request. The client's request can be either "send", or "retransmit".

If "send", the server

- Creates a buffer equal to the size of the file to be downloaded;
- Initializes an CBD-FSTP packet;
- Spirals into a loop and reads predefined chunks of data from the file into the FSTP packet, and sends it to the client. This loop iterates until the whole file is sent to the client.

If "retransmit", the server

- Reads in the missing sequence numbers sent by the client;
- Retrieves portions of the file corresponding to the sequence numbers;

- Puts them into FSTP packets and sends them to the client.

## 4.4 CBD-FSTP Client

The client spawns thread for each CBD-FSTP server. Each thread opens a TCP connection with its corresponding server. It then initiates an FSTP packet with the file component name, buffer size, packet size, and the server's IP address. Next, it receives the FSTP packets through its UDP connection with the server. It stores the packets (CBD-FSTP header plus the data itself) into a temporary file location. Once the last packet arrives, it checks for missing packets. If it finds any, it sends a "retransmit" request, along with the missing sequence numbers, to the server. It iterates until all the packets are received. It then sorts the data according to their sequential numbers and saves them on the local disk.

## 4.5 FSTP-CBD Prototype Implementation

To verify that our assumptions in designing the FSTP-CBD lead into a faster download, a prototype has been implemented in Java. The reason for choosing Java is the fact that it is a high-level language, which makes development, testing and debugging easier. The CBD-FSTP is implemented based on the codes developed for FSTP prototype implementation[6]. In this section, we briefly present the classes involved and discuss only parts of the code that ensures protocol compatibility with CBD. Figure 4-1 illustrates the sequence diagram for the CBD-FSTP.

---

[6] Refer to [SKr99] for details

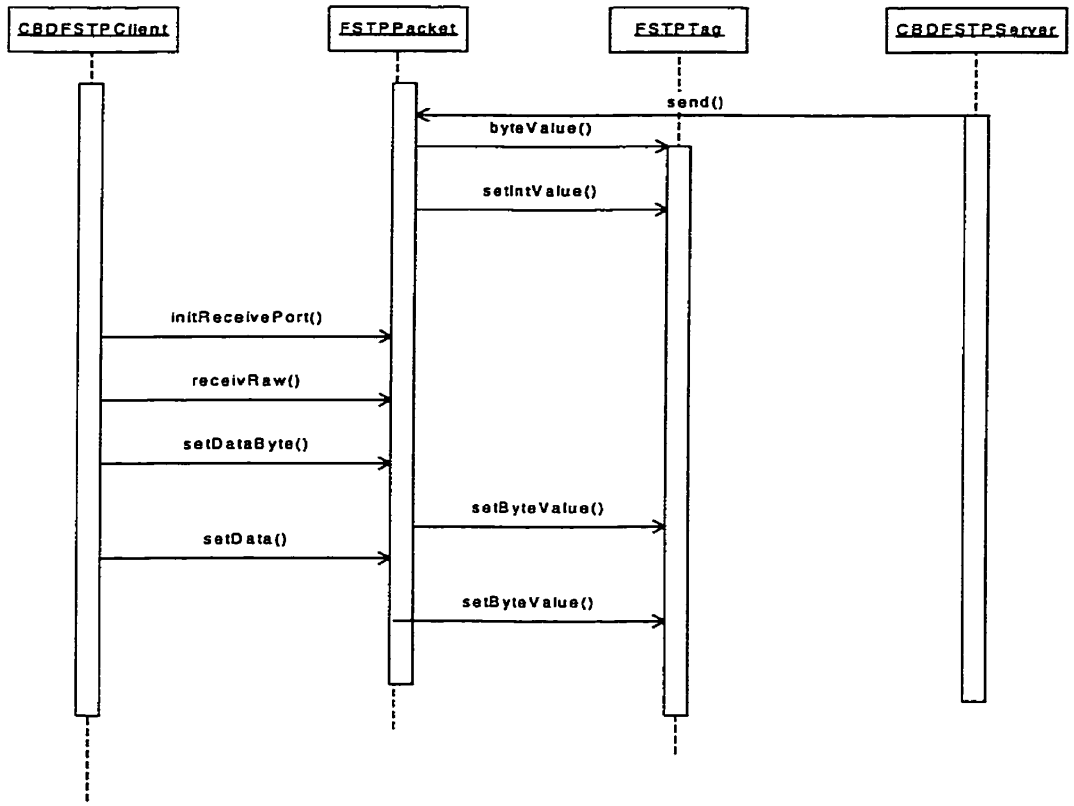


**Figure 4-1 UML Sequence Diagram of CBD-FSTP**

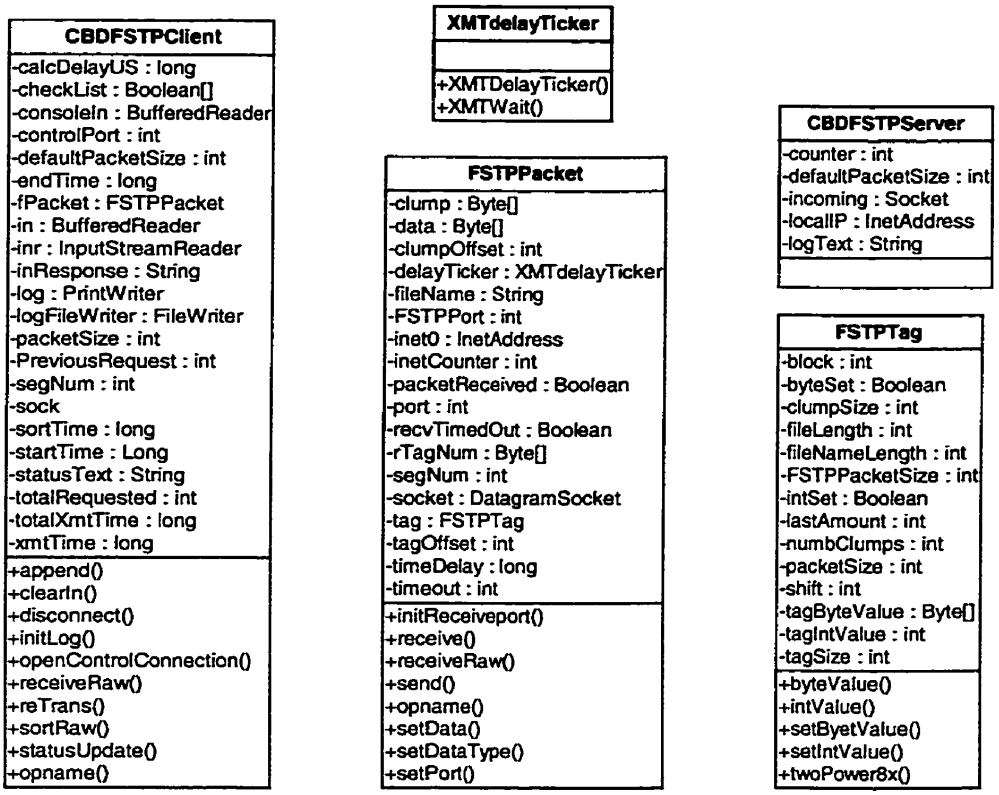


**Figure 4-2 Static Classes of CBD-FSTP**

Figure 4-2 shows the classes involved in the design of the CBD-FSTP. In following sub sections we explain the functionalities specific to the CBD-FSTP.

### 4.5.1 Multi-Threaded Client

As determined in the CBD approach, the client should establish parallel and/or concurrent connections to multiple servers. Therefore, the client should be implemented in a distributed fashion. In other words, the CBD-FSTP client should spawn a thread for handling each of its connections to multiple CBD-FSTP servers. All the threads should

complete their tasks before the next stage (appending the file components) is initiated. Figure 4-3 depicts the codes for completing this task.

```
public static void main(String[] args) throws  UnknownHostException, SocketException,
InterruptedIOException, FileNotFoundException, IOException
    {
        .
        .
        //allocate a thread for each server
        CBDFSTPClient t1 = new CBDFSTPClient(1);
        CBDFSTPClient t2 = new CBDFSTPClient(2);
        CBDFSTPClient t3 = new CBDFSTPClient(3);
        t1.start();
        t2.start();
        t3.start();
        try
        {
            //wait for all the threads to finish their tasks and then return to main
            t1.join();
            t2.join();
            t3.join();
        }
        .
        .
    }//end of main()
```

**Figure 4-3 Code Snippet for Spawning Threads for Parallel Connections**

## 4.5.2 CBD-FSTP Header

In comparison to the FSTP header, an extra field has been added to the CBD-FSTP header. This field represents the file component to which that packet belongs. This field is referred to as "*Component Number*". Figure 4-4 illustrates the CBD-FSTP packet header fields.


**Figure 4-4 CBD-FSTP Header**

*"Component Number"* field is required because the CBD-FSTP client opens a separate temporary file for saving each file component. Each temporary file should be addressable

simply by checking the header of the packet, so that the packet could be written to the proper file. The following code snippet (Figure 4-5) shows the code for creating the header. The code in bold is added to create the component number.

```
      public  FSTPPacket(String pFName, int pSize, int pPacketSize, InetAddress pInet,
int pSNum) throws SocketException
      {
            //copy parameters to instance variables
            inet0 = pInet;
            fileName = pFName;
            segNum = pSNum;
            //initialize data sorts of deals
            tag = new FSTPTag(pPacketSize, pFName.length(), pSize);
            data = new byte[tag.FSTPPacketSize];
            clump = new byte[tag.clumpSize];
            tagOffset = fileName.length() + 1;
            clumpOffset = tagOffset + tag.tagSize;
            // make a header for the packet
            //start with the File Name, '*' char, and Segment No

            String header = new String(fileName);
            Integer s = new Integer(segNum);
            String seg = s.toString();
            header+=seg;
            header +='*';
            System.arraycopy(header.getBytes(),0,data,0,header.length());
            rTagNum = new byte[tag.tagSize];
            //create socket for sending
            socket = new DatagramSocket();
            this.setPriority(MAX_PRIORITY);
            delayTicker  = new XMTDelayTicker();
      }
```

**Figure 4-5 Code Snippet for Creating CBD-FSTP Header**

### 4.5.3 Appending File Components

When all the file components are downloaded and sorted to re-build the original file, the client should append the file components at the end of each other. A permanent file with the same name as the original file is opened and the components are written sequentially. The code snippet illustrated below (Figure 4-6) performs this task.

```
static private void append(String fName, int segNum)
      {
            BufferedInputStream inStream = null;
            BufferedOutputStream outStream = null;
            boolean exceptionThrown = false;
            boolean append = true;
            File f = new File(fName + segNum);
            String file = "test.mov";
            int sNum = segNum;
            try
                  {
                        inStream = new BufferedInputStream(new FileInputStream(f));
                        outStream = new BufferedOutputStream(new
FileOutputStream(file, append));
                  }
            catch(Exception e)
            {
                  exceptionThrown = true;
            }
            try
            {
                  byte[] b = new byte[(int)f.length()];
                  inStream.read(b);
                  outStream.write(b);
                  outStream.close();
                  inStream.close();
                  System.out.println("file component "+sNum+" was appended to "+file);
            }
            catch(IOException e) {}
      }
```

**Figure 4-6 Code Snippet for Appending File Components**

### 4.5.4 Flow Control

Flow control in the CBD-FSTP is achieved using an Inter-Packet Transmission Delay (IPTD). The IPTD is the time interval between two consecutive packets transmitted by the server. If the server does not use an IPTD and continues sending out the packets without delay in between, the routers en route will flood and network congestion will occur. To make the matter worse, since the client will not be able to receive the packets dropped due to the congestion, it will ask the server retransmit them. In other words, more traffic, more congestion, more dropped packets, and consequently more latency occur. For this reason, the IPTD plays a very important role in the performance of the CBD-FSTP. Therefore, IPTD must be chosen very carefully. A large IPTD will result into a sluggish

protocol performance and a small IPTD will result in over exploitation and a waste of network resources, saturate the routers and slow performance.

The FSTP utilizes an adaptive mechanism for finding the optimized IPTD for each particular file transfer session. The operation of this mechanism is described as follows. The FSTP server sends out a number of packets to the client (e.g., 10,000) with no IPTD in between. The client calculates the number of missing packets and tells the server how many packets are missing. Based on this ratio (missing packets/sent packets), the server then calculates the optimum value for the IPTD that is used in that particular connection.

The utilization of this adaptive mechanism entails considerable improvement in the FSTP performance (refer to [SKr99]). However, we did not utilize this method in our CBD-FSTP prototype implementation. The reason being that network administration policies generally prevent the computers under its domain from flooding the network through a burst of traffic. When we tried to use the adaptive flow control mechanism (that is used in the FSTP) in our implementation, the computers used as our servers (which are administered by other authorities) did not allow our protocol to send out a burst of packets. As a final measure, we decided to find the best IPTD value for each server utilizing a brute force trial and error method. We then hard coded these values into our implementation.

### 4.5.5 Packet Size

Another factor that must be taken into consideration in implementing the CBD-FSTP prototype is the size of the packets sent by the server. Choosing the right size increasingly

improves the performance of the protocol. We measured the latency for downloading a file using various packet sizes, and found that a packet size of 1 kByte lead to the greatest performance. (This supports the results obtained in [SKr99].) This is more than likely due to the fact that the largest size is less than the Maximum Transferable Unit (MTU) for a typical network. During the implementation stages, we hard coded the packet size to1,024 bytes. Since our goal here is to prove the effect of adding the CBD to the FSTP, choosing a unique size for both FSTP and CBD-FSTP tests gives us a reliable basis for comparison.

