Efficient Bulk Data Transfer with the Phatpackets Protocol

by

Sheng Huang

A Thesis

Submitted to the Faculty of Graduate Studies

in Partial Fulfillment of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

University of Manitoba

Winnipeg, Manitoba, Canada

© Sheng Huang, 2005

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION

Efficient Bulk Data Transfer with the Phatpackets Protocol

BY

Sheng Huang

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of

of Manitoba in partial fulfillment of the requirement of the degree

of

Master of Science

SHENG HUANG © 2005

Permission has been granted to the Library of the University of Manitoba to lend or sell copies of this thesis/practicum, to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film, and to University Microfilms Inc. to publish an abstract of this thesis/practicum.

This reproduction or copy of this thesis has been made available by authority of the copyright owner solely for the purpose of private study and research, and may only be reproduced and copied as permitted by copyright laws or with express written authorization from the copyright owner.

ABSTRACT

Transferring files over the Internet is extremely common. The most common application and protocol for transmitting a file is known as FTP (File Transfer Protocol). However, the performance of FTP over TCP is far less than optimized. This is evidenced by the continuous effort to improve applications such as FTP and protocols such as TCP. Phatpackets is a term used to describe an initiative to reliably move large amounts of data more efficiently across the Internet. This thesis describes the design, implementation, and experimentation of the Phatpackets protocol. The Phatpackets protocol attempts to combine the best aspects of a variety of transport and control methods with the objective of developing a near optimal and robust protocol. The Phatpackets protocol uses multiple connections with multiple server sites transferring different segments of a file simultaneously to enhance the performance, scalability and availability. Experiments presented show that the Phatpackets protocol is both efficient and reliable.

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank all the people who have contributed towards this thesis.

First of all, I would like to greatly thank my advisor, Dr. Robert D. McLeod, for his guidance, advice and assistance throughout my academic years as well as his contributions and help with this thesis. In addition, I would like to express my appreciation for his kind and generous personality.

Finally, heartfelt appreciation is due to my family members — my parents, Yue Wan and Bingyi Huang, and my wife, Zhixin Duan. Their consistent encouragement and support have made this academic endeavor more enjoyable.

TABLE OF CONTENTS

Chapter 1

INTRODUCTION

This thesis presents an alternative transport layer in the TCP/IP protocol stack called the Phatpackets protocol. The Phatpackets protocol is an attempt to combine the best aspects of a variety of transport and control methods with the objective of developing a near optimal and robust protocol. This thesis describes the algorithm developed, and a file transfer application illustrating the algorithm's effectiveness.

The Phatpackets protocol is designed to meet the requirements of today's computing environment. It is a distributed system. It works on a wide geographic distributed network and heterogeneous systems. It simultaneously transfers a target file from multiple file servers with multiple connections from each server. The Phatpackets protocol is based on TCP and UDP. The platform independent language Java was chosen as the programming language to implement the Phatpackets protocol. With these two features, the Phatpackets protocol can work on most computers throughout the largest distributed environment, the Internet.

1.1 Evolution of Internet [Tec02]

The Internet started in 1969 as the ARPAnet. Funded by the U.S. government, the ARPAnet became a series of high-speed links between major supercomputer sites and educational and

research institutions worldwide, although mostly in the U.S. A major part of its backbone was the National Science Foundation's NFSNet. Along the way, it became known as the "Internet".

In 1995, the Internet was turned over to large commercial Internet Service Providers (ISPs), such as MCI, Sprint and UUNET, who took responsibility for the backbones and have increasingly enhanced their capacities ever since. Regional ISPs link into these backbones to provide lines for their subscribers, and smaller ISPs hook either directly into the national backbones or into the regional ISPs.

The Internet's surge in growth in the latter half of the 1990s was twofold. As the major online services (AOL, CompuServe, etc.) connected to the Internet for e-mail exchange, the Internet began to function as a central gateway. The Internet glued the world together for electronic mail, and today, the Internet mail protocol is the world standard.

Secondly, with the advent of graphics-based Web browsers such as Mosaic and Netscape Navigator, and soon after, Microsoft's Internet Explorer, the World Wide Web took off. The Web became easily available to users with PCs and Macs rather than only scientists and hackers at UNIX workstations. Delphi was the first proprietary online service to offer Web access, and all the rest followed. At the same time, new Internet service providers rose out of the woodwork to offer access to individuals and companies. As a result, the Web has grown exponentially providing an information exchange of unprecedented proportion. There has been more activity, excitement and hope over the Internet than any other computer or communications topic that was ever conceived.

Nowadays, the Internet is made up of more than 285 million computers in more than 100 countries covering commercial, academic and government endeavors while there were only about 3.2 million Internet hosts in July 1994 [ISC05].

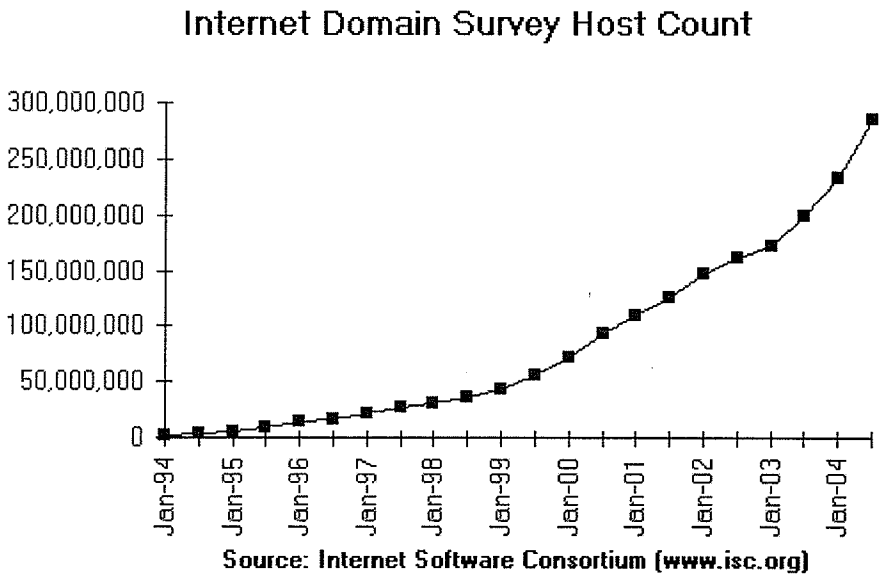


Figure 1. Internet Domain Survey Host Count

1.2 File Transfer Application

File Transfer Protocol (FTP) is the application that transfers files between FTP clients and FTP servers on the Internet. It has been used since 1971. The widely accepted standard of FTP is defined in RFC959 (Request For Comments) by the Internet Engineering Task Force (IETF) in 1985 [PR85]. It specified the application that allows a user to retrieve files from a remote FTP

server. FTP relies on TCP and the reliability of TCP provides reliability of FTP itself. Recent RFCs have developed FTP's security features [HL97] and FTP over IPv6 [AOM98].

Today, FTP is the second most widely used application on the Internet in terms of traffic. Nearly every software company has their own FTP site for their customers to download the latest products or patches. Many famous web sites such as www.download.com and www.tucows.com provide a service that collects popular download files over the Internet. Some search engines also have dedicated FTP search engines that search through the Internet for a file. One example is Lycos's search site. Because the demand for downloading a file is throughout the Internet, FTP server mirroring is widely implemented. The mirror sites contain the same copies of files as on the original server. This approach generally can keep the traffic local and achieve better performance [Fan00].

1.3 Motivation

1.3.1 Statement of Problems

1.3.1.1 Problems with Current FTP Implementation

FTP is widely used and easy to implement. However, it has problems with reliability, scalability, and performance. First, when a server is brought down for upgrade or maintenance, all current users will lose their connections and the FTP sessions will be interrupted. Secondly, the load may be imbalance. That is, while some servers are quite busy, other mirror sites may be quite idle in contrast. Thirdly and most importantly, FTP performance is relatively poor

because of TCP's working mechanism. Nowadays, there are a lot of large files such as voice, movies, and applications on Internet, which may be hundreds of Megabytes in size. Traditional FTP downloading may take an unreasonable time [Fan00].

1.3.1.2 QoS of Internet Applications

The quality of service (QoS) for an Internet application (especially a real-time application) is determined by packet loss and jitter [Sch97]. The Internet applications nowadays are implemented with transport protocols namely TCP and UDP. TCP guarantees zero packet loss through its build-in mechanism and is generally not applicable to real-time application. While the UDP based applications need to implement their own flow control/congestion control and retransmission mechanism.

a) Packet Loss

The packet loss is the fraction of packets that do not arrive at their destination. The reasons for packet loss are line failures and physical errors, receiver overflow, and network congestion. Damage to the packet due to a transmission error is becoming less common in modern fiber networks. If the checksum check fails, the packet is simply discarded. Packets with an undefined format are also dropped. The packets can also be lost due to link failures.

Another important reason for packet loss is the overflow of end hosts. If the sender can not get enough feedback information about the receiver's performance, load as well as available buffer space, packet loss can occur due to overflow of the receiver's buffer if the server is sending at a speed faster than the receiver's processing capability.

Far more often, packets are dropped at routers because of network congestion. A router usually has incoming interface buffers, system buffers and outgoing interface buffers. Large queuing delays are experienced as the packet-arrival rate nears the link capacity. Depending on where a packet is dropped, it is called an input drop or an output drop. Input drops usually occur when the router can't process packets fast enough, while output drops occur when the outgoing link is too busy. Routers also try to avoid congestion by dropping packets before the queues have reached their maximum length. This mechanism is called random early detection (RED). It causes TCP sources to back off and slow-start, thus temporarily reducing the amount of traffic going through the router. However, as the UDP portion in Internet traffic becomes larger, this mechanism becomes less effective and unfair, since only TCP sources back off. When the amount of data exceeds the capacity of the network, the overflow is stored in router buffers. The higher the buffer utilization in the network, the higher the probability of packet loss. If there is not enough buffer space in the routers, queue overflows occur. The main causes for network congestion are as follows [Wid00]:

- ✧ Long distance, high-speed connections increase the total amount of data in the network.
- ✧ Bursts of network traffic can cause temporary congestion even when the average load on the network is not critical.
- ✧ Unfavorable topologies and mismatching link speeds also contribute to congestion, for example slow routers that connect several traffic intense hosts to the Internet.
- ✧ Some protocols do not sufficiently decrease the sending rate when congestion occurs. Applications using such protocols pose an increasing problem as they become more

widespread. The Internet relies on the cooperation of communication end-points to prevent a congestion collapse.