## 4.6 Experimental Measurements

To verify the assumption that the CBD-FSTP is faster than the FSTP, a series of tests were carried out. Using our prototype during testing, we downloaded a file utilizing three different file transfer protocols: FTP, FSTP, and CBD-FSTP. During each test, the latency for downloading the file was measured. Our initial measurements have proven that CBD-FSTP makes downloading more than 50% faster as compared to the FSTP. In the following section we will describe our experimental measurements and present the end results.

### 4.6.1 Computers Used in Experimental Measurements

The computer utilized as our client is a Sun Sparc Ultra 1, with 768 Meg of RAM. Four workstations, running Sun or Linux, have used as our CBD-FSTP servers. They are located on four remote locations in Canada (in Victoria, BC, Edmonton, AB, Calgary, AB, and Regina, SK). All these machines are connected to the campus LANs (at the University

of Manitoba, University of Victoria, University of Alberta, University of Regina, & University of Calgary), which provide for high-speed connections to the Internet. Table (4-1) illustrates the specifications of the machines used in our tests. The operating systems running on these machines are all Unix or Linux. The computer located in Victoria, BC has the fastest processor.

| **Location** | UofM | UofV | UofA | UofR | UofC |
|---|---|---|---|---|---|
| **Platform** | SUNW, Ultra-1 | SUNW, Ultra-5_10 | Unknown | SUNW, Ultra-1 | SUNW, Ultra-2 |
| **Machine H/W** | Sun4u | Sun4u | i686 | Sun4u | Sun4u |
| **Node Name** | Cmc1 | Galois | 548pc1 | Ivanho | CSC |
| **Processor type** | Sparc | Sparc | Unknown | Sparc | Sparc |
| **Release** | 5.7 | 5.6 | 2.2.5-15 | 5.5.1 | 5.7 |
| **OS** | SunOS | SunOS | Linux | SunOS | SunOS |

**Table 4-1 Specifications of Machines Running the CBD-FSTP Server**

### 4.6.2 Measurements Procedure

We ran our tests by downloading a 20 Meg file using CBD-FSTP. The file to be downloaded was already split into four parts (5 Megs each) and put on the four servers. To download the 20 Meg file, four 5 Meg file components have downloaded simultaneously. This was done through a parallel of connections between our CBD-FSTP client and the servers and further appended to each other to create the original file. Next, we downloaded the same 20 Meg file using the FSTP. One of the above-mentioned computers is used as our FSTP server (the computer located in Victoria, B.C.). The file is downloaded from that server, and its latency is measured. And finally, to have a

benchmark for our measurements, the same file is downloaded from the same machine, this time using the FTP, and its latency is measured. We repeated the test 15 different times during the day, on alternate days of the week to compensate for different network traffic patterns.

We selected the machine in Victoria, BC as our server for the FSTP and the FTP. Although this machine is physically the furthest machine to our client, it gave us the fastest running time for downloading the file, as compared to the other three machines. This is due to the speed of the platform (Ultra-5_10), which is the quickest of them all. Also, This machine may also presumably be connected with higher bandwidth to its Internet gateway. By choosing the fastest connection for performing our FSTP and FTP measurements, we deliberately wanted to compare CBD-FSTP performance with the best performance obtained from the FSTP and FTP.

Previous experiences with the FSTP have proven that utilizing the packet size of 1,500 bytes will provide the fastest performance for this protocol [SKr99]. We later changed the packet size for the CBD-FSTP and measured its performance. As we predicted changing the packet size for the CBD-FSTP also related in much the way as the FSTP. Thus, the packet size of 1,500 bytes was chosen for our CBD-FSTP tests.

As mentioned earlier, the FSTP uses an adaptive mechanism to select the IPTD. In this mechanism, FSTP server sends a burst of packets (10,000 ) addressed to the client[7]. During the implementation stage of the CBD-FSTP we had chosen to set the IPTD manually, instead of using the above-mentioned algorithm. The reason for this decision is that a couple of the computers used as our servers did not allow us to send out such bursts of packets. This may be due to a security restriction imposed by system administration in order to control network traffic. The values measured in our tests do not necessarily present the quickest time. However, since the purpose of our tests is primarily to investigate the improvement of latency when using the CBD-FSTP relative to the FSTP; thus, not getting the fastest download time, will not change the comparative results.

### 4.6.3 Results

Figure 4-7 illustrates latency measurements when downloading a 20 Meg file using each of the three file transfer protocols (FTP, CBD and CBD-FSTP). According to these results, the CBD-FSTP can improve the latency up to 44%. The average value for the CBD-FSTP download is 59 seconds as compared to 85 for the FSTP. This is a significant improvement to the latency, knowing the fact that the FSTP has been already designed to achieve the fastest download possible.

These results encouraged us to repeat our tests to study the effect in size of the file and the level of distribution (or number of the participating servers) for the performance of the CBD-FSTP. To accomplish this, three sets of measurements were taken.

---

[7] Refer to [SKr99] for details

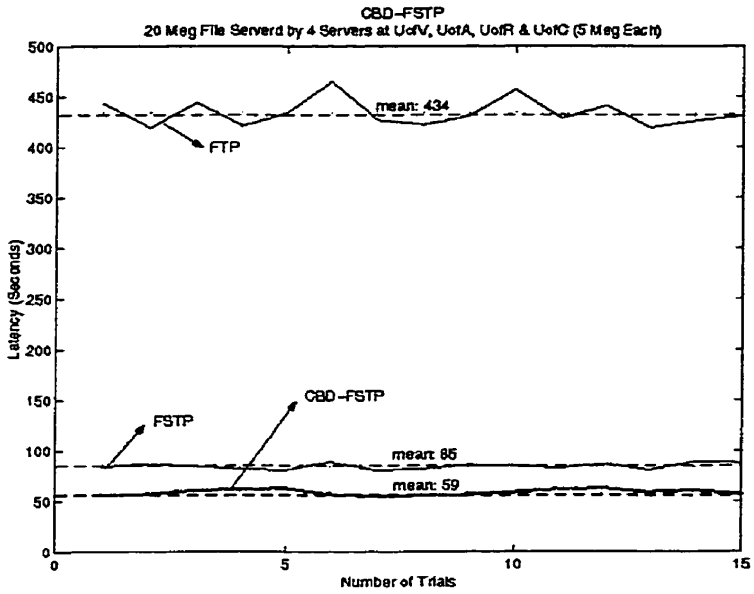


**Figure 4-7 Downloading a 20 Meg File Distributed on Four Remote Machines**

First, a 40 Meg file (instead of a 20 Meg) was downloaded. In the CBD-FSTP download, the 40 Meg file was split into four 10 Meg components. Figure 4-8 illustrates that the average value for the latency in the CBD-FSTP was 122 seconds. This means an improvement of approximately 62% in latency compared to the FSTP and 615% with respect to the FTP.

Next, we changed the distribution level from 4 to 3 and repeated the measurements, one for downloading a 15 Meg and the other for a 30 Meg file. Figure 4-9 illustrates that for a 15 Meg file, the average value for the latency in the CBD-FSTP is 52 seconds. This records a 52% improvement relative to the FSTP and 530% improvement with respect to the FTP. Figure 4-10 presents the results for downloading a 30 Meg file with an average

latency for the CBD-FSTP of 107 seconds. Comparatively, it is approximately 40% faster than the FSTP, and 545% than the FTP.

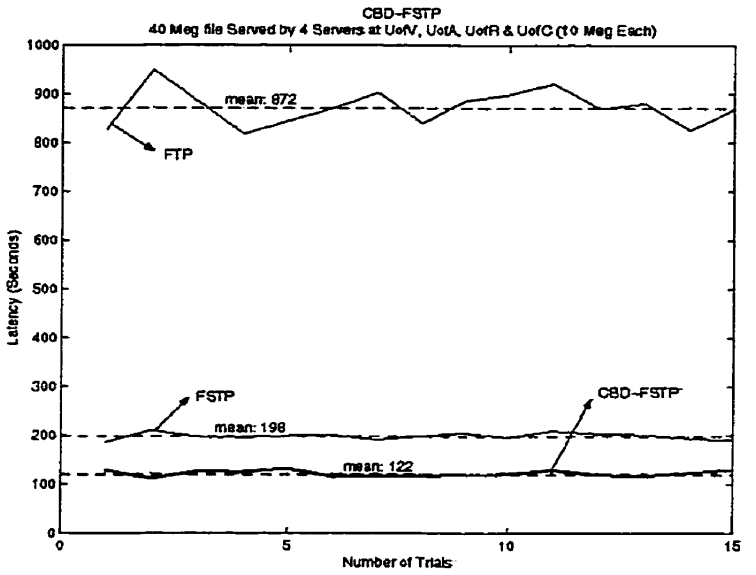


**Figure 4-8 Downloading a 40 Meg File Distributed on Four Remote Machines**

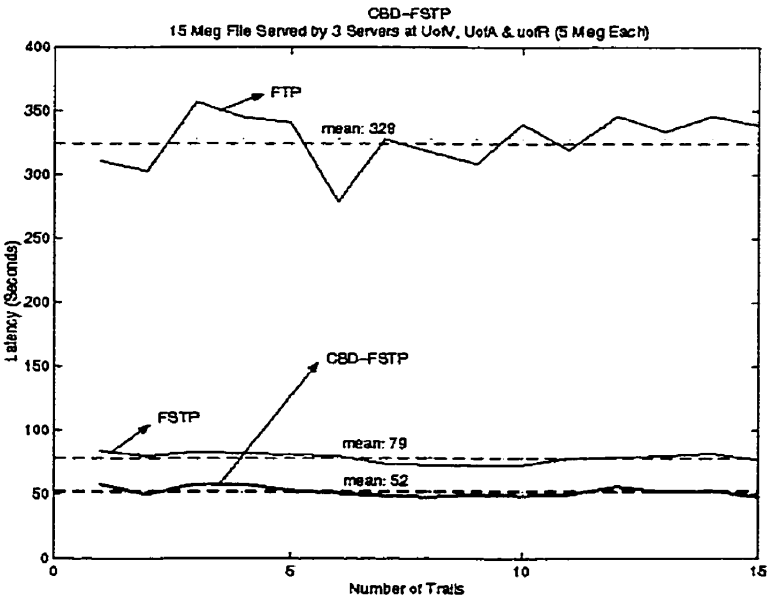


**Figure 4-9 Downloading a 15 Meg File Distributed on Three Remote Machines**

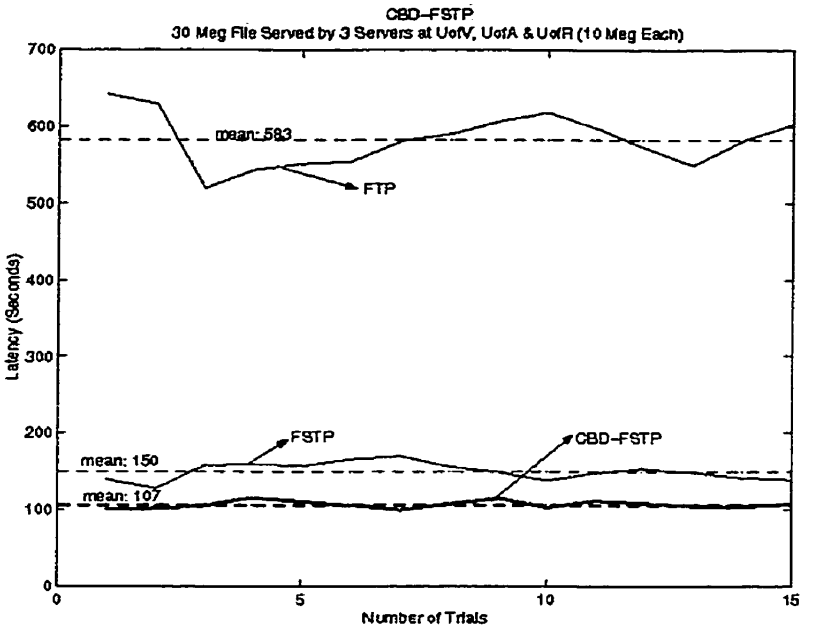


**Figure 4-10 Downloading a 30 Meg File Distributed on Three Remote Machines**

All the results obtained through our tests are summarized in Table 4-2.

| Size of File (Meg) | Number of CBD-FSTP Server | Average Latency (seconds) | | |
|---|---|---|---|---|
| | | CBD-FSTP | FSTP | FTP |
| 20 | 4 | 59 | 85 | 434 |
| 40 | 4 | 122 | 198 | 872 |
| 15 | 3 | 52 | 79 | 328 |
| 30 | 3 | 107 | 150 | 583 |

**Table 4-2 Average Latency (for Different Size of Files & Number of Servers)**

## 4.7 Analysis of the Results

To gain a better understanding of the performance of the CBD-FSTP, we calculated the relative latencies of the CBD-FSTP, FSTP and FTP with respect to each other and

identified all three latencies in Table 4-3. Results in Table 4-3 suggest that the best relative performance for the CBD-FSTP is obtained when a 40 Meg file is downloaded through four parallel connections. In other words, increasing the level of distribution makes the CBD-FSTP faster. On the same token, the bigger the file for download, the better the performance of the CBD-FSTP in comparison to the other two protocols.

These results have proven the soundness of our initial assumptions, based on the design of the protocol. We designed our protocol with the knowledge that using parallel connections provides higher throughput for an Internet user. On the other hand, this protocol was especially designed for large files, in which the extra overhead (due to setting up and tearing down the multiple connections) has less significance to the overall duration of the download.

| Size of File (Meg) | Number of CBD-FSTP Server | Relative Latency | | |
|---|---|---|---|---|
| | | CBD-STP/FTP | FSTP/FTP | CBD-FSTP/FSTP |
| 20 | 4 | 0.16 | 0.20 | 0.69 |
| 40 | 4 | 0.14 | 0.23 | 0.62 |
| 15 | 3 | 0.16 | 0.24 | 0.66 |
| 30 | 3 | 0.18 | 0.26 | 0.73 |
| **Average:** | | **0.16** | **0.23** | **0.67** |

**Table 4-3 Relative Latency (for Different Size of Files and Number of Servers)**

To investigate the connection speed between our client and servers, some data was collected. First, the round-trip times (RTT) between our client at the University of Manitoba and each of the CBD-FSTP servers were measured. To get an average result, the

measurements were repeated 20 times. Second, the number of hops between each server and our client was collected (see Table 4-4). The value for the RTT increases when the physical location of a server is further away. This explains the reason for the computer in Victoria, BC having the largest RTT, and the other in Regina, SK having the smallest RTT. (A complete list of routes between the client in University of Manitoba and each server is provided in Appendix 7.5). Looking at this routing list reveals that apart from the start and ending hops, the packets share the same route. Since all our servers and the client are located on different university domains across Canada and connected through the CANET2, all packets are coming to the client generally through the same path. This explains the reason for having a close RTT and almost the same number of hops. The only exception for this occurrence is the computer at the University of Calgary. Each packet originating from the University of Calgary should go through six local hops before entering the CANET2, while taking only two or three hops for the other three machines.

| Location of the Server | Average RTT (msec) | Number of Hops |
|---|---|---|
| Victoria, British Colombia | 83.44 | 8 |
| Edmonton, Alberta | 67.98 | 8 |
| Calgary, Alberta | 62.74 | 11 |
| Regina, Saskatchewan | 56.82 | 7 |

**Table 4-4 RTT and Number of Hops for Participating CBD-FSTP Server**

## 4.8 Comparison of CBD-FSTP with FSTP

The storing and sorting of the incoming packets is done concurrently in the CBD-FSTP (simultaneously for each file segment). This greatly reduces the delay corresponding to

this phase of the download, as compared to that of the FSTP. On the other hand, since the file components should be appended at the end of each other to re-create the original file, an extra delay exists in the CBD-FSTP due to this appending process.

CBD-FSTP is more process intensive as compared to the FSTP due to its multi-threaded design. It seams to work best on computers with higher processing power and larger RAM's. The CBD-FSTP also shows a superior performance as compared to the FSTP on machines with high-bandwidth connection to the Internet.

CBD-FSTP does not use an adaptive flow control method (as in the FSTP) to determine its Inter-Packet Transmission Delay (IPTD). If we are able to run our CBD-FSTP servers on networks that do not prevent us from sending out a burst of traffic (which is needed for utilizing the FSTP adaptive flow control), we can achieve even greater results.

# Chapter Five

# 5 Conclusion

In conclusion, our experiments and resulting data have proven the major effect of the CBD on the latency over the Internet. When the CBD mechanism was applied on the FTP, we achieved latencies as low as one third of the conventional FTP downloading time (i.e.: 300% improvement in the latency). The CBD is a simple yet powerful idea by which we can exploit the ever-increasing network bandwidth to achieve the highest possible speed.

In applying the CBD to the FSTP, it also brought us to another 30% reduction rate in the latency. Knowing the fact that the FSTP is itself an extremely fast file transfer protocol (on average, four to five times faster than FTP) shows the overall effectiveness of the CBD mechanism. Moreover, due to the constraints we had during our experiments, we could not utilize the adaptive flow control to optimize the performance of the CBD-FSTP. Another matter of consideration involves our comparison of the CBD-FSTP against the FSTP, whereby we measured the results of running the FSTP server on the fastest machine and connections available (four computers located in different universities) against running the CBD-FSTP on all four computers. In other words, we based our comparisons on a worst-case scenario. We believe that if we had machines with similar processing power and network connection capacity, the latency can be further reduced to 50%.

## 5.1 Future Work

The following outlines a variety of areas that deserve further research and improvements to the CBD and CBD-FSTP.

- The current implementation of CBD-FSTP is written in Java. Java, as a high level language, made the prototype implementation much easier. Using a faster language (e.g.: C/C++) allows this protocol to work faster and more efficiently.
- In the CBD-FSTP, only one-way data transfer from the server to the client is considered. During implementation, the client asked for a file and the server sent it to the client. This is in contrast to a more general two-way model that allows data to be sent in both directions (full duplex). In future implementations, the server and client modules should be integrated to support the two-way data transfer model.
- The CBD-FSTP currently has no provision for user authentication. This is an additional functionality that must be added in its next versions.
- The CBD-FSTP doesn't provide any error messages. Having error messages can assist system administrators and notify them of different types of errors that occure on the network.
- More tests should be done on the CBD and the CBD-FSTP (with different file-segment sizes and numbers of servers) in order to determine the best combination for optimum performance.
- To add reliability, a replication scheme should be integrated to the protocol.
- Component number can be deleted.
- A mechanism should be added to the CBD-FSTP to stop re-transmitting packets after a certain number of times in order to terminate the connection, assuming that

there is congestion en route. This prevents the protocol to hug the bandwidth and exploit the network resources unnecessarily.

- In the CBD-FSTP header, the "File Name" field can be removed. This will not create any ambiguity for the client to discern from which server the packet is coming from. It will however, reduces the overhead of the CBD-FSTP packet header.

# 6 References

[ACh95]
A. Chankhunthod, P. Danzig, Ch. Neerdales, M. Schwartz, K. Worrell, "A Hierarchical Internet Object Cache", Technical Report CU-CS-766-95, University of Colorado, Boulder, Mar 1995

[Bra95]
L. Brakmo and L. Peterson, "Performance Problems in 4.4BSD TCP", ACM Computer Communication Review, vol. 25, no. 5, pp. 69-86, Oct. 1995.

[Ber96]
T. Berners-Lee, R. Fielding, and H. Frystyk, "Hypertext Transfer Protocol-HTTP/1.0", IETF, RFC 1945, May 1996.

[BMP94] L. S. Bramko, S. W. O'Malley, & L. L. Peterson, "TCP Vegas: New Technique for Congestion Detection & Avoidance", Proceedings of SIGCOMM '94, ACM, PP. 24-35, 1999

[CaB96] B. Callaghn, Sun Microsystems, Inc. "WebNFS Server Specification", IETF, RFC 2055, Oct 1996.

[Cal96]
B. Callaghn, Sun Microsystems, Inc. "WebNFS Client Specification", IETF, RFC 2054, Oct 1996.

[Cla88]
D. D. Clark, MIT, "The Design Philosophy of the DARPA Internet Protocols", In proceedings of SIGCOMM'88, Computer Communication Review Vol. 18, No. 4

[Cro95]
M. Crovella and R. Carter, "Dynamic Server Selection in the Internet", Proceedings of the Third IEEE Workshop on the Architecture and the Implementation of High Performance Communication Subsystems (HPCS'95), August 1995.

[Cro98]
M. Crovella and P. Barford, "The Network Effects of Pre-fetching", In Proceedings of IEEE Infocom '98, San Francisco, CA, 1998.

[Fei98]
Z. Fei, S. Bhattacharjee, E. W. Zegura and M. H. Ammar, "A Novel Server Selection Technique for Improving the Response Time of a Replicated Service, Infocom'98.

[Fie97]
R. Fielding, J. Gettys, J. Mogul, H. Nielsen, and T. Berner-Lee, "Hypertext Transfer Protocol-HTTP1/1", IETF, RFC 2068, Jan. 1997

[Guy95]
J. Gutyon, M. Schwartz, "Locating Nearby Copies of Replicated Internet Servers", In Proceedings of SIGCOMM'95, August 1995

[Hoe96]
J. C. Hoe, "Improving the Start-up Behaviour of a Congestion Control Scheme for TCP", in Proceedings of the ACM SIGCOMM'96 Symposium, 1996

[ISI]
Information Science Institute, University of Southern California, "The Asynchronous Reliable Delivery Protocol", URL: <http://gost.isi.edu/info/ardp/>

[ISI81]
Information Sciences Institute, "Transmission Control Protocol", IETF, RFC 793, Sep 1981.

[JBB92]
V. Jacobson, R. Braden, D. Borman, "TCP Extensions for High Performance", IETF, RFC 1323, May 1992

[Kar87]
A. Karn, "Improving Round-Trip Time Estimates in Reliable Transport Protocols", in Proceedings of ACM SIGCOMM'87, 1987

[MMR96]
M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, "TCP Selective Acknowledgment Options", IETF, RFC 2018, Sun Microsystems, October 1996.

[Nag84]
J. Nagle, "congestion Control in TCP/IP internetworks, RFC 897, 1984

[Pad94]
V. N. Padmanabhan, and Jeffrey C. Mogul. "Improving HTTP Latency", Proceedings of the second International World Wide Web Conference, Chicago, IL, pages 995-1005, October, 1994.

[PoR85]
J. Postel, J. Reynolds, "File Transfer Protocol", IETF, RFC 959, ISI, October 1985

[Pos94]
J. Poster "User Datagram Protocol", IETF, RFC 768, USC/Information Sciences Institute, 28 August 1980.

[RGu98]
R. Guerraoui, A. Shiper, "Fault-Tolerance by Replication in Distributed Systems", in Proceedings of Reliable Software Technologies, Ada-Europe'96, Springer

[Skr99]
S. Kretschmann, "FSTP an Application-Level File Transfer Protocol as an Alternative to FTP", Bachelor of Science thesis, Electrical and Computer Engineering Department, University of Manitoba, 1999

[Sol92]
K. Sollins, "The TFTP Protocol (Revision 2)", IETF, RFC 1350, MIT, July 1992.

[Sun89]
Sun Microsystems, Inc., "NFS: Network File System Protocol Specification", IETF, RFC 1094, March 1989

# 7 Appendices

## 7.1 Transfer Control Protocol

### 7.1.1 Introduction

FTP and HTTP are dominant application-level protocols for transferring files over the Internet. These two both use TCP for creating their reliable network connections. TCP was developed in the late 1970's to transmit data reliably in the presence of Internet packet loss, primarily due to network congestion. This protocol later became the standard transport protocol for the Internet. TCP and other reliable transport protocol handle lost packets by having the sender detect the loss and then retransmit the lost packet. TCP also uses a congestion control algorithm to dynamically react to changing bandwidth limits of the Internet. It is formally defined in the RFC 793. Errors and inconsistencies are detected and debugged in the RFC 1122. Extensions are given in RFC 1323.

This section briefly describes how TCP works and discusses its mechanism for flow and congestion control. It also gives a brief overview of the modifications added to the original design of TCP to enhance its performance.

### 7.1.2 General Description

TCP is intended for use as a highly reliable host-to-host protocol between hosts on the Internet. TCP is a connection-oriented, end-to-end reliable protocol. TCP assumes it can

obtain a simple, potentially unreliable datagram service from the lower level protocols. In principle, TCP should be able to operate above a wide spectrum of communication systems ranging from hard-wired connections to packet-switched or circuit-switched networks.

TCP fits into a layered protocol architecture (Figure 7-1) just above a basic Internet Protocol, which provides a way for the TCP to send and receive variable-length segments of information enclosed in IP packets. The IP layer provides a means for addressing source and destination TCP's in different networks. IP also deals with any fragmentation or re-assembly of the TCP segments required to achieve transport and delivery through multiple networks and interconnecting gateways. IP also carries information on the precedence, security classification and compartmentation of the TCP segments, so that information can be communicated end-to-end across the Internet.

On the higher level, TCP interfaces to application processes and on the lower side to a lower level protocol such as IP. This interface consists of a set of calls much like the calls an operating system provides to an application process for manipulating files. For example, there are calls to open and close connections and to send and receive data on established connections. The interface between the TCP and lower level protocol is essentially unspecified except that it is assumed there is a mechanism whereby the two levels can asynchronously pass information to each other. The TCP is designed to work in a very general environment of interconnected networks.

As previously noted above, the primary purpose of the TCP is to provide reliable, securable logical circuit or connection service between pairs of processes. Providing this service on top of a less reliable Internet communication system, requires facilities in the following areas:

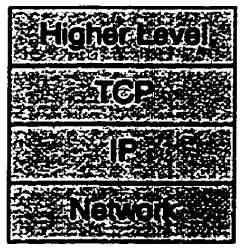


**Figure 7-1 Protocol Layering**

### 7.1.3 Basic Data Transfer

The TCP is able to transfer a continuous stream of octets in each direction between its users by packaging a number of octets into segments for transmission through the Internet

#### 7.1.3.1 Reliability

The TCP must recover data that is damaged, lost, duplicated, or delivered out of order by the Internet. This is achieved by assigning a sequence number to each octet transmitted, and requiring a positive acknowledgment (ACK) from the receiving TCP. If the ACK is not received within a timeout interval, the data is retransmitted. At the receiver, the sequence numbers are used to correctly order segments that may be received out of order and to

eliminate duplicates. Corrupted data is detected by adding a checksum to each segment transmitted, and checking it at the receiver. Corrupted segments are then discarded.