- ✧ The fast growth of the Internet will be a problem if the supporting infrastructure is unable to keep up with the increase in network traffic. Over the past twenty years, the number of users and the speed of their connections to the Internet have increased by several orders of magnitude. In addition, the usage of the Internet has grown with an increasing number of services being replaced or supplemented with their e-counterparts (e.g. e-mail, e-business, etc.). Existing applications are upgraded to support higher bandwidths. New high volume applications that were previously possible only in local area networks have been moved to the Internet.

b) Jitter

Some authors understand jitter as the difference between the longest and the shortest delay in some period of time. Others define jitter as the maximum delay difference between two consecutive packets in some period of time. In [Sch97], jitter is defined as a smoothed function of the delay differences between consecutive packets over time. Jitter can be introduced by both hosts and routers. If sending hosts and receiving hosts have more than one task to do as is common in multi-process OS environments, they can also induce jitter. However, the most common cause for jitter is introduced by routers.

Queues build up in a router or switch whenever the input rate is larger than the bandwidth of the outgoing link or the processing speed of the interface. If all packets of a flow encounter the same queues and queue lengths on the path, they all wait for about the same time. The end-to-

end delay may be high, but there is no delay variance. Jitter comes into play when consecutive packets experience different waiting periods in the queues. If packet scheduling is done in strict first-in first-out manner, a difference in delay means that a queue grew or shrunk between arrivals of two consecutive packets. Imminent packet loss can be predicted by longer delay and larger delay variance.

1.3.2 Objective of the Phatpackets Protocol

The Phatpackets protocol is designed to solve these problems. The objective of the Phatpackets protocol is to improve transmission quality and performance when moving large files across the Internet or an Intranet. It can achieve reliability, scalability and efficiency by making better use of all the Phatpackets protocol servers across the Internet. It may be used with stream media applications by adopting efficient congestion and flow mechanism in the Phatpackets protocol to prevent congestion at an early phase thus reduce the possibility of packet loss and jitter.

The Phatpackets protocol is built on top of UDP and TCP and provides a reliable connectionless protocol for the transfer of large files. The Phatpackets protocol achieves its goal of improved transmission quality and performance by adapting parameters associated with the flow of data packets within the process of a file transfer. These include flow rate, packet size, number of packets sent at once, packet reception rate, analysis window size, number of connections, and number of collaborating servers.

The main goals that the Phatpackets protocol will achieve include:

Efficiency

A client should download multiple segments simultaneously from a group of servers at same time to improve efficiency.

Adaptability

The Phatpackets protocol should attempt to maximize throughput and minimize packet loss across all connections for all applications within the resource limits of the hosts by combining network status probing with server/client host load adaptation, and load balancing techniques. That is, load should be well distributed as a function of link bandwidth and server/client performance and load variation.

Scalability

A new server should be able to be added at any time and anywhere. Load can be distributed to the new server as long as it registers itself with other servers.

Compatibility

The Phatpackets protocol should be compatible with the existing Internet infrastructure.

Transparency

The file segmentation, load distribution, and failure recovery process should be transparent to users.

Reliability

When some servers or links are down, the Phatpackets protocol should be able to distribute the load to other available servers.

Summary

This chapter gave a brief background on evolution of Internet and file transfer application. Based on that, the problems of current file transfer application are presented. Some of the reasons are poor performance and load allocation. Therefore, the Phatpackets protocol is presented. The last section describes the concepts and the objectives of the Phatpackets protocol.

Chapter 2

RELATED WORK

This chapter introduces some related work on the efficiency and reliability of bulk data transferring. Today's Internet potentially offers gigabit-per-second bandwidth. While the national high-performance network infrastructure has grown tremendously both in bandwidth and accessibility, it is still common for applications, hosts, and users to be unable to take full advantage of this new and improved infrastructure. Such network performance problems are mostly caused by poorly designed and poorly implemented commercial host software in Layers 3 through 7 of the Open System Interconnection (OSI) Reference Model [HM102].

The July 1999 report from the Advanced Networking Infrastructure Needs in the Atmospheric and Related Sciences (ANINARS) Workshop stated: "The workhorse networking application in the atmospheric community is still FTP. FTP is practically the only networking tool used to construct applications in this scientific discipline. There was also a universal cry for FTP to actually deliver the available network bandwidth to the end-user. The lament was that the bandwidth actually obtained is much lower than the apparently available bandwidth." This same report continued: "Atmospheric science is fundamentally dependent upon networking technology. Programs should be developed that foster improvement to networking in the following area: FTP (or FTP-like) bulk data-transfer is the most important networking function used to construct applications in this scientific discipline, yet failure to achieve effective

bandwidths equal to apparently available bandwidths is most evident with bulk data transfer applications. A variety of host-software problems contribute to this failure, and programs should be developed to help solve these problems." This inefficiency is not unique to just FTP, similarly poor results are seen with almost all other out-of-the-box networking software.

TCP is extensively used and has an impressive track record for reliable communication. However there is clearly considerable room for improving transport layer performance. This is evidenced by the continual effort to improve applications such as FTP (e.g. TFTP [Sol92], WEB100 FTP [Web02], GridFTP [Gri01]), and protocol improvements within TCP (e.g. per-destination MTU-discovery [Dee90], "Large Windows" extensions to TCP [JBB92]).

2.1 File Transfer Protocol

2.1.1 Introduction to FTP

FTP Sessions maintain two connections between an FTP client and an FTP server. One connection is control connection and the other is data connection. The control connection is connected to the well-known TCP port 21 on the server end. While the data connection port is connected to port 20 on the server end.

In FTP, the packet level error detection and recovery is carried out by TCP. However, at the application level, FTP has the checkpoint restart mechanism that user can resume downloading from a given checkpoint. This is accomplished by using REST command of FTP.

2.1.2 Commands and Semantics of FTP

RFC959 defines the widely implemented FTP protocol [PR85]. The commonly used FTP commands include USER, PASS, PORT, TYPE, RETR, REST, SIZE, etc. An FTP reply consists of a three-digit number followed by some text. The number is intended for use by automata to determine what state to enter next; the text is intended for the human user. The three digits of the reply each have a special significance. This is intended to allow a range of very simple to very sophisticated responses by the user-process. The first digit denotes whether the response is good, bad or incomplete. An unsophisticated user-process will be able to determine its next action (proceed as planned, redo, retrench, etc.) by simply examining this first digit. A user-process that wants to know approximately what kind of error occurred (e.g. file system error, command syntax error) may examine the second digit, reserving the third digit for the finest gradation of information.

There are five values for the first digit of the reply code:

- ✧ 1yz Positive Preliminary reply
- ✧ 2yz Positive Completion reply
- ✧ 3yz Positive Intermediate reply
- ✧ 4yz Transient Negative Completion reply
- ✧ 5yz Permanent Negative Completion reply

In the second digit the following function groupings are encoded:

- ✧ x0z Syntax - These replies refer to syntax errors, syntactically correct commands that don't fit any functional category, unimplemented or superfluous commands.

- x1z Information - These are replies to requests for information, such as status or help.
- x2z Connections - Replies referring to the control and data connections.
- x3z Authentication and accounting - Replies for the login process and accounting procedures.
- x4z Unspecified as yet.
- x5z File system - These replies indicate the status of the Server file system vis-à-vis the requested transfer or other file system action.

The third digit gives a finer gradation of meaning in each of the function categories, specified by the second digit.

2.1.3 Improved FTP Products

The traditional FTP is the command line FTP client and FTP server shipped with the operating system. There are a lot of improved FTP products in market nowadays. Some of these products are CuteFTP and WS_FTP on Windows, and NcFTP and gFTP on Linux. They improve functionality, security and confidentiality, and efficiency of file transfer.

Some products have features such as bookmarks, command histories, support for recursive gets, downloading entire directories and subdirectories, queues for tasks, automatic resumption, and automatic logins. Most of them have easy-to-use graphical client. Some products support HTTP and SSH protocols, fxp third-party file transfers (transferring files between 2 remote servers via ftp), FTP and HTTP proxy server, passive and non-passive file transfers, and multithreading to allow for simultaneous downloads. Some products are firewall friendly so

that they can be configured to transfer files through firewall configuration. Some products even provide synchronization feature, which allows users to "mirror" directory structures between a local system and a remote FTP server with minimal intervention.

2.1.4 Web100 Project [Web02]

The Web100 initiative includes instrumentation allowing the tuning of TCP performance with considerable effort on better estimating parameters such as the bandwidth delay product and effecting transport layer performance through the TCP-MIB. Performance diagnoses are made at the sender and receiver ends of a network connection and at any point along the network path. Web100 achieves TCP performance tuning transparency and thus automates the network tuning. It refines TCP software in the Linux operating system, through open standard and open source manner so that they can be ported to other operating systems. Web100 facilitates adoption by the commercial community by using the IETF standards process to standardize any modifications to the TCP-MIB. It adopts Automatic Bandwidth Delay Product (BDP) Discovery, which is used to specify the simplified maximum TCP window size for each TCP session.

2.1.5 GridFTP project [Gri01]

The GridFTP project is an initiative aiming to produce a high-performance, secure, reliable data transfer protocol optimized for high-bandwidth wide-area networks. The GridFTP protocol is based on FTP and the researchers implemented a subset of the existing FTP standard, enhanced it with some extensions defined already in IETF RFCs and added a few additional features to meet requirements from current data grid projects such as automatic

negotiation of TCP buffer/window sizes, Grid Security Infrastructure (GSI) security on control and data channels, multiple data channels for parallel transfers, partial file transfers, third-party transfers, reusable data channels, and support for reliable data transfer.

2.2 TCP

TCP (Transmission Control Protocol) is a reliable, connection-oriented transport protocol. TCP establishes a connection by three-way handshake. TCP provides flow control to its applications to eliminate the possibility of the sender overflowing the receiver's buffer. TCP provides congestion control to eliminate the possibility of the sender overflowing of router buffers as the network becomes congested.

Flow control and congestion control are accomplished by the use of sliding window protocol. The size is defined in terms of MSS (Maximum Segment Size) segments. MSS is the maximum amount of data that can be grabbed and placed in a segment. The MSS depends on the TCP implementation (determined by the operating systems) and can often be configurable. The typical MSS is 536 or 1460 bytes. These segment sizes are chosen in order to avoid IP fragmentation. The receiver-advertised window value reflects the approximate amount of data that the receiver is capable of receiving without having its receive buffer overflow. In a full-duplex connection, the sender at each side of the connection maintains a distinct receive window. The congestion window is the window used by the sender and it represents the amount of data the sender can transmit before receiving an acknowledgement. The actual

window used by the sender is the minimum of the congestion window and the receiver-advertised window.

The flow control mechanisms employed by TCP are Retransmission Time-out (RTO) mechanism and the Fast Recovery and Fast Retransmission mechanism with some schemes. The congestion control mechanisms have two phases: slow start and congestion avoidance.

In slow start, the congestion window starts off with a small size, typically one MSS. The congestion window increases in size by one MSS for every acknowledgement received, resulting in an exponential congestion window growth rate. The sender starts by transmitting one segment and waiting for its ACK. When that ACK is received, the congestion window is incremented from one to two, and two segments can be sent. When each of those two segments is acknowledged, the congestion window is increased to four. This provides an exponential growth, although it is not exactly exponential because the receiver may delay its ACKs, typically sending one ACK for every two segments that it receives [Ste97].