### 7.1.3.2 Flow Control

TCP provides a means for the receiver to govern the amount of data sent by the sender. This is achieved by returning a "window" with every ACK indicating a range of acceptable sequence numbers beyond the last segment successfully received. The window indicates an allowed number of octets that the sender may transmit before receiving further permission.

### 7.1.3.3 Multiplexing

To allow for many processes within a single Host to use TCP communication facilities simultaneously, the TCP provides a set of addresses or ports within each host. Concatenated with the network and host addresses from the IP, this forms a socket. A pair of sockets uniquely identifies each connection. That is, a socket may be simultaneously used in multiple connections. The binding of ports to processes is handled independently by each Host.

### 7.1.3.4 Connections

The reliability and flow control mechanisms described above require TCPs initialization and maintain certain status information for each data stream. The combination of this information, inciuding sockets, sequence numbers, and window sizes, is called a connection. Each connection is uniquely specified by a pair of sockets identifying its two sides. When two processes wish to communicate, their TCP's must first establish a

connection (initialize the status information on each side). When their communication is complete, the connection is terminated to free up the resources for other uses.
Since connections must be established between unreliable hosts and over the unreliable Internet, a handshake mechanism with clock-based sequence numbers is used to avoid erroneous initialization of connections.

### 7.1.4 Congestion control

TCP also provides an algorithm to prevent congestion in the routers, which is called "S*low Start*". When a connection is established, the sender initializes the congestion window to the size of maximum segment in use on the connection. It then sends one maximum segment. If this segment is acknowledged before the timer goes off, it adds one segment's worth of bytes to the congestion window to make it two maximum size segments and sends two segments. As each of these segments is acknowledged, the congestion window is increased by one maximum segment size. When the congestion window is *n* segments, if all *n* are acknowledged on time, the congestion window is increased by the byte count corresponding to *n* segments. In effect, each burst successfully acknowledged doubles the congestion window. When a timeout occurs, a threshold is set to half the current congestion window, and the congestion window is reset to one maximum segment. The congestion window grows with the exception that it stops growing exponentially when the threshold is hit. From that point on, successful transmissions grow the congestion window linearly. In effect, this algorithm is guessing that it is probably acceptable to cut the congestion window in half, and then gradually works its way up from there.

Continuous work is required in order to improve the congestion control mechanism. For example, TCP throughput can be improved by managing the clock more accurately, predicting congestion before timeouts occur, and using this early warning system to improve the slow start algorithm [BMP94]. Or, TCP Fast Retransmit algorithm developed by Jacobson [JBB92].

### 7.1.5 TCP Improvements

To enhance the performance of TCP protocol, various modifications have been added to the original protocol. For instance, TCP accommodates varying Internet delays by using an "adaptive retransmission algorithm". In essence, TCP monitors the performance of each connection and deduces reasonable values for timeouts. As the performance of connection changes, TCP revises its timeout value (i.e.: it adapts to the change). Karn's algorithm [Kar87] is now being used for this purpose. Another Problem with early implementations of TCP was "silly window syndrome" in which each acknowledgment advertises a small amount of buffer space available and each segment carries a small amount of data. This leads to inefficient use of available bandwidth. TCP now requires the sender and receiver to implement heuristics that avoid the "silly window syndrome". This is accomplished through the utilization of the Nagle algorithm [Nag84]. According to this algorithm, a receiver avoids advertising a small window, and a sender uses an adaptive scheme to delay transmission so it clumps data into large segments.

## 7.2 Trivial File Transfer Protocol (TFTP)

### 7.2.1 Introduction

Currently, TCP exhibits inefficiencies in terms of bandwidth consumption, retransmission latency, and server processing. TFTP attempts to reduce TCP's inefficiencies by shifting the reliability burden from the server to the client.

Although FTP is the most prevailing file transfer protocol in the TCP/IP suite, it is also the most complex and difficult to program. Many applications do not need the full functionality FTP offers, nor can they afford the complexity.

### 7.2.2 Trivial File Transfer Protocol

TFTP is a simple protocol to transfer files, and therefore earned the name Trivial File Transfer Protocol (TFTP) [Sol92]. It is built on top of UDP and is designed to be small and easy to implement. Therefore, it lacks most of the features of a regular FTP. It can only read and write files from and to a remote server. Any transfer begins with a request to read or write a file. If the server grants the request, the connection opens and the file is sent in fixed length blocks of 512 bytes. Each data packet contains one block of data, and must be acknowledged by an acknowledgment packet before the next packet is sent. A data packet of less than 512 bytes indicates termination of a transfer. If a packet gets lost in the network, the intended recipient will timeout and may retransmit his last packet (which may be data or an acknowledgment), thus causing the sender of the lost packet to retransmit that lost packet. The sender has to keep just one packet on hand for retransmission, since the lock step acknowledgment guarantees that all older packets

have been received. Notice that both machines, involved in a transfer, are considered senders and receivers. One sends data and receives acknowledgments while the other ends acknowledgments and receives data. Most errors cause termination of the connection. An error is signaled by sending an error packet. This packet is not acknowledged, and, therefore, not retransmitted. In other words, the TFTP server or user may terminate after sending an error message and in effect the other end of the connection may not get the message. Therefore, timeouts are used to detect such a termination when the error packet has been lost. Errors maybe caused by any one of three types of the following events:

- Not being able to satisfy the request (e.g., file not found, access violation, or no such user)
- Receiving a packet which cannot be explained by a delay or duplication in the network (e.g., an incorrectly formed packet)
- Losing access to a necessary resource (e.g., disk full or access denied during a transfer)

This protocol is very restrictive, in order to simplify the implementation. For example, the fixed length blocks allows for straightforward allocation, and the lock step acknowledgement provides flow control and eliminates the need to re-order incoming data packets.

The TFTP operates in a very simplistic way. The first packet sent asks for a file transfer – the packet specifies a file name and whether the file will be read or written. Blocks of files are numbered consecutively starting at number one. Each data packet contains a header that specifies the number of the block it carries, and each acknowledgement contains the

number of the block being acknowledged. A block of less than 512 bytes signals the end of the file. Error messages can be sent either in place of data or an acknowledgement; errors terminate the transfer.

TFTP retransmission is unusual because it is symmetric. Each side implements a timeout and retransmission. If the side sending data times out, it retransmits the last data block. If the side responsible for acknowledgments times out, it retransmits the last acknowledgement. Having both sides participate in retransmission helps to ensure that transfer will not fail after a single packet loss.

## 7.3 User Datagram Protocol (UDP)

### 7.3.1 Introduction

User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communications in an environment of interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) is used as the underlying protocol. This protocol provides a procedure for application programs to send messages to other programs with a minimum protocol mechanism. UDP is a thin protocol in a sense that it does not add significantly to the semantics of IP. It merely provides application programs with the ability to communicate using the unreliable connectionless packet delivery service. The protocol is transaction oriented, and delivery/duplicate protection are not guaranteed. UDP is an alternative to the TCP. Much like the TCP, UDP uses Internet Protocol to receive a datagram from one computer to another. Unlike TCP, however, UDP

does not provide the service of dividing a message into packets and reassembling it at the other end. Specifically, UDP doesn't provide for sequencing of the packets where data arrives in and it does not provide feedback to control the rate at which information flows between the machines. UDP provides protocol ports used to distinguish multiple programs executed on a single machine. In addition to the data sent, each UDP message contains both a destination port number and a source port number, making it possible for the UDP software at the destination to deliver the message to the correct recipient and for the recipient to send a reply.

An application program that uses UDP accepts full responsibility for handling the problem of reliability, including message loss, duplication, delay, out-of-order delivery, and loss of connectivity.

### 7.3.2 UDP Encapsulation and Protocol Layering

Figure 7-3 shows the protocol layers hierarchy and the position of UDP in it. UDP lies in the layer above the IP layer. Conceptually, application programs access UDP, which uses IP to send and receive datagrams. The IP layer is responsible for transferring data between a pair of hosts, while the UDP layer is responsible only for differentiating among multiple sources or destinations within one host.

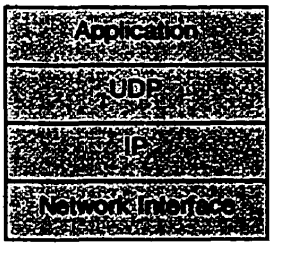


**Figure 7-2 Protocol layering**

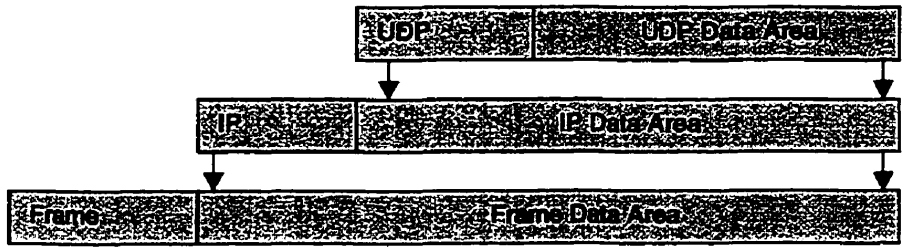


**Figure 7-3 UDP Encapsulation**

A UDP datagram is encapsulated in an IP datagram for transmission. This datagram is then encapsulated in a frame for traversing the network. This is illustrated in Figure 7-4.

### 7.3.3 UDP Multiplexing

UDP accepts datagrams from many application programs and forwards them on to the IP for transmission and accepts incoming UDP datagrams from the IP and forwards them on to the appropriate application program.

Conceptually, all multiplexing and demultiplexing between UDP and application programs occur through the port mechanism. In practice, each application program must negotiate with the operating system to obtain a protocol port and an associated port number before it can send a UDP datagram. Once the port has been assigned, any datagram the application program sends through the port will have that port number. While processing input, UDP accepts incoming datagrams from the IP software and demultiplexes based on the UDP destination port, as figure 7-5 shows.

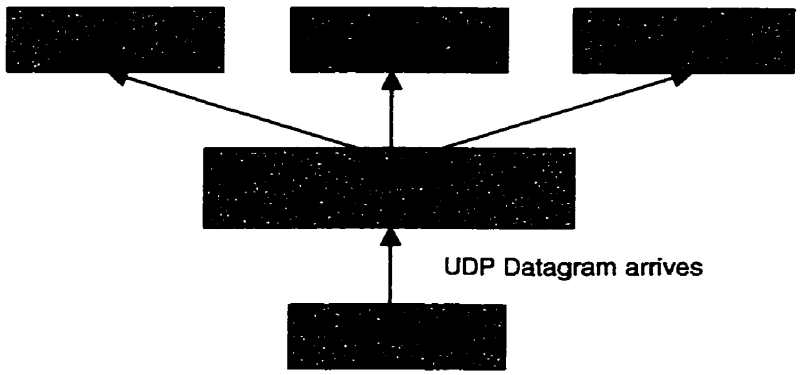


**Figure 7-4 UDP demultiplexing**

## 7.4 Source Code

In this section the source code are provided (in Java) for the CBD simulation and the CBD-FSTP prototype implementation.

### 7.4.1 CBD Implementation

The first two sections (FTPClient3 and FTPClientThread) show the source code used for simulating the concurrent download. The last section (CBDClientSequential) is the source code for simulating the conventional (sequential) FTP download.

#### 7.4.1.1 FTPClient3

```
/*
this class downloads three files from different servers in "concurrent" fashion. This is done by
spawning threads (FTPClientThread)
*/

import java.io.*;
import java.net.*;

public class FtpClient3
{
  //class variables
  static String logName = "thrLog";
```

```
public static void main(String arg[])
{
static RandomAccessFile log;
  String site1;
  String site2;
  String site3;

     if (arg.length == 3)
     {
    site1 = arg[0];
    site2 = arg[1];
    site3 = arg[2];
  }
     else
     {
    site1 = "ftp://ftp.dti.ad.jp/pub/unix/editor/xemacs/Attic/leim-skk.tar.gz";
    site2 = "ftp://ftp.netscape.com/pub/vjscdk/pr3/vjscdkb3.jar";
    site3 = "ftp://ftp.zweb.com/MiscUtilities/TFTPServer1-1-980730.exe";
  }
     // create the log file
  try
     {
    log = new RandomAccessFile(logName, "rw");
  }
  catch (FileNotFoundException e)
     {
    System.err.println(e);
  }
  // repeat the test for 10 consecutive times
  for (int i = 0; i<10; i++)
     {
            System.out.println("this is the " + i + "th iteration of loop");
            FtpClientThread t1 = new FtpClientThread(site1);
            FtpClientThread t2 = new FtpClientThread(site2);
            FtpClientThread t3 = new FtpClientThread(site3);
            t1.start();
            t2.start();
            t3.start();
        try
            {
                  t1.join();
            t2.join();
            t3.join();
      }
            catch (InterruptedException e)
            {
                  System.err.println(e);
      }
   }
  }
}
```

### 7.4.1.2 FTPClientThread

```
/*
this class downloads a file from a ftp server, and measures the time taken for the download.
written by babak s. noghani
```

```
import java.io.*;
import java.net.*;

*/
public class FtpClientThread extends Thread
 {

    String site;
    public FtpClientThread (String site)
    {
        super(site);
        this.site = site;
    }
    public void run()
    {
        try
        {
            URL ftpURL = new URL (site);
            downloadFile(ftpURL);
        }
        catch (MalformedURLException e)
        {
            System.err.println(site + "is not URL I understand!");
        }
    }
    private long downloadFile(URL ftpURL)
    {
        long start = System.currentTimeMillis();
        long finish;
        long duration = 0;
        //open a connection with the FTP server
        try
        {
            URLConnection uConn = ftpURL.openConnection();

            //extract the file name out of URL
            String fileName = ftpURL.getFile();
            fileName = fileName.substring(fileName.lastIndexOf('/') + 1);
            InputStream in = uConn.getInputStream();

            //get the connection specifications
            int cl = uConn.getContentLength();
            String ct = uConn.getContentType();

            //check to make sure it is a binary file
            if (cl == -1 || ct.startsWith( "text/"))
            {
                System.err.println("The size of this file: " + fileName + " is zero, or it's not a
binary file");
                System.exit(0);
            }
            //initialize the local variables for implementing read method
            int bufr = 128;
            byte[] b = new byte[cl+bufr]     ;
            int bytesread = 0;
            int offset = 0;

            //read the data into the temporary buffer "b"
            while (bytesread >= 0) {
            bytesread = in.read(b, offset, bufr);
            //System.out.println("remaining bytes: " + (cl - (offset+bytesread)));
            if (bytesread == -1)
            break;
            offset += bytesread;

            //for some reason read method blocks on some servers, when the remainig bytes are less
than byffer size. for our purpose, it doesn't matter, so we close the connection at this point.
            if ((cl - offset) < bufr)
            break;
```

```
            FileOutputStream fos = new FileOutputStream(fileName);
            fos.write(b);
            System.out.println("size of the file: " + fileName + " is: " + cl);
            //write the file stored in buffer "b", into the local disk
            finish = System.currentTimeMillis();
            duration = (finish - start)/1000;
            String message = "Time taken to download the file " + fileName + " is " + duration + "
seconds\n";
            System.out.println(message);
            FtpClient3.log.writeChars(message);
        }
        catch (Exception e)
        {
            System.err.println(e);
        }
        return duration;
    }//end of method downloadFile
}
```

### 7.4.1.3 FTPClientSequential

```
/*
this class downloads three files from different ftp servers in "sequential" fashion, and measures
the time taken for each download, and their total as well
written by babak s. noghani
*/
import java.io.*;
import java.net.*;

public class FtpClientSequential
 {
  //class variables
  static String logName = "seqLog";
  static RandomAccessFile log;

  public static void main(String arg[])
  {
    String site1 = "ftp://ftp.dti.ad.jp/pub/unix/editor/xemacs/Attic/leim-skk.tar.gz";
    String site2 = "ftp://ftp.netscape.com/pub/vjscdk/pr3/vjscdkb3.jar";
    String site3 = "ftp://ftp.zweb.com/MiscUtilities/TFTPServer1-1-980730.exe";
    long totDuration = 0;
    try
    {
      log = new RandomAccessFile(logName, "rw");
    }
    catch (FileNotFoundException e)
    {
      System.err.println(e);
    }
    String key;
    for (int j = 0; j < 10; j++)
    {
        // creates a URL object for each of 3 given ftp sites, and passes it on to the
"downloadFile" method
        for (int i = 1; i < 4; i++)
        {
            switch (i)
            {
```

```
                key = site1;
                break;
                case 2 :
                key = site2;
                break;
                case 3 :
                key = site3;
                break;
                default :
                key = site1;
            }
            try
            {
                URL FtpURL = new URL(key);
                totDuration += downloadFile(FtpURL);
            }
            catch (MalformedURLException e)
            {
                System.err.println(site1 + " this is not a URL I can understand!");
            }
        } //end of for loop
        String result = "total duration is :" + totDuration + " seconds\n";
        System.out.println(result);
        try
        {
            log.writeChars(result);
        }
        catch (IOException e)
            {
                 System.err.println("caught IOException: " + e.getMessage());
            }
    }//end of 2nd for loop
  }//end of main

  // method to download a file from an FTP server, save it on the local disk, and returns the time
taken for this process
  static long downloadFile(URL ftpURL)
  {
    long start = System.currentTimeMillis();
    long finish;
    long duration = 0;
    URLConnection uConn = null;
    FileOutputStream fos = null;
    InputStream in = null;

    //open a connection with FTP server
    try
    {
      uConn = ftpURL.openConnection();
    }
    catch (IOException e)
    {
      System.err.println("caught IOException: " + e.getMessage());
    }
    //extract the file name out of URL
    String fileName = ftpURL.getFile();
    fileName = fileName.substring(fileName.lastIndexOf('/') + 1);
    try
    {
        in = uConn.getInputStream();
    }
    catch (IOException e)
    {
      System.err.println("caught IOException: " + e.getMessage());
    }
    //check to see if the connection is established
    int cl = uConn.getContentLength();
    String ct = uConn.getContentType();
    if (cl == -1 || ct.startsWith( "text/"))
```

```
        System.err.println("The size of this file: " + fileName + " is zero, or it's not a binary
file");
        System.exit(0);
    }
    //initialize the local variables for implementing read method
    int bufr = 128;
    byte[] b = new byte[cl+bufr]     ;
    int bytesread = 0;
    int offset = 0;

    while (bytesread >= 0)
    {
        try
        {
            bytesread = in.read(b, offset, bufr);
        }
        catch (IOException e)
        {
            System.err.println("caught IOException: " + e.getMessage());
        }
        if (bytesread == -1)
            break;
        offset += bytesread;
        //for some reason read method blocks on some servers, when the remainig bytes are less than
byffer size. for our purpose, it doesn't matter, so we close the connection at this point.
        if ((cl - offset) < bufr)
            break;
    }//end of while loop
    //write the file, stored in buffer "b", into the local disk
    try
    {
        fos = new FileOutputStream(fileName);
    }
    catch (FileNotFoundException e)
    {
        System.err.println("caught FileNotFoundException: " + e.getMessage());
    }
    try
    {
      fos.write(b);
    }
    catch (IOException e)
    {
      System.err.println("caught IOException: " + e.getMessage());
    }
    System.out.println("size of the file: " + fileName + " is: " + cl);
    finish = System.currentTimeMillis();
    duration = (finish - start)/1000;
    String message = "Time taken to download the file " + fileName + " is " + duration + "
seconds\n";
    System.out.println(message);
    try
    {
      log.writeChars(message);
    }
    catch (IOException e)
    {
      System.err.println("caught IOException: " + e.getMessage());
    }
    return duration;
  }
}
```

## 7.4.2 CBD-FSTP Prototype Implementation

This section provides the source code for the CBD-FSTP. The code has been taken from [SKr99]. Apart from slight modifications, we have not changed the original design and naming. These changes were necessary to adopt to the distributed nature of the CBD-FSTP. Similar to its original design, it consists of five classes. The first two act as the client and server. The other three encapsulates the process of creating a CBD-FSTP packet.