Once the congestion window size reaches the value of the slow start threshold (ssthresh), slow start mode terminates and congestion avoidance mode commences. The ssthresh is initially set to the value of the receiver-advertised window at the start of the connection. Congestion avoidance phase increases the congestion window size by one for every RTT thus resulting in a linear sliding window growth rate.

Every segment sent by TCP has a timer associated with that segment. The timeout value associated with each TCP segment is a smoothed estimation of the Round Trip Time (RTT) plus some variation. While duplicate acknowledgements, typically three, trigger fast retransmission. The congestion window will be halved as opposed to setting it to one MSS as with slow start, and ssthresh is set to the new congestion window size. This is referred to as fast recovery.

The topic on TCP performance has been studied quite extensively since the birth of TCP. Several of the most important proposals and initiatives are as follows:

2.2.1 Large Initial Congestion Window [AFP98]

TCP uses slow start algorithm at the beginning of a new connection, after congestion has been detected or after long idle periods. Starting from a larger size would improve efficiency during the startup phase of a connection and an optional initial window size of two has already been proposed by [APS99]. Experiments have also been done with initial window size of 3 and 4, which showed improved performance although a small increase in dropped packets was also observed.

2.2.2 ssthresh Selection

Choosing an appropriate ssthresh value can eliminate the problem of having slow start not end soon enough leading to congestion of the link and dropped packets. One algorithm proposed to determine appropriate ssthresh values is the packet pair algorithm in combination with the measured round-trip time proposed by [Hoe96]. The algorithm estimates the link bandwidth by

observing the spacing of the ACKs in the reverse link, and together with the estimated RTT. Then the BDP is used as ssthresh value.

2.3 UDP

UDP (User Datagram Protocol) is the other important transport protocol besides TCP. UDP is commonly used with multimedia applications. It is also used in Domain Name Service (DNS), Simple Network Management Protocol (SNMP), and Network File System (NFS) version 1 and 2.

UDP is an unreliable transport protocol. Aside from the multiplexing/demultiplexing function and some light error checking, UDP adds nothing to IP. In an Internet environment, a message can be duplicated, delayed, lost, corrupted or delivered out of order. UDP is an unreliable protocol because it has no retransmission facility, nor the sender is informed if failure occurs. The checksum field in UDP header is calculated over the UDP segment to eliminate the possibility of wrong delivery. Although UDP provides error checking, it does not do anything to recover from an error. Some implementations of UDP simply discard the damaged segment; others pass the damaged segment to the application with a warning.

UDP is a connectionless protocol. There is no connection establishment and no connection state. So less resource is needed comparing with TCP. There is small packet header overhead because UDP has 8 bytes of header whereas TCP segment has 20 bytes of header.

UDP does not have flow control and congestion control. The speed at which UDP sends data is only constrained by the rate at which the application generates data, the capability of the source (CPU speed, memory size, etc.), and the access bandwidth of the Internet. If the sender overflows the network or receiver, the message will simply be discarded. For a typical UDP implementation, UDP will append the received segments in a finite-size queue that "precedes" the corresponding socket. The process reads one entire segment at a time from the queue. If the process does not read the segments fast enough from the queue, the queue will overflow and segments get lost. Unlike TCP, UDP is a packet protocol because it maintains message boundaries. It also allows many-to-many communication as in multicast and broadcast.

2.4 Load Balancing

Load balancing in distributed systems has been the subject of research for the last few decades. A large number of schemes and implementations have been presented since the birth of the Internet. The most recent work are mainly in a Web context. A number of approaches have been developed to provide transparent access to multi-server Internet services including HTTP redirect, DNS aliasing, mobile agents, and Active Networks. Network load balancing algorithms have been incorporated in operating systems (e.g. Windows 2000 Advanced Server and Datacenter Server ([NLB00]), Web Servers (e.g. Apache [Jse99]) and Resource Brokers (e.g. IBM Websphere Network Dispatcher [IBM02]).

There are four major network load balancing approaches: DNS-based (Constant and Dynamic TTL algorithms), client-based (e.g. Smart Client [YCE97] and Client side proxy [BBM97]),

server-based, and dispatcher-based [Aga01][Eng98]. DNS-based load balancing is affected adversely by remote and intermediate Domain Name Server caching, hence dispatcher-based, client-based and server-based approaches are preferable. Performance comparison for several server selection schemes – round robin, random, weighted capacity, and nearest cluster as a function of RTT and server performance/load has been studied in [Aga01].

Summary

This chapter gave a brief background on FTP, TCP and UDP. The problems associated with current FTP applications that are unable to deliver the available network bandwidth to the end-user are presented. Some initiatives to improve the performance of FTP and TCP are briefly described. In section 3, the lightweight and unreliable transport protocol UDP is introduced. While the national high-performance network infrastructure has grown tremendously both in bandwidth and accessibility, network performance problems are arguably caused by poorly designed and poorly implemented commercial host software in Layers 3 through 7 of the OSI Reference Model. The Phatpackets protocol is an initiative to solve these problems.

Chapter 3

THE PHATPACKETS PROTOCOL ARCHITECTURE

3.1 Introduction to the Phatpackets Protocol [HSM01]

Transmitting a file from a server to a client requires negotiating a "connection" between the client and server to determine a number of details prior to actually transmitting the file. These include establishing port numbers, access, and the name of the file to be transmitted. Once these parameters are set the actual task of downloading begins. The most common application and protocol for transmitting a file is known as FTP over TCP. From the application perspective the server basically writes the file to a socket stream and the client reads from a stream. TCP takes care of all the network detail such as end-to-end flow control and requesting the retransmission of any packets that may have been lost or corrupted.

The Phatpackets approach is radically different addressing the issue more explicitly through the development and experimentation with improving the transport layer directly. This chapter outlines the basic algorithms and parameters that are adapted and how they are adapted. The emphasis is on the algorithm during the actual transfer process as opposed to the procedures for establishing connections and selecting files to download. It should be noted that in the process of initializing the actual transfer none of the data received is lost. The Phatpackets protocol adapts these parameters at the beginning and during a transfer to achieve maximum throughput

within the resource limits of the collaborating hosts. If resources permit, it enables the end hosts to run applications at 100% of the available bandwidth, regardless of the magnitude of a network's capability. As such it is classified as a protocol for end-to-end performance enhancement.

The Phatpackets algorithm is based on the notion that the network is a black box. End-systems determine the network's state by probing for the network state and dynamically adjust the load on the network. This method is appropriate for pure best-effort data that has little or no sensitivity to delay or loss of individual packets. In addition, the Phatpackets protocol tries to minimize the impact of losses from a throughput perspective. The following parameters are the most relevant to understanding the Phatpackets protocol:

Flow rate: A basic parameter to adapt to is the flow rate. The flow rate controls the rate at which a server delivers packets to a client. This is adapted during transport to ensure a reasonable reception rate. The client compares the number or packets received with the number expected within an analysis window. The calculated flow rate is sent back to the server to adjust the flow to maximize the bandwidth of the connection. In effect the rate is set such that packet loss will occur in controlled manner.

Packet size: The packet size is the number of data bytes sent at once. An ideal packet size is one that adapts to the maximum allowed on a connection without IP fragmentation. This is accomplished in a similar manner to the adjustment for the block size (see below). The only

difference is that once adapted to the optimal packet size no further adjustment is required for the session.

Block size and block delay: The number of packets sent at once or more precisely the number of packets sent immediately one after another represents the block size. From here on this is defined as the block size. Following each burst of packets the server waits a specific amount of time determined by the flow rate. From here on this is defined as the block delay. The block size and block delay are adaptive and are adjusted to optimize the effective available bandwidth. They are adjusted as a function of packet drop rate, packet reception rate, delay variance, and ssthresh to acquire maximum bandwidth efficiency. The Phatpackets protocol avoids creating bursty traffic that degrades network throughput.

Packet reception rate: The packet reception rate is the ratio of the number of received packets to that of sent packets within a given period. The desired reception rate (alpha) is a value between 0 and 1 preset by the user and is adapted during the transfer as a function of alpha value and current packet reception rate. The reception rate is used to control the flow rate as well as in determining the block size and the block delay.

Analysis window size: The analysis window impacts performance by allowing the averaging of more or less data. Control information is sent from the client to the server at a rate determined by the analysis window. The control information should be fine-grained enough for efficient transmission while not introducing feedback implosion problems that affects scalability.

Figure 2 illustrates the basic process of sending and subsequently retransmitting missing packets. The figure illustrates the mode where a complete file is transmitted followed by the retransmission of missing packets and subsequently the retransmission of any packets dropped during retransmission. The actual process operates with missing packet notification occurring asynchronously.