### 7.4.2.1 CBDFSTPClient

```
/**
* This is a CBD-FSTP Client class. It opens up a TCP connection to an already know CBD-FSTP server.
It asks the server to "SEND" it a file. The packets containing the file are received through a UDP
connection. It checks for missing/corrupted and sends a "RETRANAMIT" to the server, along with the
sequence # of the missing packets. It then receives the missing packet. This loop iterates untill
all the packets are transferred to the client. it then closes its connection with the CBD-FSTP
server.
The original code is written by Steve Kretshmann, and modified by babak s. Noghani. Modifications
include:
1. Changing the single-threaded client to multi-threaded
2. Adding an additional field to the packet header
3. Adding the functionality of appending the file components
4. Disabling the adaptive flow control mechanism
5. Hard-coding the packet size

Last Modified: 22/11/99
**/

import java.net.*;
import java.io.*;
import java.util.*;

public class CBDFSTPClient extends Thread
{
    static private final int defaultPacketSize = 1024;
    static private final int controlPort = 4712; //TCP port number
    private int packetSize = defaultPacketSize;
    private FSTPPacket fPacket;
    private Socket sock = null;
    private InputStreamReader inr = null;
    private BufferedReader in = null;
    private PrintWriter outRequest = null;
    private String inResponse = null;
    private String statusText;
    private boolean[] checkList;
    private int missingCounter;
    private PrintWriter log;
```

```
    private long calcDelayUS = 0;
    static private BufferedReader ConsoleIn = null;
    private FileWriter logFileWriter = null;
    private boolean logInitialized = false;
    private long startTime, endTime, sortTime, totalXmtTime, xmtTime = 0;
    private int totalRequested, previousRequested = 0;
    int segNum = 0;

    private CBDFSTPClient(int segNo) throws java.io.IOException, java.io.FileNotFoundException,
java.io.InterruptedIOException, java.net.SocketException, java.net.UnknownHostException
    {
        segNum = segNo;
    }

    public static void main(String[] args) throws  UnknownHostException, SocketException,
InterruptedIOException, FileNotFoundException, IOException
    {
        String fileN = "test.mov";
        long beginTime=System.currentTimeMillis();
        // initialize Log
        ConsoleIn = new BufferedReader(new InputStreamReader(System.in));
        InetAddress localip=InetAddress.getLocalHost();
        System.out.println("localip = " + localip);
        //allocate a thread for each server
        CBDFSTPClient t1 = new CBDFSTPClient(1);
        CBDFSTPClient t2 = new CBDFSTPClient(2);
        CBDFSTPClient t3 = new CBDFSTPClient(3);
        t1.start();
        t2.start();
        t3.start();
        try
        {
            //wait for all the threads to finish their tasks and then return to main
            t1.join();
            t2.join();
            t3.join();
        }
        catch (InterruptedException e)
        {
              System.err.println("Caught InterruptedException: " + e.getMessage());
        }
        //append the three file segments at the end of each other
        for (int segmt =1; segmt<4; segmt++)
        {
            append(fileN, segmt);
        }
        long appendTime=System.currentTimeMillis();
        System.out.println("Overall Time for CBD-FSTP File Transfer = "+((appendTime-
beginTime)/1000)+" S");
    }//end of main()

    public void run()
    {
        try
        {
            String host = null;
            int seg = this.segNum;

            switch (seg)
            {
                case 1: host = "galois.csc.uvic.ca"; break;
                case 2: host = "548pc15.ee.ualberta.ca"; break;
                case 3: host = "ivanhoe.engg.uregina.ca"; break;
                case 4: host = "csc.cpsc.ucalgary.ca"; break;
            }
            packetSize=1024;
            logInitialized = false;
            initLog("log"+seg);
            openControlConnection(host, seg);
            //receive file
```

```
            String fileName = new String("test.mov" + seg);
            startTime = System.currentTimeMillis();
            //initialize the packets based on the sizes of file components
            log.println("File = test.mov" + seg );
            if ((seg ==1) || (seg ==3))
            {
                fPacket = new FSTPPacket(fileName, 10021564, packetSize,
InetAddress.getByName(host), seg);
            }
            else
            {
                fPacket = new FSTPPacket(fileName, 10021192, packetSize,
InetAddress.getByName(host), seg);
            }
            fPacket.initReceivePort(seg);
            receiveRaw(fileName, seg);
            endTime=System.currentTimeMillis();
            Disconnect();
            log.close();
        }
        catch (Exception e)
        {
            System.out.println(e);
        }
    }

    private void clearIn()
    {
        while ((!statusText.startsWith("000")) || (statusText==null))
        {
            try
            {
                statusText = in.readLine();
            }
            catch(IOException e){}
            statusUpdate();
        }
    }

    private void Disconnect()
    {
        //issue command and display results
        outRequest.println("QUIT");
        outRequest.flush();
        try
        {
            statusText = in.readLine();
        }
        catch(IOException e){}
        statusUpdate();
    }

    private void initLog(String logFileName)
    {
        try
        {
            logFileWriter = new FileWriter(logFileName);
            log = new PrintWriter(logFileWriter);
            logInitialized = true;
            //statusText = "Local Log file created: ";
            //statusText += logFileName;
            //statusUpdate();
            //System.out.print("Enter Log Header Information:");
            //String Header = " ";
            /*
            while (Header.length() > 0)
            {
                Header=ConsoleIn.readLine();

                log.println(Header);
```

```
            */
            Date now = new Date();
            log.println("Date/Time of Test:"+now);
            log.println("*****************************************************");
        }
        catch (Exception e)
        {
            logInitialized = false;
            statusText = "Could Not Open Log File: ";
            statusText += logFileName;
            statusText += e;
            statusUpdate();
        }
    }

    private void openControlConnection(String host, int segment)
    {
        //initialize the TCP socket to FSTPServer
        boolean exceptionThrown = false;
        try
        {
            System.out.println("file component "+segment+" is serverd by: " + host);
            sock = new Socket(host,controlPort);

            if (logInitialized) log.println("Connected to :"+host+", control port:"+controlPort);
        }
        catch (Exception e)
        {
            exceptionThrown = true;
            //print to status window
            statusText = "570 Error. Could not Open connection to ";
            statusText +=host;
            statusText +=" ";
            statusText += e;
            statusUpdate();
        }
        if(!exceptionThrown) try
        {
            inr = new InputStreamReader(sock.getInputStream());
            in = new BufferedReader(inr);
            outRequest = new PrintWriter(sock.getOutputStream());
        }
        catch(Exception e)
        {
            System.out.println("Error Creating Control In/Out Readers:"+e);
            exceptionThrown = true;
        }

        if (!exceptionThrown)
        {
            try
            {
                statusText = in.readLine();
            }
            catch(IOException e){}
            clearIn();
        }
    }

    private void receiveRaw(String rFileName, int segNo) throws SocketException,
FileNotFoundException, IOException
    {
        boolean done = false;
        boolean exceptionThrown = false;
        boolean reTR = false;
        String rawFileName = new String(rFileName + ".raw");
        //System.out.println("rawFileName: "+rawFileName);
        File f=new File(rawFileName);
        FileOutputStream outStream = null;
```

```
            //initialize packet checklist
              checkList = new boolean[fPacket.tag.numbClumps];
            FileInputStream rawStream = null;
            while (!done)
            {
                //open outputfile
                try
                {
                    outStream = new FileOutputStream(rawFileName, reTR);
                    statusText = "(file component "+segNo+") Local Output file Opened :";
                    statusText += rawFileName;
                    statusUpdate();
                }
                catch(Exception e)
                {
                    exceptionThrown = true;
                    statusText ="(file component "+segNo+") Could Not Create RAW output file:";
                    statusText += rawFileName;
                    statusText += " ";
                    statusText += e;
                    statusUpdate();
                } // end of try and exceptions for

                //receive Packets
                if (reTR)
                {
                    System.out.println("(file component: " + segNo + ") Generating Retrans Request");
                    reTrans(segNo);
                }
                else
                {
                    reTR = true;
                    String reqOut = new String("SEND ");
                    reqOut += rFileName;
                    System.out.println("(file component: " + segNo + ") Command sent to the server =
"+reqOut);
                    outRequest.println(reqOut);
                    outRequest.flush();
                    if (logInitialized)
                    {
                        log.print("Init Req:"+fPacket.tag.numbClumps+" ");
                        previousRequested = fPacket.tag.numbClumps;
                    }
                    statusText=in.readLine();
                    statusUpdate();

                }
                fPacket.recvTimedOut = false;
                while (!fPacket.recvTimedOut)
                {
                    try
                    {
                        fPacket.receiveRaw(segNo);
                        if (fPacket.packetReceived)
                        {
                            outStream.write(fPacket.data);
                        } // 4 end if packetReceived
                    }

                    catch(Exception e){}
                } //end while !fPacket,recvTimedOut
                outStream.close();
                statusText=in.readLine();
                statusUpdate();
                if (logInitialized)
                { //extract XMT time from server
                    int index0 = statusText.indexOf("#");
                    int index1 = statusText.indexOf(" ",(index0+1));
                    xmtTime = Long.decode(statusText.substring((index0 + 1), (index1))).longValue();
```

```
                totalXmtTime += xmtTime;
            }
            clearIn();
            log.print("XMT Time = "+xmtTime+" ");
            //check to see if all packets have been received
            try
            {
                rawStream = new FileInputStream(f);
                statusText = "(file component: " + segNo + ") Local RAW file Opened :";
                statusText += rawFileName;
                statusUpdate();
            }

            catch(Exception e)
            {
                exceptionThrown = true;
                statusText = "(file component: " + segNo + ") Could Not Open RAW file:";
                statusText += rawFileName;
                statusText += " ";
                statusText += e;
                statusUpdate();
            }

            long AmtRead = 0;
            while (AmtRead < f.length())
            {
                int amt = rawStream.read(fPacket.data);
                fPacket.setDataByte();
                checkList[fPacket.tag.intValue()] = true;
                AmtRead += amt;
            }

            System.out.println("(file component: " + segNo + ") Checking to see if we got all the
packets");
            done = true;
            //check to see if all packets have been received
            for(int i=0;i<fPacket.tag.numbClumps;i++)
            {
                if (!checkList[i])
                {
                    done = false;
                    i = fPacket.tag.numbClumps-1;
                }
            }
        } //end  !done while
        if (logInitialized)
        {
            endTime=System.currentTimeMillis();
            log.println("Pkts Rec'd:"+previousRequested+" =100%
T/Pkt="+((float)xmtTime/(float)previousRequested));
            log.println("*****************************************************");
            log.println("Total Server Transmission Time = "+(totalXmtTime/1000)+" S");
            log.println("Total Receive Time = "+((endTime-startTime)/1000)+" S");
            log.println("Total ProcessingTime = "+((endTime-startTime-totalXmtTime)/1000)+" S");
        }
        sortRaw(rFileName, segNo);
        if (logInitialized)
        {
            sortTime=System.currentTimeMillis();
            log.println("Sort Time = "+((sortTime-endTime)/1000)+" S");
            log.println("*****************************************************");
            log.println();
            log.println("Overall Time for FSTP File Transfer = "+((sortTime-startTime)/1000)+" S");
        }
    }

    private void reTrans(int segNo)
    {
      int seg = segNo;
        String rTransHead = new String(fPacket.data, 0, fPacket.tagOffset);
```

```
        rTrans += rTransHead;
        //add packets and count how many are missing
        missingCounter = 0;
        String rTrans = new String("RETR ");
        int last = 0;
        for(int i=0;i<fPacket.tag.numbClumps;i++)
        {
            if (!checkList[i])
            {
                missingCounter++;
                rTrans = rTrans + i + ",";
                last = i;
            } //end if not checked off
        }//end for
        //send retrans request
        System.out.println("(file component: " + seg + ") Sending Retrans request for
"+missingCounter+" Packets.");
        if (logInitialized)
        {
            int pktsRcvd = previousRequested-missingCounter;
            log.print("Pkts Rec'd:"+(pktsRcvd)+" ");
            log.println("="+(int)(100*((float)(pktsRcvd))/previousRequested)+"%
T/Pkt="+((float)xmtTime/(float)previousRequested));
            previousRequested = missingCounter;
            log.print("Retr Req:"+missingCounter+" ");
        }
        System.out.println( "(file component: " + seg +") rTrans : "+rTrans);
        outRequest.println(rTrans);
        outRequest.flush();
        try
        {
            statusText = in.readLine();
            statusUpdate();
        }
        catch(IOException e)
        {
            System.out.println("(file component: " + seg + ") Error Reading Retr Msg. from Server
"+e);
        }

    }
    static private void append(String fName, int segNum)
    {
        BufferedInputStream inStream = null;
        BufferedOutputStream outStream = null;
        boolean exceptionThrown = false;
        boolean append = true;
        File f = new File(fName + segNum);
        String file = "test.mov";
        int sNum = segNum;
        try
            {
                inStream = new BufferedInputStream(new FileInputStream(f));
                outStream = new BufferedOutputStream(new FileOutputStream(file, append));
            }
        catch(Exception e)
        {
            exceptionThrown = true;
        }
        try
        {
            byte[] b = new byte[(int)f.length()];
            inStream.read(b);
            //outStream.write(b, ((segNum - 1)*(int)f.length()), (int)f.length());
            outStream.write(b);
            outStream.close();
            inStream.close();
            System.out.println("file component "+sNum+" was appended to "+file);
        }
        catch(IOException e) {}
```

```
private void sortRaw(String rFileName, int segNum)
{
    boolean exceptionThrown = false;
    String rawFileName = new String(rFileName + ".raw");
    checkList = new boolean[fPacket.tag.numbClumps];
    FileInputStream inStream = null;
    RandomAccessFile rOutFile = null;
    int sNum = segNum;
    try
    {
        inStream = new FileInputStream(rawFileName);
        inStream.skip(0);
        rOutFile = new RandomAccessFile(rFileName, "rw");
        statusText = "(file component "+sNum+") Local Raw file Opened for Sorting :";
        statusText += rawFileName;
        statusUpdate();
    }

    catch(Exception e)
    {
        exceptionThrown = true;
        //print to status window
        statusText = "(file component "+sNum+") Error Opening Raw file for Sorting : ";
        statusText += rFileName;
        statusText += " ";
        statusText += e;
        statusUpdate();
    }

    boolean inAvailable = false;
    try
    {
        inAvailable = (inStream.available() > 1);
    }
    catch(IOException e)
    {
        exceptionThrown = true;
    }

    while (inAvailable && !exceptionThrown)
    {
        try
        {
            inStream.read(fPacket.data);
            fPacket.setData();
        }
        catch(IOException e)
        {
            exceptionThrown = true;
        }
        int pNumber = fPacket.tag.intValue();
        checkList[pNumber] = true;
        if (fPacket.tag.last)
        {
            //seek to the position of the last packet
            try
            {
                rOutFile.seek((fPacket.tag.numbClumps - 1)* fPacket.tag.clumpSize);
                rOutFile.write(fPacket.clump,0,fPacket.tag.lastAmount);
            }
            catch(IOException e)
            {
                exceptionThrown = true;
            }
        }
        else
        {
            try
```

```
                    rOutFile.seek(pNumber * fPacket.tag.clumpSize);
                    rOutFile.write(fPacket.clump);
                }
                catch(IOException e)
                {
                    exceptionThrown = true;
                }
            }
            try
            {
                inAvailable = (inStream.available() > 0);
            }
            catch(IOException e)
            {
                exceptionThrown = true;
            }
        }//end while

        if (!exceptionThrown)
        {
            try
            {
                rOutFile.close();
                inStream.close();
            }
            catch(IOException e){}
        }
        boolean all = true;
        for(int j=0;j>fPacket.tag.numbClumps;j++)
        {
            if (!checkList[j]) all = false;
            System.out.println("(file component "+sNum+") Didn't get: "+j);
        }
        if (all) System.out.println("(file component "+sNum+") Got all packets");
        boolean rawDeleted = false;
        try
        {
            File rawInFile = new File(rawFileName);
            rawDeleted = rawInFile.delete();
            if (rawDeleted) System.out.println("(file component "+sNum+") Raw File Deleted");
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
        if (!rawDeleted) System.out.println("(file component "+sNum+") Raw File Could NOT be
Deleted");
    }

    private void statusUpdate()
    {
        System.out.println(statusText);
    }
}
```

### 7.4.2.2 CBDFSTPServer

```
/**
* This is a CBDFSTP Server class (#1). It listens on TCP port #1024. Upon receiving a request for
sending a file, it initializes a FSTP packet and starts sending the file back to the client through
a UDP socket connection.
* In this version we set the inter-packet transmisson time manually. That's because of some
restrictions imposed by remote servers while trying to send out a burst of packets (i.e.: 10000
packets in our case), which is needed in our adaptive mechanism.
The original code is written by Steve Kretshmann, and modified by babak s. Noghani. Modifications
include:
1. Changing the single-threaded client to multi-threaded
2. Adding an additional field to the packet header
3. Adding the functionality of appending the file components
4. Disabling the adaptive flow control mechanism
5. Hard-coding the packet size

Last Modified: 22/11/99
**/

import java.io.*;
import java.net.*;

public class CBDFSTPServer_1 extends Thread
{
  // Class Variables
  static int defaultPacketSize = 1024;     // Port and defaultPacketSize cannot be changed
  static InetAddress localip;          //local ip address -- the same for all instances
  private static String root;
  private static String logText;
  private Socket incoming;
  private int counter;     //counts the number of running server threads

  public CBDFSTPServer_1(Socket income, int c) throws InterruptedException
  {
    incoming = income;
    counter = c;
  }

  public static void main(String[] args)
  {
    //find local ip
    try
    {
      localip = InetAddress.getLocalHost();
      System.out.println("Local IP Address = "+localip);
    }
    catch(UnknownHostException e)
    {
      System.err.println("Caught UnknownHostException: " +  e.getMessage());
    }
    // root directory remains the same regardeless of clients
    if(args.length != 0) root = args[0];
    else root = "/c:/users/babak/CBD_FSTP";
    int i = 0;
    System.out.println("CBD_FSTP_01 Server is Ready..");
    System.out.println("Root Dir = " + root);

    try
    {
      ServerSocket s = new ServerSocket(4712);
      for(;;)
      {
        Socket incoming = s.accept();
        new CBDFSTPServer_1(incoming,++i).start();
```

```
      }
      catch(Exception e){}
    }   }
    public void run()
    {   //beginning of run
      long timeDelay = 0;
      int packetSize = defaultPacketSize;
      int i,ip = 1,h1;
      InetAddress inet;
      int sNum = 1;
      long LoopsPerPacket = 80000;    //set the delay time manually
      int skip=0;
      String host, dir, comm, param;
      dir = root;

      try
      { //try to do eveything
        inet = incoming.getInetAddress();
        host = inet.toString();
        h1 = host.indexOf("/");
        host = host.substring(h1 + 1);
        BufferedReader in = new BufferedReader(new InputStreamReader(incoming.getInputStream()));
        PrintWriter out = new PrintWriter(incoming.getOutputStream(),true);
        boolean done = false;

        out.println("(File Component No"+sNum+") 120 CBD_FSTP_01 Server Ready.");
        out.flush();

        while(!done)
        { //begin while !done
          out.println("000 <"+counter+"> "+dir+" : ");
          out.flush();
          String str = in.readLine();
          if(str==null)break;
          if(str.length()>4)
          { //begin in str > 4
            comm=str.substring(0,4).trim().toUpperCase();
            param = str.substring(4).trim();
          }
          else
          {
            comm=str.toUpperCase();param="";
          } //end str > 4

          if(comm.equals("SEND"))
          {
            File f;
            if(param.startsWith("/"))
            {
              f=new File(root,param);
              System.out.println("100 Sending : "+root+param);
              out.flush();
            } //end if startsWith("/")
            else
            {
              f=new File(dir,param);
              System.out.println("100 Sending : "+dir+"/"+param);
              out.flush();
            } //end else startsWith("/");
            if (f.exists())
            {
              // Initialize an FSTPPacket & Clump
              Long LongFileLength = new Long(f.length());
              FSTPPacket fPacket = new FSTPPacket(param,
LongFileLength.intValue(),packetSize,inet,sNum);
              byte[] clump = new byte[fPacket.tag.clumpSize];
              //Open file
              FileInputStream outFile=new FileInputStream(f);
```

```
            int blockNumb = 0;
            //set the delay based on the manually chosen number of "loops per packet"
            fPacket.timeDelay = LoopsPerPacket;
            int amount = fPacket.tag.packetSize; //initial setting for send loop
            out.println("(File Component "+sNum+") 101 Sending "+ param);
            out.flush();
            sleep(500);
            System.out.println("Transmitting File "+param);
            System.out.println("LoopsPerPacket= "+LoopsPerPacket);
            long startSend = System.currentTimeMillis();
            for (blockNumb=0;blockNumb<fPacket.tag.numbClumps;blockNumb++)
            {
              //set block number
              fPacket.tag.setIntValue(blockNumb);
              //set data clump
              amount = outFile.read(clump);
              fPacket.setClump(clump);
              // send the packet
              fPacket.send(sNum);
            } //end while amount >= clumpSize
            long endSend = System.currentTimeMillis();
            outFile.close();
            System.out.println("Transfer Completed in "+(endSend - startSend)+" mS.");
            double perPacket = (endSend-startSend);
            perPacket /= fPacket.tag.numbClumps;
            System.out.println("Average Time per Packet ="+perPacket+" mS");
            out.println( "(File Component "+sNum+") 300 Transfer Completed in #"+(endSend -
startSend)+" mS.");
            skip=0;
            outFile.close();
          } // end if f.exists
          else out.println("(File Component No"+sNum+") 550 "+f.getName()+": no such file or
directory");
          out.flush();
        } // end if comm.equals('SEND')

        else if(comm.equals("RETR"))
        {
          File f;
          //strip the file name from the retransmission parametr
          //int startIndex = param.indexOf('"');
          System.out.println("param = "+ param);
          int startIndex = 9;
          String fileName = param.substring(0,startIndex);
          System.out.println("fileName = "+fileName);
          if(fileName.startsWith("/"))
          {
            f=new File(root,fileName);
          } //end if startsWith("/")
          else
          {
            f=new File(dir,fileName);
          } //end else startsWith("/");
          if (f.exists())
          {
            // Create new FSTPPacket
            Long LongFileLength = new Long(f.length());
            FSTPPacket fPacket = new FSTPPacket(fileName,LongFileLength.intValue(), packetSize,
inet,sNum);
            byte[] clump = new byte[fPacket.tag.clumpSize];
            //initialize checklist
            boolean checkList[] = new boolean[fPacket.tag.numbClumps];
            Integer tInteger = null;
            int reCount = 0;
            //Check off Retransmission Clumps
            while(startIndex + 1 < param.length())
            {
              int endIndex = param.indexOf(",",startIndex + 1);
              String stringTag = param.substring(startIndex+1,endIndex);
              try
```

```
                tInteger = new Integer(stringTag);
              }
              catch(NumberFormatException e)
              {
                System.out.println(e+ stringTag);
              }
              checkList[tInteger.intValue()] = true;
              reCount++;
              startIndex = endIndex;
            } //for k
            FileInputStream outFile=new FileInputStream(f);
            //retransmitt files in checklist
            out.println("(File Component "+sNum+") 111 Retransmitting #" +reCount+" Clump(s) from :
"+fileName);
            out.flush();
            System.out.println("Retransmitting " + reCount + " Clump(s) from :"+fileName);
            sleep(500);
            fPacket.timeDelay = LoopsPerPacket;
            if ((reCount<2000) && (reCount>100))
            {
              float fltReCount = (float) reCount;
              float fltModifier = 4*(2000-fltReCount)/1900;
              float fltLoopsPerPacket = (float)LoopsPerPacket;
              fPacket.timeDelay = (long)(fltLoopsPerPacket*fltModifier);
            }
            if (reCount<100) fPacket.timeDelay*=4;
            long startSend = System.currentTimeMillis();
            for(int m=0; m<fPacket.tag.numbClumps;m++)
            {
              if (checkList[m])
              {
                long amount = outFile.read(clump);
                fPacket.tag.setIntValue(m);
                fPacket.setClump(clump);
                fPacket.send(sNum);
              } //end if checked
              else outFile.skip(fPacket.tag.clumpSize);
            } //end for m
            long endSend = System.currentTimeMillis();
            double perPacket = (endSend-startSend);
            perPacket /= fPacket.tag.numbClumps;
            System.out.println("Retransfer Completed in "+(endSend - startSend)+" mS.");
            System.out.println("Average Time per Packet ="+((endSend - startSend)/reCount)+" mS");
            outFile.close();
            out.println("(File Component "+sNum+") 310 Retransmission Completed in #"+(endSend -
startSend)+" mS.");
          } // end if f.exists
          else out.println("(File Component No"+sNum+") 550 "+f.getName()+": no such file or
directory");
          out.flush();
        }//end of "RETR"
        else if(comm.equals("QUIT"))
        {
          out.println("(file component "+sNum+") 290 GOOD BYE");
          out.flush();
          done = true;
        }
        else out.println("(File Component No"+sNum+") 500 \""+str.substring(0,4)+"\" : command not
understood");
        out.flush();
            } //end while !done
            incoming.close();
        } //end try everything
        catch (Exception e)
        {
          System.out.println(e);
        }
    } //End Run()
}//end class
```

### 7.4.2.3 FSTPPacket

```
/*
The original code is written by Steve Kretshmann, and modified by babak s. Noghani. Modifications
include:
1. Changing the single-threaded client to multi-threaded
2. Adding an additional field to the packet header
3. Adding the functionality of appending the file components
4. Disabling the adaptive flow control mechanism
5. Hard-coding the packet size
*/
import java.net.*;
import java.io.*;

public class FSTPPacket extends Thread
{
  //parameters
  static int FSTPPort = 4711;
  private InetAddress inet0, inet1, inet2, inet3;
  private String fileName;
  private int segNum;
  private int port = FSTPPort;

  //calculated variables
  private int clumpOffset;
  public int tagOffset;

  //internal variables
  public FSTPTag tag;
  public boolean packetReceived = false;
  public long timeDelay = 0;
  public byte[] data;
  private int inetCounter = 0;
  public byte[] clump;
  private byte[] rTagNum;// received tag number
  private int timeout = 5000; // default timeout
  private DatagramPacket packet;
  private DatagramSocket socket;
  public  boolean recvTimedOut = false;
  private XMTDelayTicker delayTicker;


  public  FSTPPacket(String pFName, int pSize, int pPacketSize, InetAddress pInet, int pSNum) throws
SocketException
  {
    //copy parameters to instance variables
    inet0 = pInet;
    fileName = pFName;
    segNum = pSNum;

    //initialize data sorts of deals
    tag = new FSTPTag(pPacketSize, pFName.length(), pSize);
    data = new byte[tag.FSTPPacketSize];
    clump = new byte[tag.clumpSize];
    tagOffset = fileName.length() + 1;
    clumpOffset = tagOffset + tag.tagSize;

    // make a header for the packet
    //start with the File Name and '*' char
    String header = new String(fileName);
    Integer s = new Integer(segNum);
```

```
    header+=seg;
    header +='*';
    System.arraycopy(header.getBytes(),0,data,0,header.length());
    String seg = s.toString();
    rTagNum = new byte[tag.tagSize];

    //create socket for sending
    socket = new DatagramSocket();

    this.setPriority(MAX_PRIORITY);
    delayTicker  = new XMTDelayTicker();
  }
  void addIP(InetAddress fIP)
  {
    if(inetCounter==0) inet1 = fIP;
    else if(inetCounter==1) inet2 = fIP;
    else inet3 = fIP;
    if (inetCounter<3) inetCounter++;
  }
  void initReceivePort(int segNum) throws SocketException
  {
    //initialize recieve socket on "port"
    port+=segNum;
    socket = new DatagramSocket(port);
    socket.setSoTimeout(timeout);
    socket.setReceiveBufferSize(10000000);
    packet = new DatagramPacket(data, tag.FSTPPacketSize);
    recvTimedOut = false;
    System.out.println("FSTPPacket.initReceivePort: port# on client # "+segNum+" is: "+port);
  }
  void receive() throws SocketException, IOException
  {
    recvTimedOut = false;
    packetReceived = false;
    try
    {// wait for a packet to arrive until timeout
      socket.receive(packet);

      //verify that this is coming from a designated FSTP server
      InetAddress address = packet.getAddress();
      boolean correctSender = address.equals(inet0);
      System.out.println("correctsender: "+correctSender);
      if (inetCounter>0) {if (address.equals(inet1)) correctSender = true;}
      if (inetCounter>1) {if (address.equals(inet2)) correctSender = true;}
      if (inetCounter>2) {if (address.equals(inet3)) correctSender = true;}

      if (correctSender)
      {
        //verify that the packet is the right file name
        String recdName = new String(data, 0,fileName.length());
        if (recdName.equals(fileName))
        {
          //get packet number
          System.arraycopy(data, tagOffset ,rTagNum, 0, tag.tagSize);
          tag.setByteValue(rTagNum);

          //capture data clump
          System.arraycopy(data, clumpOffset, clump, 0, tag.clumpSize);

          packetReceived = true;
        }
      }
      else System.out.print("X");
    }
    catch(InterruptedIOException e)
    {
      recvTimedOut = true;
    }
  }
  void receiveRaw() throws SocketException, IOException
```

```
    packetReceived = false;
    recvTimedOut = false;
{   try
    {// wait for a packet to arrive until timeout
      socket.receive(packet);


      //verify that this is coming from a designated FSTP server
      InetAddress address = packet.getAddress();
      boolean correctSender = address.equals(inet0);


      if (inetCounter>0) {if (address.equals(inet1)) correctSender = true;}
      if (inetCounter>1) {if (address.equals(inet2)) correctSender = true;}
      if (inetCounter>2) {if (address.equals(inet3)) correctSender = true;}

      if (correctSender)
      {
        //verify that the packet is the right file name
        String recdName = new String(data, 0,fileName.length());
        if (recdName.equals(fileName)) packetReceived = true;
        else System.out.print(recdName + " ");

      }
      else System.out.print("Inet0 ="+ inet0 + "Incoming = "+packet.getAddress());
    }
    catch(InterruptedIOException  e)
    {
      recvTimedOut = true;
      System.out.println(" Timeout ");
    }
  }
  void send(int seg) throws SocketException, IOException
  {
    if (timeDelay != 0) delayTicker.XMTWait(timeDelay);

    //add tag to data packet
    System.arraycopy(tag.byteValue(),0,data,tagOffset, tag.tagSize);

    //add clump
    System.arraycopy(clump, 0,data, clumpOffset, tag.clumpSize);
    packet = new DatagramPacket(data, tag.FSTPPacketSize, inet0, port+seg);
    socket.send(packet);

  }
  void  setClump(byte[] SClump)
  {
    clump = SClump;
    packetReceived = false;
  }
  void  setData()
  {
    System.arraycopy(data, tagOffset ,rTagNum, 0, tag.tagSize);
    tag.setByteValue(rTagNum);
    System.arraycopy(data, clumpOffset ,clump, 0, tag.clumpSize);
  }
  void setDataByte()
  {
    System.arraycopy(data, tagOffset ,rTagNum, 0, tag.tagSize);
    tag.setByteValue(rTagNum);
  }
  void setPort(int fPort)
  {
    port = fPort;
  }
```

```
}
```