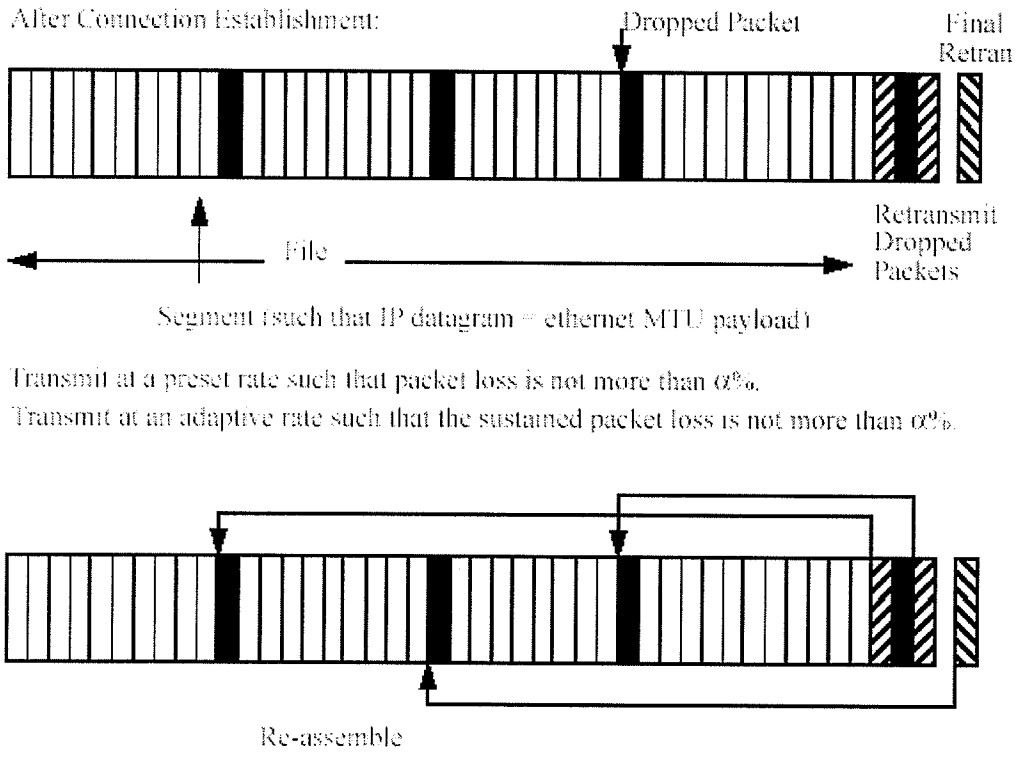
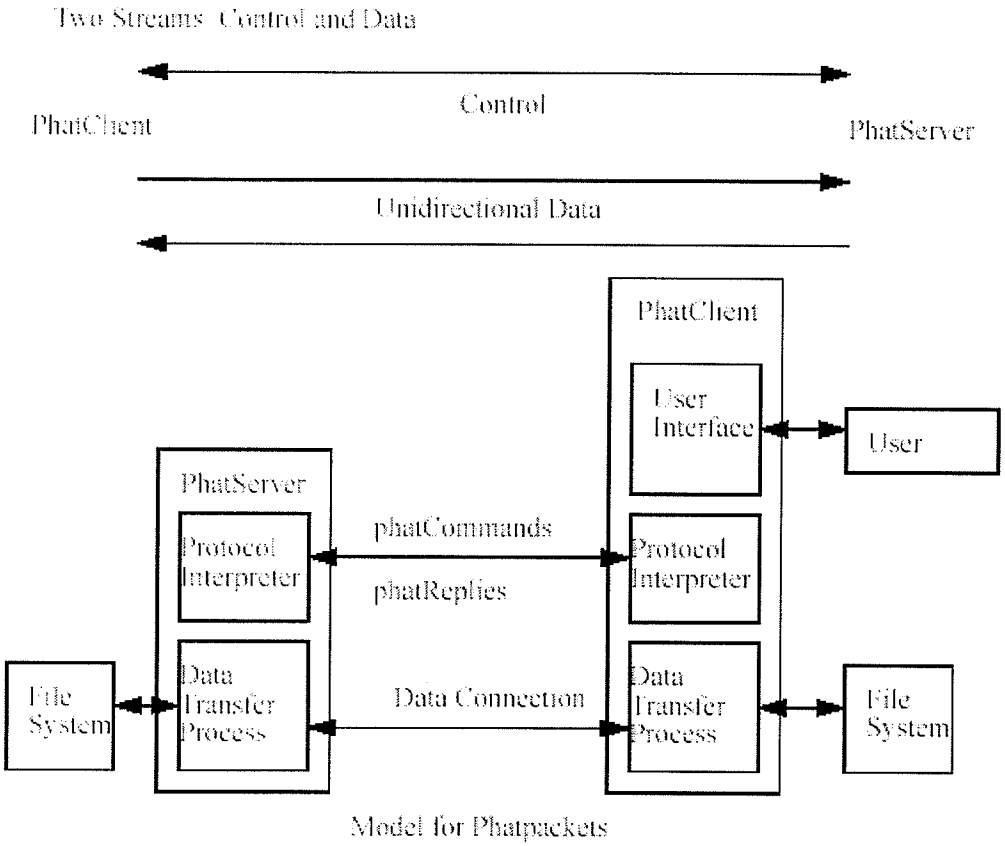


Figure 2. Basic View of the Phatpackets Protocol

Figure 3 illustrates another aspect of the protocol. The basic architecture should be flexible enough for two channels, one control and the other data. The data channel should be as streamlined as possible. The underlying mode of operation is to send data in as continuous a stream as possible recovering from any dropped packets after sending the entire file or a

considerably large portion of it. To be as flexible as possible the overhead data in the UDP packet should be minimal. To date, the minimal overhead has been to associate a stream identifier, segment number (the actual number of the segment as opposed to the byte) with the actual payload. The actual payload should be as large as possible without requiring fragmentation and reassembly at the IP layer at end points (VPN gateways) or at any intervening networks. In the majority of cases this is a limit imposed by the maximum Ethernet frame size. In order to improve efficiencies across networks where packets are encapsulated such as for transmission of VPNs the Phatpackets data payload is reduced to prevent fragmentation. At this time the size of the data payload is set as an option for the file transfer process with 1472 as the default.



Model for Phatpackets

Figure 3. Basic View of the Complete Architecture

Figure 4 illustrates the protocol as it relates to the structure of sending packets and the associated flow control mechanism. The block size is the number of packets streamed out of the server at one time. Different batches are interspersed by the block delay. An analysis window is also illustrated providing a basis for analyzing the packet reception rate thereby providing a feedback mechanism from client to server.

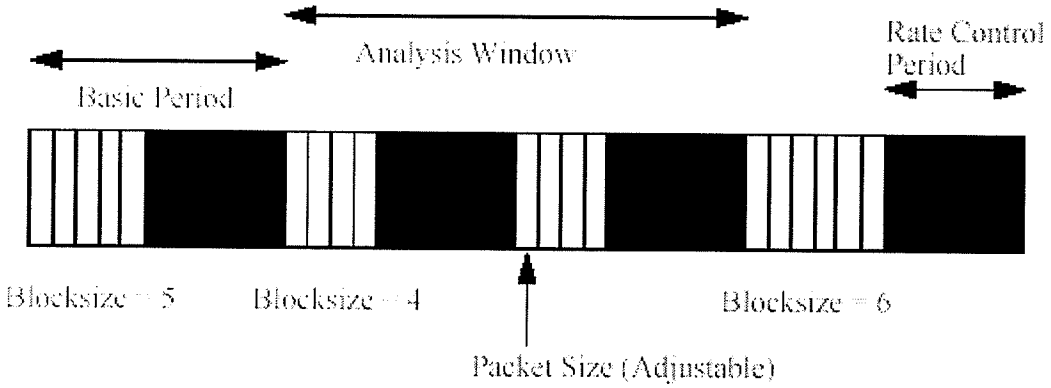


Figure 4. The Block Size Illustration

The basic setting of the block delay is accomplished by sending a stream of packets, estimating a delay that would be required to sustain a connection at an α% packet reception rate. Equation 1 illustrates a simplified rate control equation currently implemented. The alpha value is also smoothed (exponentially averaged) using Equation 2.

$$Delay_{xfer(i+1)} = Delay_{xfer(i)} \times \left(\frac{packets\,sent}{packets\,received}\right) \times \alpha \qquad (1)$$

$$\alpha_{xfer(i+1)} = \alpha_{xfer(i)} \times \beta + NewReceptionRate \times (1-\beta) \qquad (2)$$

In this manner the delay is adapted on the fly. The same basic equation is used recalculating the delay after x packets are sent or received, where x is also an adjustable parameter known as the analysis window size. This simple control structure has the following desirable property: It attempts to maximize flow through the channel if alpha is less than 1.

3.2 The Phatpackets Protocol Algorithm

3.2.1 Flow and Congestion Control Algorithm

In environments with lower levels of statistical multiplexing or with per-flow scheduling, the delay and loss rate experienced by a flow is in part a function of the flow's own sending rate. Thus, a flow can use end-to-end flow and congestion control to limit the delay or loss experienced by its own packets [Flo00].

The flow control and congestion control mechanism should be able to adapt to all kinds of network situations and establish a steady transfer state as fast as possible in order to maximize throughput and minimize packet loss. It should also be fine-grained enough for efficient transmission while not introducing feedback implosion problem that affects scalability. An optimized flow control and congestion control algorithm has been incorporated in the Phatpackets protocol.

The most relevant parameters to the flow control and congestion control mechanism of the Phatpackets protocol for a single "connection" are the block size, block delay, packet reception rate, alpha, ssthresh, and delay variance. The alpha and ssthresh are preset values and are

adjusted subsequently as a function of the packet reception rate. The delay variance is a variable to aggregate the delay variations during the transfer. Imminent packet loss can be predicted by long delay, and network variation can be observed by large delay variance.

The packet losses should be kept to a minimum because not only must dropped packets be retransmitted but also packet losses degrade the overall network throughput. The reason for packet loss may be network congestion as well as client side overflow. To minimize the effect of packet losses, it is necessary to identify the reason and process with different flow control techniques. The flow control and congestion control mechanism of the Phatpackets protocol is used to avoid congesting the network. To avoid overflowing the client's buffer, a customized buffer can be used if the status of the network I/O buffer cannot be acquired. The amount of data in the buffer is used as feedback information to adapt the parameters of the sender in order not to exceed the maximal output rate allowed. The Phatpackets protocol changes the buffer size dynamically, transparently, and automatically to use system memory more effectively and prevent excessive memory consumption.

The Phatpackets protocol has two modes for file transfer; one is Adjust Block Size mode and the other Adjust Block Delay mode. In the Adjust Block Size mode, the block size is adjusted to control the flow rate of the sender. In the Adjust Block Delay mode, the block delay of the sender is adjusted. The Phatpackets algorithm uses the packet reception rate and the delay variance to adapt to network conditions dynamically. An attempt is made to use these variables to probe for additional network capability to get the maximum throughput with a minimum loss rate. In addition, all control parameters are adjusted dynamically to achieve effective rate

control. If the reception rate is higher than alpha, and the block size is lower than ssthresh and the packet reception rate is higher than the previous value, then the algorithm increases the block size exponentially. If block size is lower than ssthresh but the packet reception rate begins to drop below the previous value, it decreases the block size linearly. If the block size is higher than ssthresh and the packet reception rate is higher than the previous value, the algorithm increases the block size linearly. If the block size is higher than ssthresh but the packet reception rate drops below the previous value, the algorithm decreases the block size linearly. Or else, the algorithm decreases the block size exponentially.

The pseudo code for the simplified control algorithm is as follows:

```
oldRate = previous packet reception rate
newRate = current packet received rate
IF newRate > alpha
 IF block size < ssthresh
  IF newRate > oldRate
   Increase the block size exponentially
  ELSE
   Decrease the block size linearly
  END IF
 ELSE
  IF newRate > oldRate
   Increase the block size linearly
  ELSE
```

```
    Decrease the block size linearly
  END IF
 END IF
ELSE
 Decrease the block size exponentially
END IF
Adjust the alpha, ssthresh, block delay and delay variance
```

3.2.2 Integrated Layer Processing Approach (ILP)

As networks proceed to higher speeds, the performance bottleneck is shifting from the bandwidth of the transmission media to the processing time necessary to execute higher layer protocols. Efficient transmission can only be achieved if the unit of control is exactly equal to the unit of transmission, the unit of transmission is the unit of processing, and the unit of processing is also the unit of control [DD97].