### 7.4.2.4 FSTPTag

```
/*
The original code is written by Steve Kretshmann, and modified by babak s. Noghani. Modifications
include:
1. Changing the single-threaded client to multi-threaded
2. Adding an additional field to the packet header
3. Adding the functionality of appending the file components
4. Disabling the adaptive flow control mechanism
5. Hard-coding the packet size
*/

public class FSTPTag {
  //parameters
  public int packetSize;
  public int fileLength;
  public int fileNameLength;

  //calculated sizes
  public int clumpSize;
  public int numbClumps;
  public int tagSize = 1;
  public int lastAmount;
  public int FSTPPacketSize;

  //internal variables
  public boolean byteSet = false;
  public boolean intSet = false;
  private byte[] tagByteValue;
  private int tagIntValue;
  public boolean last = false;

  //temporary variables declared once here to increase
  //the speed of the code (memory will not have to be
  //assigned each time the methods start
  private int block;
  private int shift;

  public FSTPTag(int pSiz, int fNameLength, int fLength)
  {
    //copy parameters to instance variables
    packetSize = pSiz;
    fileNameLength = fNameLength;
    fileLength =fLength;
    tagIntValue = 0;

    //FSTPPacketSize is total packetSize - Header Size
    //UDP Headers=8 Bytes
    // IP header=20 Bytes = 28 bytes header for UDP Packets
    FSTPPacketSize = packetSize - 28;

  //initial sizes
  tagSize = 1;
  clumpSize = FSTPPacketSize - 1 - fileNameLength -tagSize;
  numbClumps = fileLength / clumpSize;
```

```
  while ((twoPower8x(tagSize) < numbClumps) && (twoPower8x(tagSize) > 1))
  {
    tagSize++;
//iteratively calculate size of tag
    clumpSize = FSTPPacketSize - 1 - fileNameLength - tagSize;
    numbClumps = fileLength / clumpSize;
  } //end while calculation tagSize

  //calculate size of last clump and adjust numbClumps
  Integer IntegerLastAmount = new Integer(fileLength % clumpSize);
  lastAmount = IntegerLastAmount.intValue();
  if (lastAmount!= 0) numbClumps++;
  else lastAmount = clumpSize;

  tagByteValue = new byte[tagSize];
} // end FSTPTag initialization
final public byte[] byteValue()
{
  // This is written for speed
  // I fully realize this looks really bad, but it is
  // likely a small amount faster than implementing this in loops or
  // some other form.  This could make a difference when we are encoding
  // or decoding thousands of packets.

  if (!byteSet)
  {

    switch(tagSize)
    {
      case 4:
      block = tagIntValue & 255;
      if (block>127) tagByteValue[0] = (byte)(block - 256);
      else tagByteValue[0] = (byte)block;

      shift = tagIntValue >> 8;
      block = shift & 255;
      if (block>127) tagByteValue[1] = (byte)(block - 256);
      else tagByteValue[1] = (byte)block;

      shift = shift >> 8;
      block = shift & 255;
      if (block>127) tagByteValue[2] = (byte)(block - 256);
      else tagByteValue[2] = (byte)block;

      block = shift >> 8;
      if (block>127) tagByteValue[3] = (byte)(block - 256);
      else tagByteValue[3] = (byte)block;
      break;

      case 3:
      block = tagIntValue & 255;
      if (block>127) tagByteValue[0] = (byte)(block - 256);
      else tagByteValue[0] = (byte)block;

      shift = tagIntValue >> 8;
      block = shift & 255;
      if (block>127) tagByteValue[1] = (byte)(block - 256);
      else tagByteValue[1] = (byte)block;

      block = shift >> 8;
      if (block>127) tagByteValue[2] = (byte)(block - 256);
      else tagByteValue[2] = (byte)block;
      break;

      case 2:
      block = tagIntValue & 255;
      if (block>127) tagByteValue[0] = (byte)(block - 256);
      else tagByteValue[0] = (byte)block;

      block = tagIntValue >> 8;
```

```
          else tagByteValue[1] = (byte)block;
          break;
          if (block>127) tagByteValue[1] = (byte)(block - 256);
          case 1:
          if (tagIntValue>127) tagByteValue[0] = (byte)(tagIntValue - 256);
          else tagByteValue[0] = (byte)tagIntValue;
          break;
        }
        byteSet = true;
      } // end if !byteSet
      return tagByteValue;
    }
  public int intValue()
     {
      if (!intSet)
      { //This method was modified for faster speed
        //rather than compact programming structure

        block = 0;
        switch(tagSize)
        {
          case 4:
          block = (int)tagByteValue[3];
          if (block < 0) block += 256;
          block <<= 24;

          shift = (int)tagByteValue[2];
          if (shift < 0) shift += 256;
          shift <<=  16;
          block += shift;

          shift = (int)tagByteValue[1];
          if (shift < 0) shift += 256;
          shift <<= 8;
          block += shift;

          shift = (int)tagByteValue[0];
          if (shift < 0) shift += 256;
          block += shift;
          break;

          case 3:
          block = (int)tagByteValue[2];
          if (block < 0) block += 256;
          block <<= 16;

          shift = (int)tagByteValue[1];
          if (shift < 0) shift += 256;
          shift <<=  8;
          block += shift;

          shift = (int)tagByteValue[0];
          if (shift < 0) shift += 256;
          block += shift;
          break;

          case 2:
          block = (int)tagByteValue[1];
          if (block < 0) block += 256;
          block <<= 8;

          shift = (int)tagByteValue[0];
          if (shift < 0) shift += 256;
          block += shift;
          break;

          case 1:
          block = (int)tagByteValue[0];
          if (block < 0) block += 256;
```

```
        tagIntValue = block;
        intSet = true;
      }
      last = (tagIntValue == (numbClumps-1));
      return tagIntValue;
      }
    public void setByteValue(byte[] bTag)
    {
      byteSet = true;
      intSet = false;
      tagByteValue = bTag;
    }
    public final void setIntValue(int lTag)
    {
      byteSet = false;
      intSet = true;
      tagIntValue = lTag;
      last = (lTag == (numbClumps-1));
    }
    //The following method is no longer used
    //in this implementation.
    /*  public Integer decodeString(String stringTag)
      {
        int decoded = 0;
        for(int i=0;i<stringTag.length();i++)
        {
          Character c = new Character(stringTag.charAt(i));
          int hash = c.hashCode();
          if (hash < 0) hash = hash + 256;
          decoded=decoded + twoPower8x(i)*hash;
        }
        Integer IntegerDecoded = new Integer(decoded);
        return IntegerDecoded;
      }

    */
      static private int twoPower8x(int p)
      {
        int twoP = 1;
        for (int i=1;i<=p;i++) twoP *= 256;
        return twoP;
      }
}
```

## 7.4.2.5 TagBenchmark

```
/*
The original code is written by Steve Kretshmann, and modified by babak s. Noghani. Modifications
include:
1. Changing the single-threaded client to multi-threaded
2. Adding an additional field to the packet header
3. Adding the functionality of appending the file components
4. Disabling the adaptive flow control mechanism
5. Hard-coding the packet size
*/

import java.io.*;
public class TagBenchmark
  {
```

```
    {
      Integer Input = new Integer(args[0]);
      int numbTests = Input.intValue();
    public static void main(String[] args)
      int fNameLength = 1;
      int fLength = 2147483647;
      int pSiz = 15;
      byte byteValue[] = new byte[4];
      byte byteValue2[] = new byte[4];
      FSTPTag FTag = new FSTPTag(pSiz, fNameLength, fLength);
      FSTPTag2 FTag2 = new FSTPTag2(pSiz, fNameLength, fLength);
      System.out.println("TagSize = " + FTag.tagSize);
      System.out.println("NumbTests ="+numbTests);

      boolean bad = false;
      for (int i=0;i<numbTests;i+=17)
      {
        FTag.setIntValue(i);
        FTag2.setIntValue(i);
        byteValue = FTag.byteValue();
        byteValue2 = FTag2.byteValue();
        for (int j=0; j<FTag.tagSize;j++)
        {
          if (byteValue[j]!=byteValue2[j])
          {
            bad=true;
            System.out.print("X"+i+" ");
          }
        }
        FTag.setByteValue(byteValue);
        FTag2.setByteValue(byteValue);
        if (FTag2.intValue() != FTag.intValue())
        {
          bad=true;
          System.out.print("Q"+i+" ");
        }

      }

      if (!bad) System.out.println("Everything Matches Up");
    }
}
```

## 7.5 Trace Route

Below are the results of running TraceRoute application on the machines utilized during our tests of the CBD-FSTP.

**From University of Alberta to University of Manitoba**

1 ee-gw (129.128.68.1) 1.047 ms 0.917 ms 0.817 ms
2 canet2fddi.gw.ualberta.ca (129.128.1.19) 1.223 ms 1.451 ms 1.422 ms
3 206.75.91.17 (206.75.91.17) 6.211 ms 7.047 ms 5.929 ms
4 205.189.32.58 (205.189.32.58) 42.297 ms 46.806 ms 44.525 ms
5 205.189.32.81 (205.189.32.81) 84.251 ms 66.069 ms 68.320 ms
6 atrouter.cc.umanitoba.ca (207.161.242.18) 90.931 ms 66.387 ms 67.016 ms
7 bbrouter.cc.umanitoba.ca (130.179.16.210) 78.759 ms 76.125 ms 84.079 ms
8 ic14.ee.umanitoba.ca (130.179.8.80) 67.492 ms * 76.507 ms

**From University of Manitoba to University of Alberta**

1 enrouter.cc.umanitoba.ca (130.179.8.70) 1.070 ms 6.338 ms 0.854 ms
2 atrouter.cc.umanitoba.ca (130.179.16.1) 1.260 ms 0.730 ms 0.865 ms
3 mrnet.mbnet.mb.ca (207.161.242.17) 1.618 ms 1.854 ms 1.448 ms
4 205.189.32.82 (205.189.32.82) 24.553 ms 24.275 ms 24.061 ms
5 wnet-ab.canet2.net (205.189.32.57) 61.399 ms 61.289 ms 61.332 ms
6 206.75.91.18 (206.75.91.18) 66.445 ms 65.697 ms 65.716 ms
7 gsb04.gw.ualberta.ca (129.128.1.1) 71.586 ms 77.877 ms 74.094 ms
8 548pc15.ee.ualberta.ca (129.128.68.184) 66.665 ms 66.769 ms 66.654 ms

---

**From University of Calgary to University of Manitoba**

1 fivegate (136.159.5.1) 0.828 ms 0.630 ms 0.580 ms
2 towergate (136.159.28.1) 1.279 ms 0.814 ms 0.819 ms
3 campus (136.159.30.1) 1.238 ms 0.878 ms 0.855 ms
4 136.159.251.2 (136.159.251.2) 1.307 ms 1.002 ms 0.957 ms
5 192.168.47.1 (192.168.47.1) 1.355 ms 1.140 ms 1.001 ms
6 192.168.46.10 (192.168.46.10) 1.455 ms 2.046 ms 1.406 ms
7 205.189.32.58 (205.189.32.58) 38.281 ms 38.398 ms 38.189 ms
8 205.189.32.81 (205.189.32.81) 62.915 ms 62.193 ms 62.268 ms
9 atrouter.cc.umanitoba.ca (207.161.242.18) 62.451 ms 63.364 ms 62.294 ms
10 bbrouter.cc.umanitoba.ca (130.179.16.210) 63.261 ms 65.198 ms 64.212 ms
11 mcleod2.ee.umanitoba.ca (130.179.8.25) 62.781 ms * 62.491 ms

**From University of Manitoba to University of Calgary**

traceroute to csc.cpsc.ucalgary.ca (136.159.5.16), 30 hops max, 40 byte packets
1 enrouter.cc.umanitoba.ca (130.179.8.70) 1.142 ms 1.078 ms 0.905 ms
2 atrouter.cc.umanitoba.ca (130.179.16.1) 0.767 ms 0.829 ms 0.688 ms

3 mrnet.mbnet.mb.ca (207.161.242.17) 2.168 ms 1.799 ms 1.787 ms
4 205.189.32.82 (205.189.32.82) 24.229 ms 24.436 ms 25.652 ms
5 wnet-ab.canet2.net (205.189.32.57) 62.159 ms 61.791 ms 61.251 ms
6 192.168.46.9 (192.168.46.9) 61.723 ms 62.275 ms 61.627 ms
7 192.168.47.3 (192.168.47.3) 62.241 ms 64.860 ms 62.421 ms
8 136.159.251.1 (136.159.251.1) 62.270 ms 62.302 ms 61.938 ms
9 136.159.30.2 (136.159.30.2) 62.194 ms 63.677 ms 61.972 ms
10 tsa.cpsc.ucalgary.ca (136.159.28.2) 63.006 ms 62.625 ms 62.133 ms
11 csc.cpsc.ucalgary.ca (136.159.5.16) 62.626 ms * 70.946 ms

---

**From University of Regina to University of Manitoba**

1 NET-ED-UOFRGATE.CC.UREGINA.CA (142.3.1.1) 1.748 ms 0.691 ms 0.699 ms
2 142.165.3.105 (142.165.3.105) 1.295 ms 1.360 ms 0.946 ms
3 205.189.32.54 (205.189.32.54) 30.506 ms 30.442 ms 32.426 ms
4 205.189.32.81 (205.189.32.81) 54.933 ms 54.448 ms 53.937 ms
5 atrouter.cc.umanitoba.ca (207.161.242.18) 55.226 ms 54.845 ms 58.264 ms
6 bbrouter.cc.umanitoba.ca (130.179.16.210) 55.296 ms 55.028 ms 55.078 ms
7 mcleod2.ee.umanitoba.ca (130.179.8.25) 55.674 ms * 56.953 ms

**From University of Manitoba to University of Regina**

1 enrouter.cc.umanitoba.ca (130.179.8.70) 1.038 ms 0.776 ms 0.689 ms
2 atrouter.cc.umanitoba.ca (130.179.16.1) 1.408 ms 0.726 ms 0.585 ms
3 mrnet.mbnet.mb.ca (207.161.242.17) 1.831 ms 1.343 ms 1.302 ms
4 205.189.32.82 (205.189.32.82) 24.049 ms 25.149 ms 24.571 ms
5 srnet-sk.canet2.net (205.189.32.53) 54.088 ms 53.891 ms 53.878 ms
6 142.165.3.106 (142.165.3.106) 54.658 ms 54.679 ms 54.677 ms
7 IVANHOE.ENGG.UREGINA.CA (142.3.212.1) 55.507 ms * 56.806 ms