Protocol processing can be divided into two parts, control functions and data manipulation functions. In the control part there are functions for control message processing. It has been demonstrated that the control part processing can match gigabit network performance for the most common size of PDUs (Protocol Data Unit) with appropriate implementations [CJR89]. When the control channel is implemented over TCP, the control message is guaranteed to be delivered. If the control channel is implemented over UDP, the application may need to provide a mechanism to keep important control messages from being lost. One such method is implemented in NETBLT [CLZ87]. It reduces control message loss by using a single long-

lived control packet; the packet is treated like a FIFO queue, with new control messages added to the end and acknowledged control messages removed from the front. The implementation places control messages in the control packet and transmits the entire control packet, consisting of any unacknowledged control messages plus new messages just added. The entire control packet is also transmitted whenever the control timer expires. Since control packet transmissions are fairly frequent, unacknowledged messages may be transmitted several times before they are finally acknowledged. This redundant transmission of control messages provides automatic recovery for most control message losses over a noisy channel.

Examples of data manipulation functions are checksumming, compression, encryption and decryption. Data manipulation functions present a bottleneck. They consist three phases. First a read phase where data is loaded from physical disks to memory then to cache or registers, then a manipulation or processing phase, followed by a write phase. For very simple functions, e.g. checksumming or byte swapping, the time to read and write to memory dominates the processing time. For other processing oriented functions, like encryption and some presentation encodings, the manipulation time dominates the processing time. However, the situation is expected to change with the increase of processor performance. The memory access will be the major bottleneck rather than data processing.

The increase of processor speeds and the discrepancy between processor and memory performance is pushing toward the integration of data manipulation operations. The main concept behind ILP is to minimize costly memory read/write operations by combining data

manipulation oriented functions instead of performing them serially as is most often done today [DD97].

The Phatpackets protocol integrates the layer 4 to layer 7 of the OSI Reference Model into one layer, thus reducing the overhead for passing data between protocol stacks and costly memory operations. As regards disk I/O operations, the Phatpackets algorithm avoids accessing the disk unnecessarily and avoids processing bytes and characters individually.

3.2.3 Adaptability

As the technology progresses, the networking conditions are going to become more and more variable. It should be quite obvious that applications that can gracefully adapt to multiple environments will outlive those that will have to die when their requirements are not met. It should also be quite obvious that an application that can automatically characterize its environment will be more robust and easier to deploy than an application that has to exchange signaling information with the network for that purpose. Adaptability guarantees that the application will always use all the available resources and make the most efficient usage of these resources [DD97]. By combining network status probing with server/client host load adaptation, and load balancing techniques, the Phatpackets protocol attempts to maximize throughput and minimize packet loss across all connections for all applications within the resource limits of the hosts.

3.2.4 Adaptive Network Load Balancing Algorithm (ANLB)

In a heterogeneous network, hosts differ in processing power, memory, and swap space availability. The load on these hosts also varies over time. The network latency, which is determined by the total network traffic and not by the load on an individual host also changes quickly. Thus a good network load balancing algorithm should have at least two properties: Sensitivity to network latency and Anticipatory load distribution [Mon00].

The Phatpackets protocol utilizes multiple connections from multiple servers simultaneously to enhance the performance, scalability and availability of mission-critical transport services, such as Web, Terminal Services, Virtual Private Networking, and streaming media servers [HM202]. As such load balancing is essential to the Phatpackets protocol.

The Phatpackets protocol has an Adaptive Network Load Balancing algorithm (ANLB) to maximize the overall throughput for operation in a network of heterogeneous client and server hosts. ANLB utilizes an approach which is a combination of dispatcher-based, client-based and server-based approaches in sever selection, and adopts predictive load balancing and subsequent load rebalancing in achieving effective load balancing. As such the ANLB within the Phatpackets protocol has a full distributed and autonomous architecture.

Factors such as current Internet traffic conditions, loads and locations of PhatServers, and locations of PhatClients are critical to server selection. This information should be updated regularly and maintained in a directory service. The PhatServers exchange load and availability information with other PhatServers. In partial data replication schemes, the file information is also exchanged. To avoid feedback implosion problems that affect scalability, PhatServers can

be organized into clusters and one cluster elects a PhatServer to delegate the requests and responses for all its members with other clusters. Since there is always a tradeoff between the overhead due to collection of system state information and performance gain by use of available state information, the Phatpackets protocol tries to strike a balance.

When a PhatClient sends a file transfer request to a PhatServer, the PhatServer replies with the IP address/file information of PhatServers that contain data replicas. The server selection is based on client information, request frequency, and server location, load and availability. By using the highest 8 bits of the IP address only, the geographical region of the client can be approximated [HKC96]. For requests from the same PhatClient, the server selection can be based on a round robin scheduler. For requests from other clients, a random selection can be made from the candidate host set based on server locations, relative loads and available resources. While making server selections, a heuristic approach is used to ensure that no server is overloaded.

The PhatClient then tries to establish control connections with these servers and measures the BDP with each server. The PhatClient uses the BDP approximation to decide the number of packets to be delivered from each server.

3.2.4.1 Appropriate Initial Status Setup with BDP estimation

Prior to the initiation of a file transfer it is important to estimate a number of parameters associated with the transfer. At the beginning of a transmission through a network with unknown conditions the client/server hosts need to probe the network to determine the

available capability in order to avoid congesting the network with an inappropriate initial large burst of data. A small closely spaced series of data packets that are sent to probe the network arrive at the receiver at the rate that approximates the bottleneck link bandwidth [Hoe96].

BDP estimation depends on two major factors, namely network conditions and server load/performance. If there is a capable server system present, but the connectivity of the client in terms of delay and available bandwidth is not good, a large BDP will be manifested. If the server system is saturated with requests or has a poor performance in regards to CPU, RAM, disk and network I/O resources, the BDP will also tend to be large [Aga01]. Although the Phatpackets protocol is inherently rate based and less susceptible to limitations caused by high BDP within protocols such as TCP it is still a good function for initial load balance estimation.

3.2.4.2 Load Rebalancing

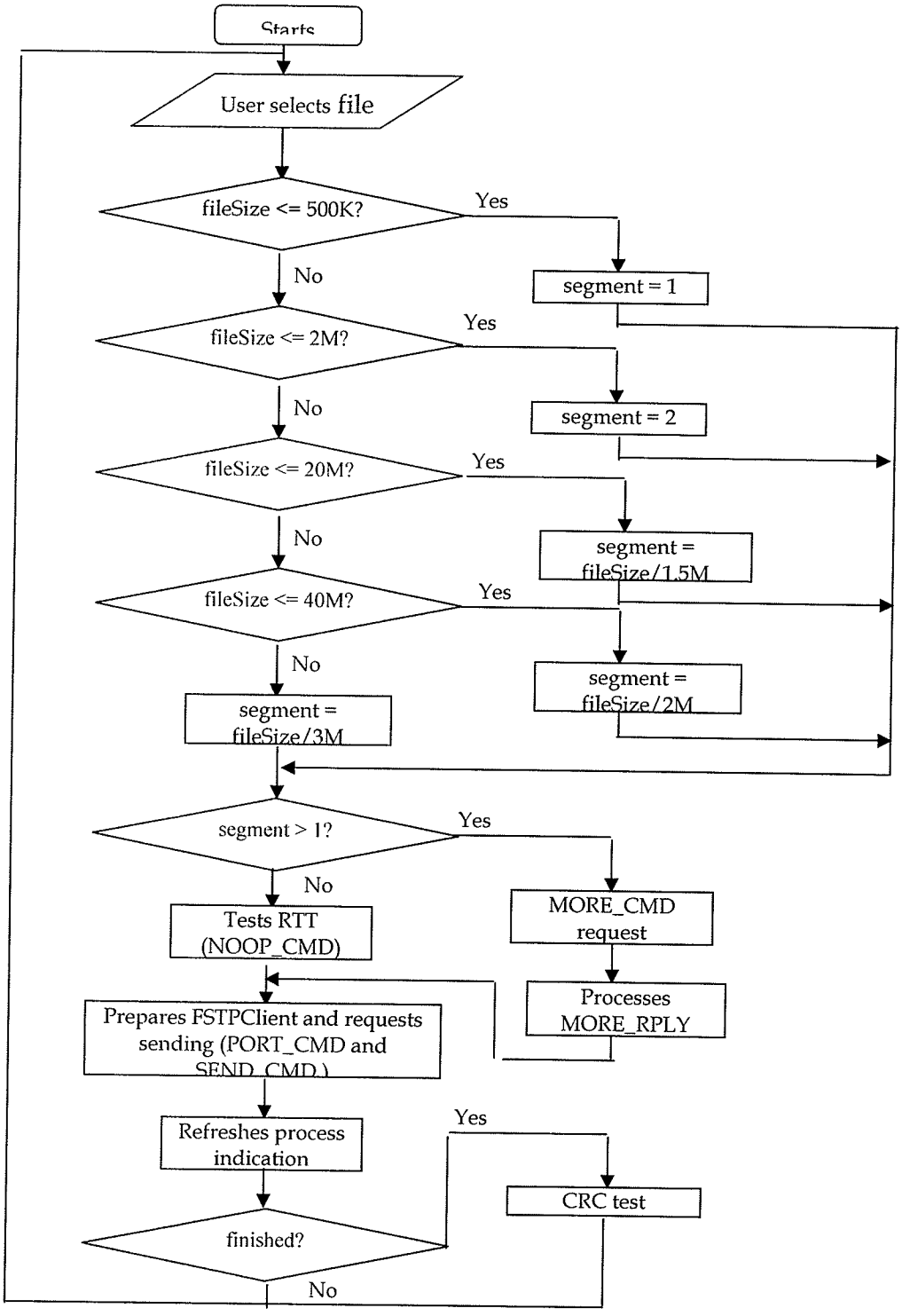
Since the performance of the transfer is heavily affected by the server and client performance, for example, the averaged throughput for a slow server machine may be only 30% - 40% of that of a fast server machine collocated at the same site, and the network traffic and server load vary over time, load rebalancing during the transfer process is essential. The Phatpackets protocol utilizes a load rebalancing algorithm based on a round robin scheduler.

The load among the different connections from same server tends to be balanced and normally does not need to be rebalanced since these connections share the same path and experience same network situations in general. The load among different servers however needs to be rebalanced on-the-fly during the transfer process.

The load rebalancing algorithm determines the reusable priority based on the finished task size. The data downloading connection that finished the largest task size is given the highest priority. Then the connections that finish their task parts are queued for reuse purposes. If there are some connections that haven't downloaded a reasonable part, the connections in the queue can take over part or all their task parts. The slowest connection is given the highest priority for rebalancing purpose. The switch should be as smooth as possible. Test results for predictive load balancing approaches such as BDP estimation without and with later round robin rebalancing are discussed in Chapter 5.

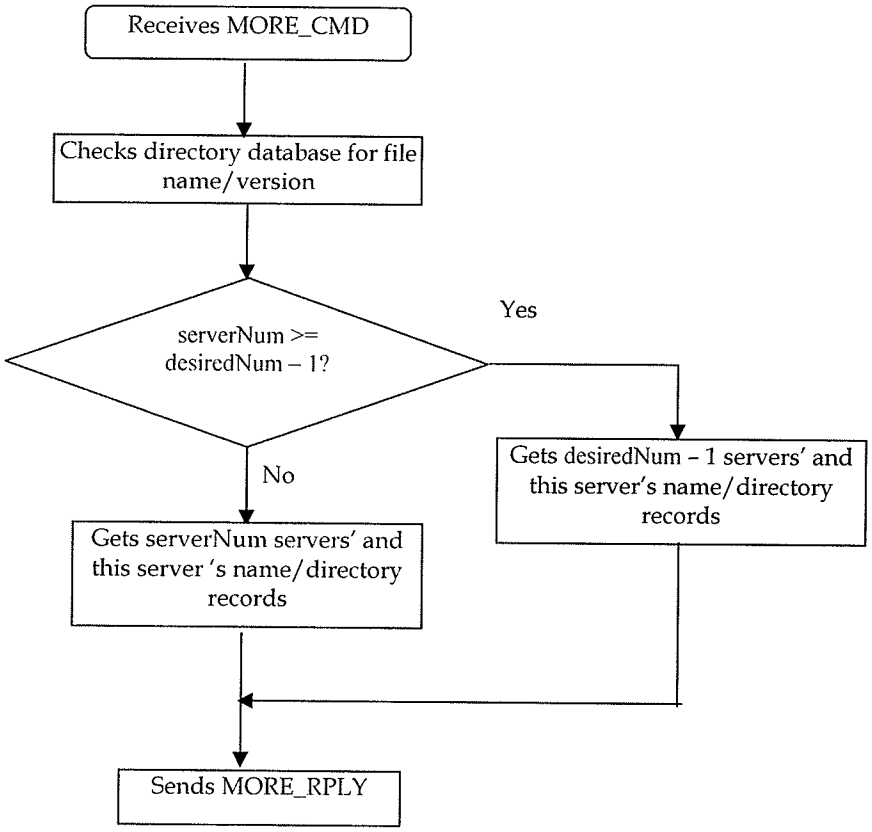
3.3 Flow Charts of the Phatpackets Protocol

3.3.1 PhatClient of the Phatpackets Protocol



3.3.2 Process MORE_CMD command of the Phatpackets Protocol

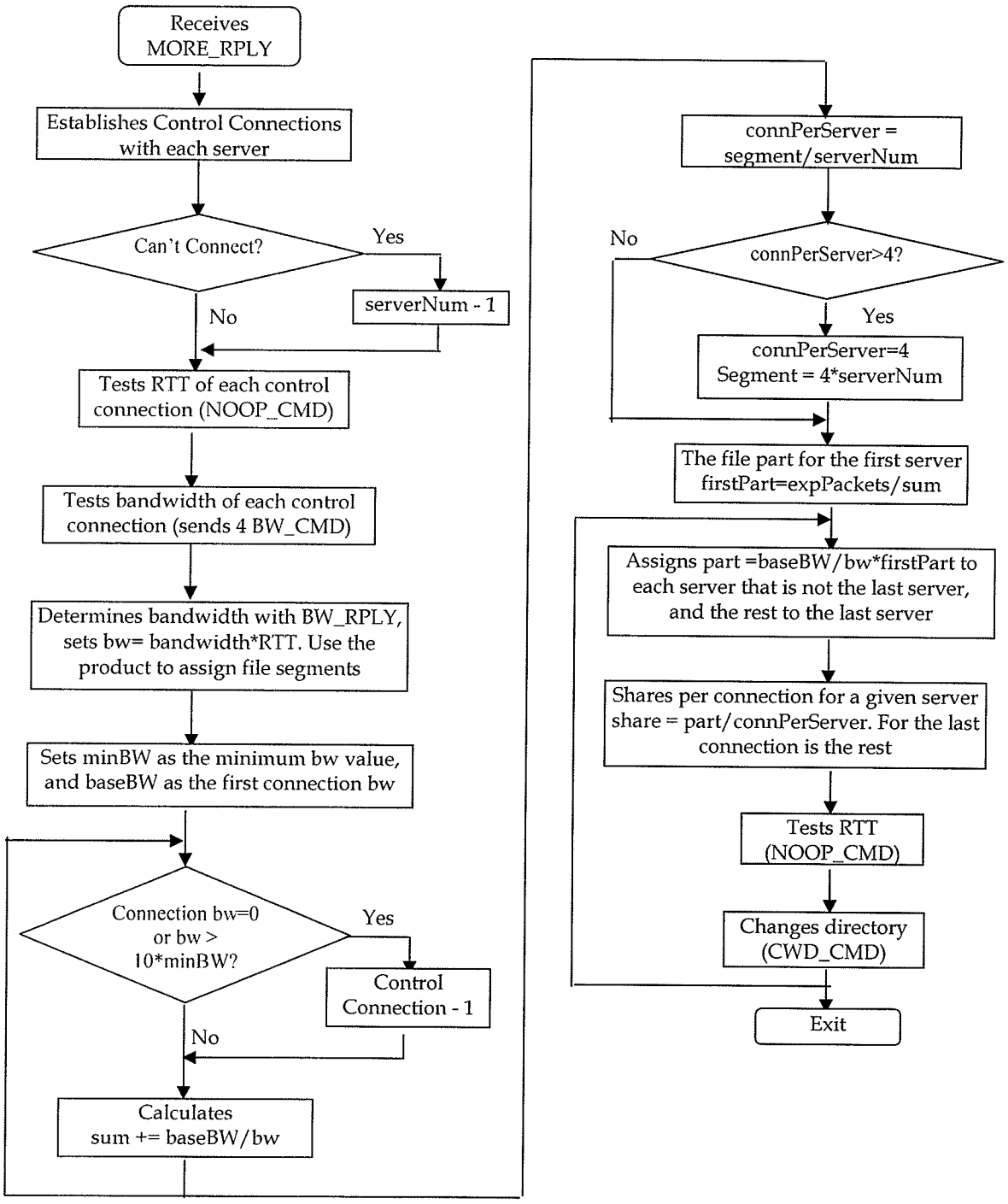
The Phatpackets Protocol control commands are listed in Appendix A.



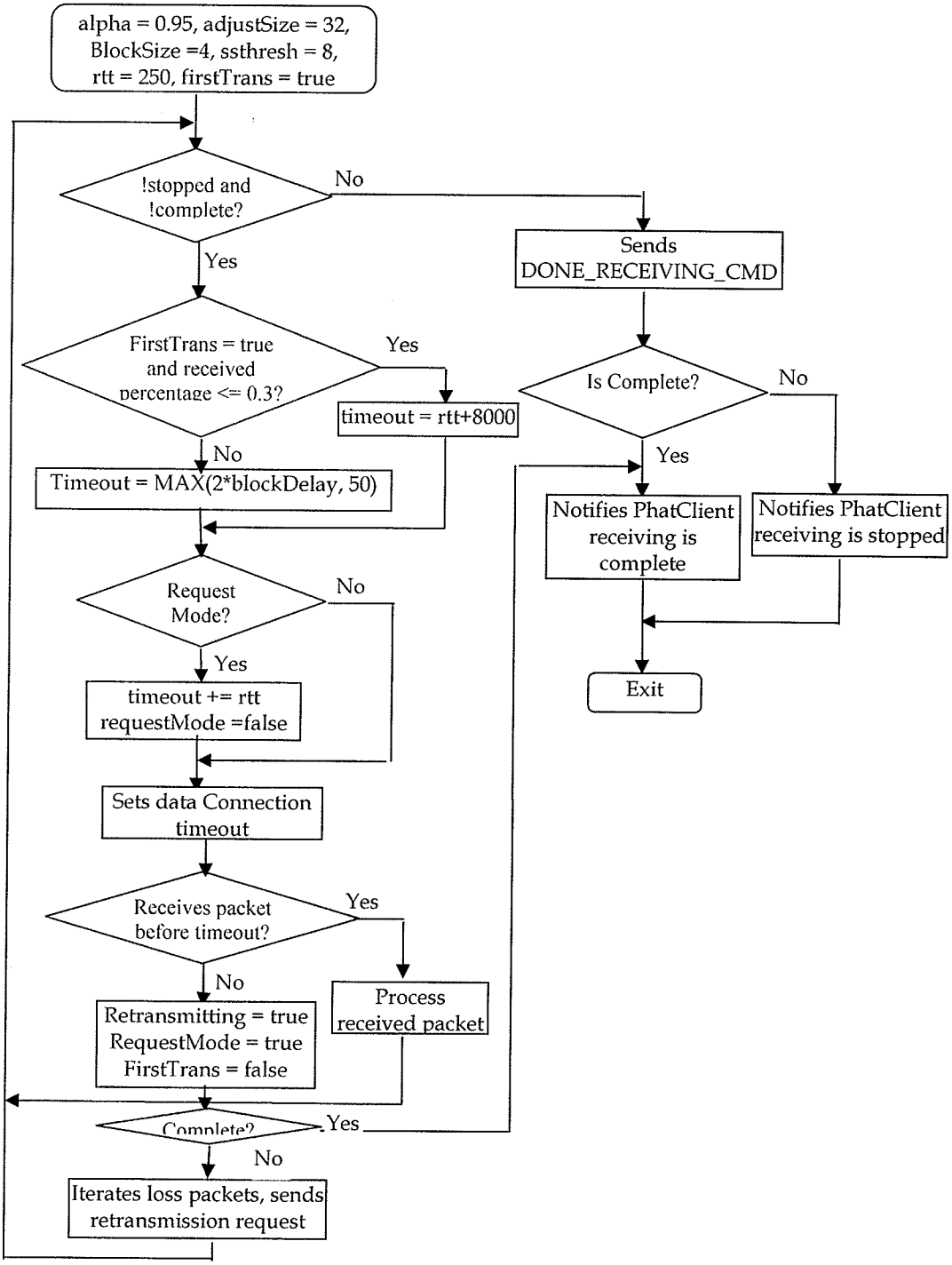
3.3.3 Process MORE_RPLY command of the Phatpackets Protocol

The basic work principle is as follows:

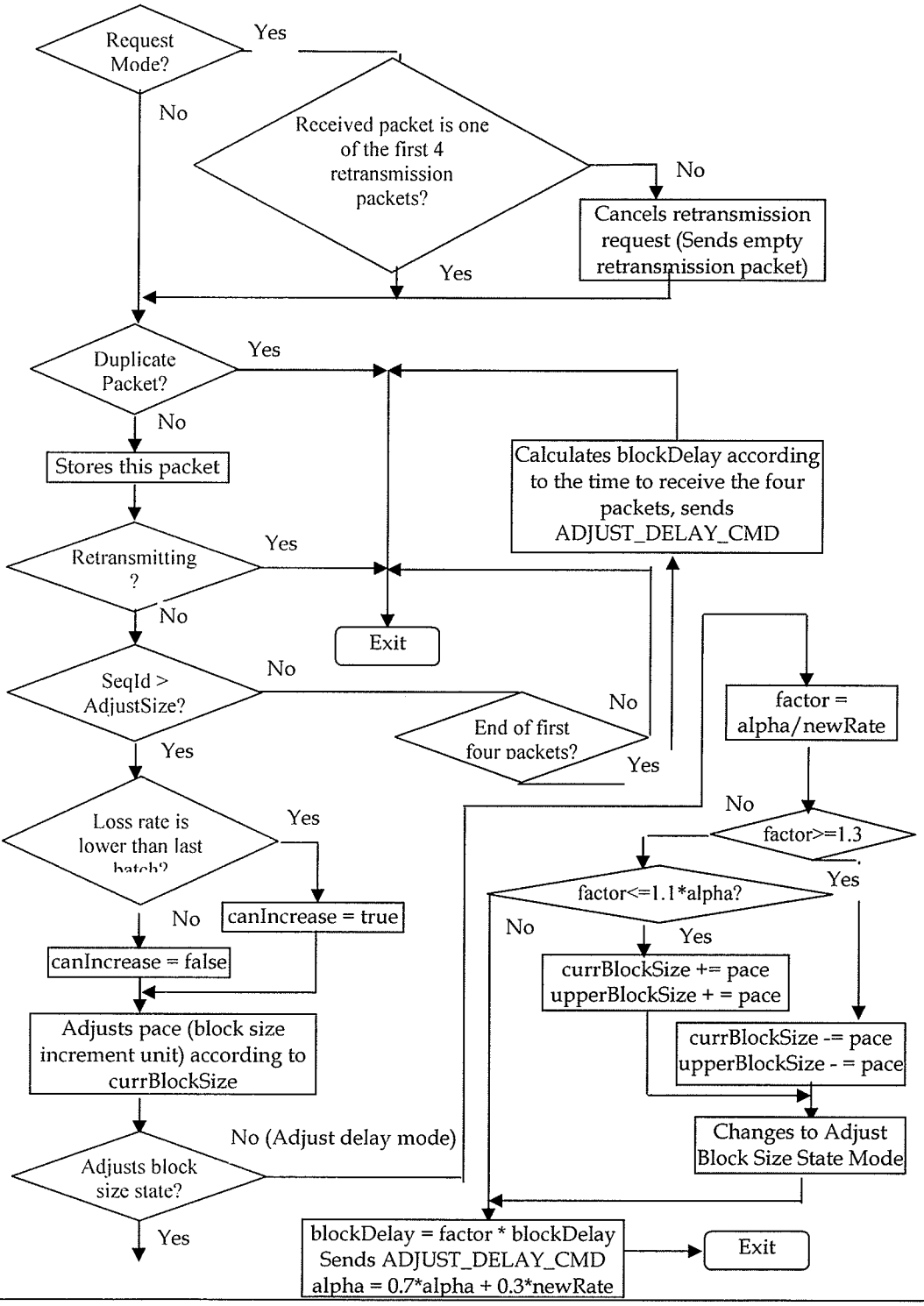
If a server is down or there is connection error with a server, or if the BDP value with a server is larger than ten times the minimum BDP value, this server is ignored. And at most four connections with a given server are used.

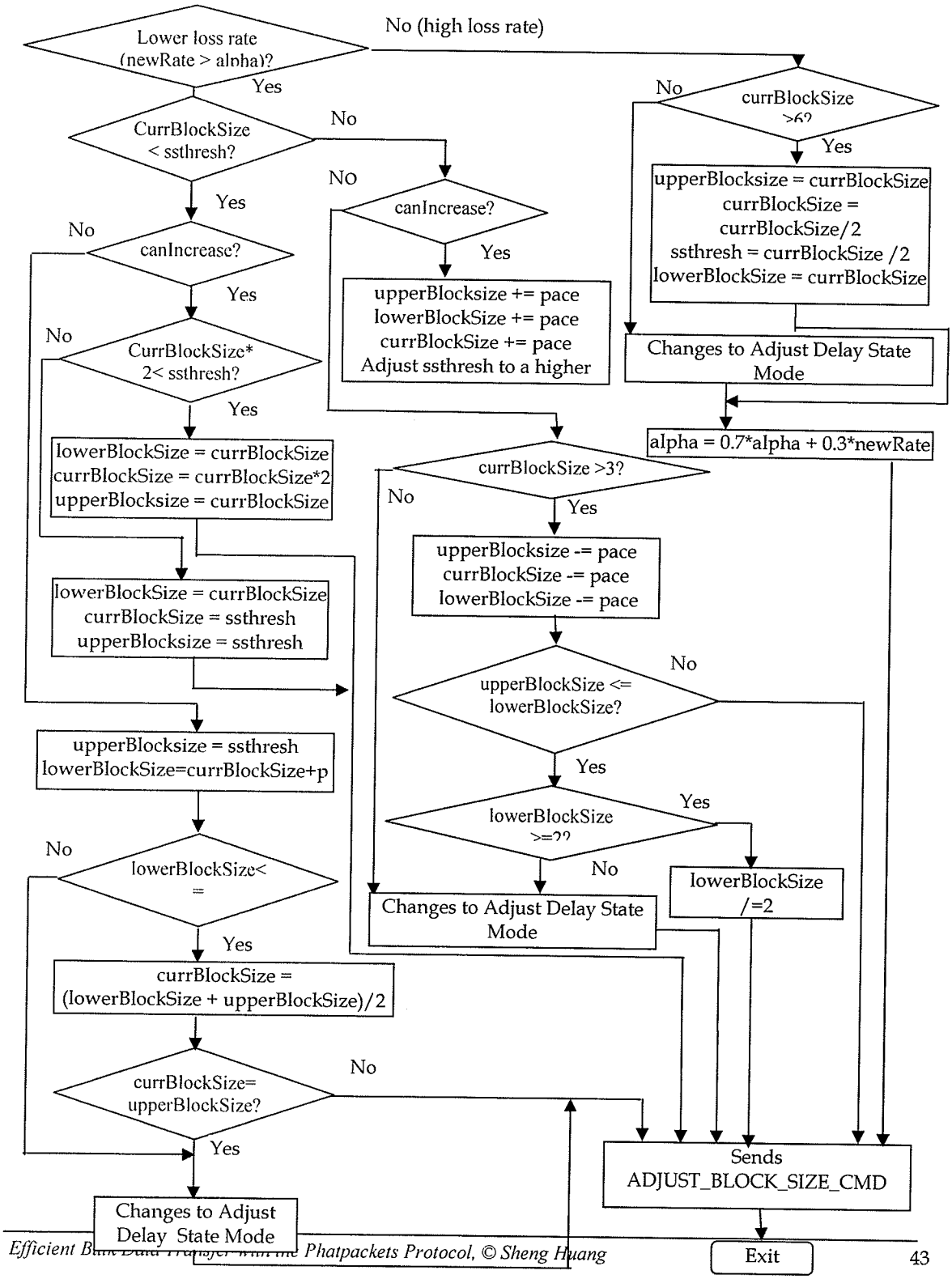

Receives MORE_RPLY
Establishes Control Connections with each server
Can't Connect?
Yes
serverNum - 1
No
Tests RTT of each control connection (NOOP_CMD)
Tests bandwidth of each control connection (sends 4 BW_CMD)
Determines bandwidth with BW_RPLY, sets bw= bandwidth*RTT. Use the product to assign file segments
Sets minBW as the minimum bw value, and baseBW as the first connection bw
Connection bw=0 or bw > 10*minBW?
Yes
Control Connection - 1
No
Calculates sum += baseBW/bw
connPerServer = segment/serverNum
No
connPerServer>4?
Yes
connPerServer=4 Segment = 4*serverNum
The file part for the first server firstPart=expPackets/sum
Assigns part =baseBW/bw*firstPart to each server that is not the last server, and the rest to the last server
Shares per connection for a given server share = part/connPerServer. For the last connection is the rest
Tests RTT (NOOP_CMD)
Changes directory (CWD_CMD)
Exit

3.3.4 FSTPClient of the Phatpackets Protocol



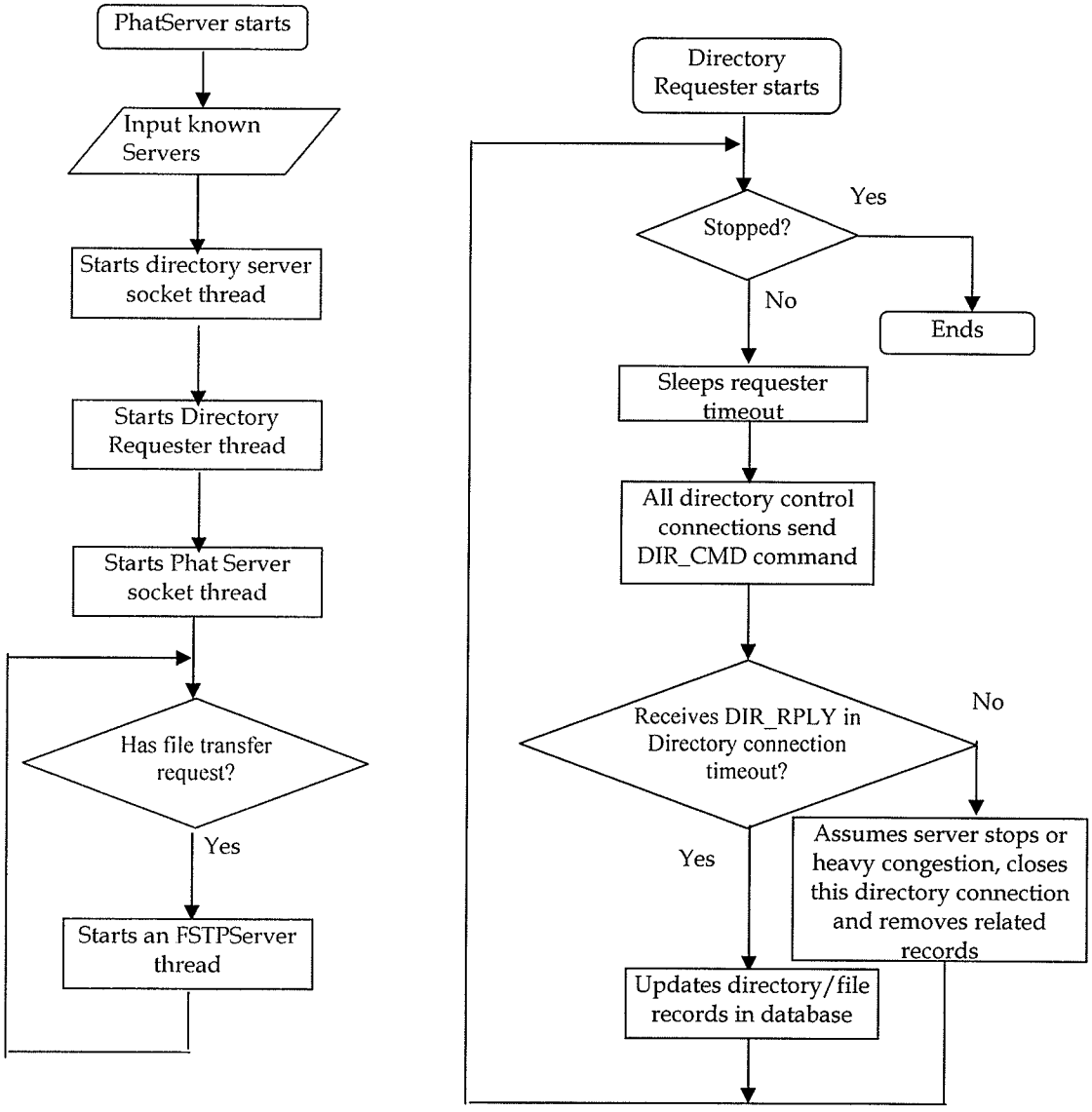
3.3.5 Process Received Packet



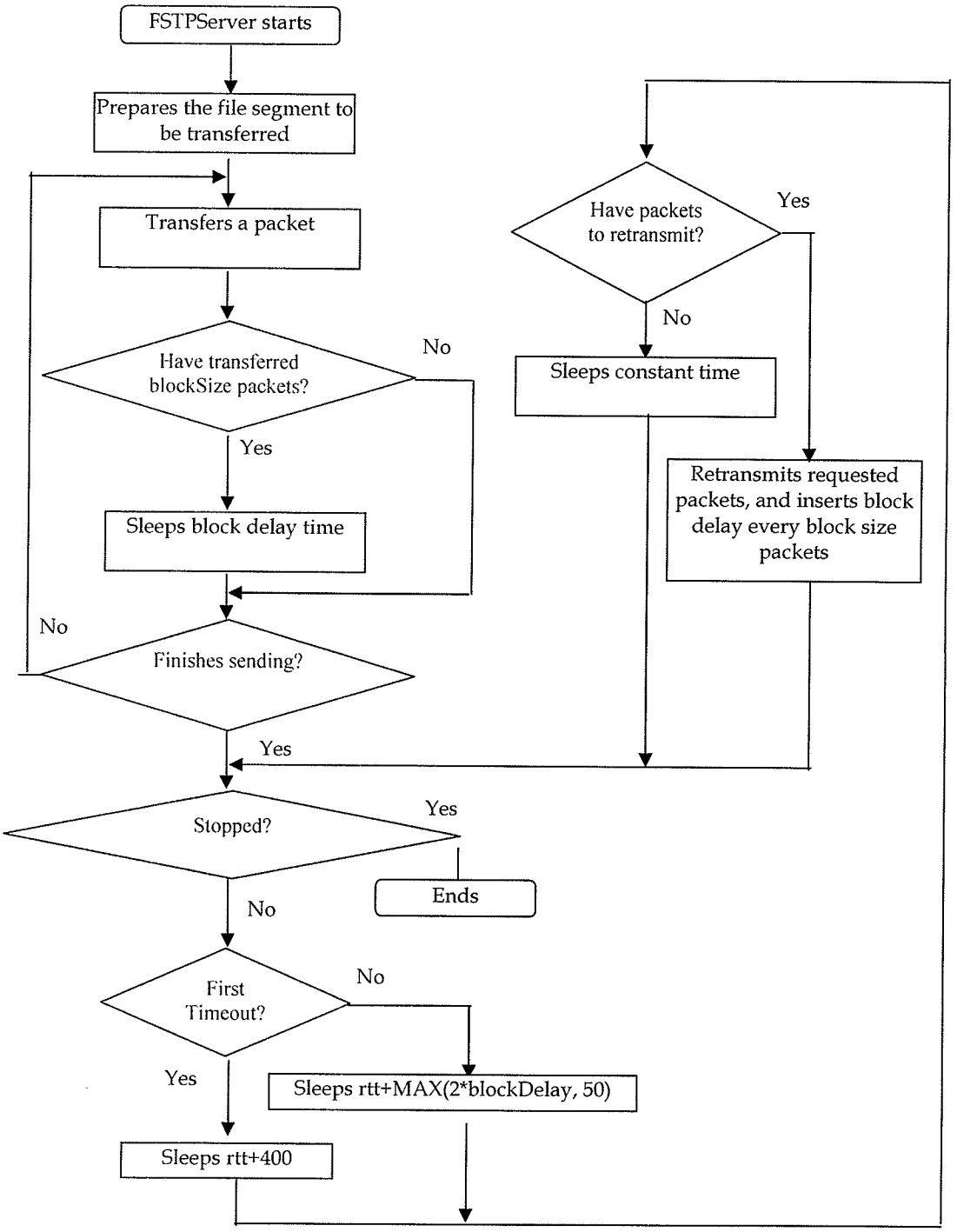

Lower loss rate (newRate > alpha)?
No (high loss rate)
Yes
CurrBlockSize < ssthresh?
No
Yes
canIncrease?
No
Yes
CurrBlockSize* 2< ssthresh?
No
Yes
lowerBlockSize = currBlockSize
currBlockSize = currBlockSize*2
upperBlocksize = currBlockSize
lowerBlockSize = currBlockSize
currBlockSize = ssthresh
upperBlocksize = ssthresh
upperBlocksize = ssthresh
lowerBlockSize=currBlockSize+p
lowerBlockSize<=
No
Yes
currBlockSize = (lowerBlockSize + upperBlockSize)/2
currBlockSize= upperBlockSize?
No
Yes
Changes to Adjust Delay State Mode
NO
canIncrease?
Yes
upperBlocksize += pace
lowerBlockSize += pace
currBlockSize += pace
Adjust ssthresh to a higher
currBlockSize >3?
No
Yes
upperBlocksize -= pace
currBlockSize -= pace
lowerBlockSize -= pace
upperBlockSize <= lowerBlockSize?
No
Yes
lowerBlockSize >=??
Yes
No
Changes to Adjust Delay State Mode
lowerBlockSize /=2
No
currBlockSize >6?
Yes
upperBlocksize = currBlockSize
currBlockSize = currBlockSize/2
ssthresh = currBlockSize /2
lowerBlockSize = currBlockSize
Changes to Adjust Delay State Mode
alpha = 0.7*alpha + 0.3*newRate
Sends ADJUST_BLOCK_SIZE_CMD
Exit

3.3.6 PhatServer of the Phatpackets Protocol

When a PhatServer is started, the Internet address of other started PhatServers can be inputted so that this server can join the server cluster. Directory information is sorted using Merge Sort for the purpose of faster directory requests and processing using a binary search.



3.3.7 FSTPServer of the Phatpackets Protocol



Summary

This chapter presented an architecture for improving the waiting time to download files from the Internet and a set of methods. The basic model effectively represents an alternative transport layer in the TCP/IP protocol stack. The Phatpackets protocol attempts to combine the best aspects of a variety of transport and control methods with the objective of developing a near optimal and robust protocol. The flow control and congestion control mechanism for a single "connection" uses parameters such as block size, block delay, packet reception rate, alpha, ssthresh, and delay variance to realize efficient rate control. The Phatpackets protocol utilizes multiple connections from multiple servers simultaneously to enhance the performance, scalability and availability. The Phatpackets protocol uses a predictive load balancing approach namely BDP estimation at the beginning of transfer for initial load balancing and it uses a load rebalancing approach based on round robin algorithm during the transfer to dynamically adjust the load on all connections from all servers.

Chapter 4

THE PHATPACKETS PROTOCOL IMPLEMENTATION

4.1 Introduction to the Implementation Tool - Java

Java was introduced by Sun Microsystems in 1995 as a cross-platform environment for building and deploying business applications. Java started as a client-side player and rapidly moved to the enterprise server. In 2000, Java moved back to "modern clients" — digital assistants, cell phones, and global positioning systems in cars, etc.. During the late 1990s, as Java for the enterprise matured, culminating with the release of Java 2 Platform Enterprise Edition (J2EE) in 1999, the power of Java was able to bring together disparate operating environments in the server space. The concern about using Java as a server-side language because of performance issues has really gone away since the last several releases of the JDK have shown dramatic improvement in performance with a lot of the just-in-time compilation technology. As Java's ties to XML are strengthened, more Java based Web services will be seen on the Internet.

Java is selected as the development tool for several reasons. First, Java is platform independent. Java has the virtue of "compile once, run everywhere". Second, the applications are more neatly and cleanly written in Java than in other languages. Third, client/server programming in Java is becoming increasingly popular, and may even become the norm in the upcoming years.

Its networking related packages provide powerful APIs for implementations. It has exception mechanisms for robust handling of common problems that occur during I/O and networking operations. And its threading facilities provide a way to easily implement powerful client and server programs.

4.1.1 Introduction to Java

The Java platform has two components: the Java Virtual Machine (Java VM) and the Java Application Programming Interface (Java API). The Java VM is the base for the Java platform and is ported onto various hardware-based platforms. It mediates between Java applications and the underlying hardware platform —executing application bytecodes, managing system memory, providing system security, and juggling multiple execution threads. The Java API is a large collection of ready-made software components that provide many useful capabilities, such as graphical user interface (GUI) widgets. The Java API is grouped into libraries of related classes and interfaces; these libraries are known as packages. Java's APIs are among the most well-documented and complete APIs in all programming languages.

The Java™ 2 Platform, Standard Edition 1.4 has introduced several new features and enhancements for Object Serialization, Remote Method Invocation (RMI), Common Object Request Broker Architecture (CORBA), and networking in general [Mah02]. New Features and Enhancements in Networking are:

- ✧ IPv6: Internet Protocol version 6 support for TCP and UDP application (including multicast).

- ✧ HTTP Digest Authentication: The HTTP digest authentication implementation has been updated to support proxies and origin servers.
- ✧ Unconnected/Unbound Socket: This allows more flexible socket creation, binding, and connection and enables manipulation of socket options before establishing or accepting connections. In addition a timeout can be specified when establishing a connection. A new class, javax.net.ssl.SSLSocket, which is a subclass of java.net.Socket has been added to provide security for data sent through sockets through encryption.
- ✧ Connected UDP Socket: The UDP protocol is a connectionless protocol, however, the DatagramSocket connect method now establishes the address association at the native level. Where supported, this allows an application to have visibility of ICMP port unreachable messages as an indication that the remote application is unavailable.
- ✧ Uniform Resource Identifier (URI): The java.net.URI is a new class that allows URI construction and parsing without the presence of a protocol handler, which is not possible with the URL class. JNDI DNS Service Provider in InetAddress: This enhancement in class java.net.InetAddress enables applications to configure a pure Java name service provider by using a DNS name service provider through JNDI.
- ✧ URLEncoder and URLDecoder: These have been added to enable applications to use other character encoding/decoding schemes.
- ✧ TCP Out-of-Band Data: New methods in class java.net.Socket have been added to provide limited support for TCP urgent data for support certain legacy applications. Urgent data may be sent on any TCP socket. However, only partial support for receiving urgent data is provided.

- ✧ SOCKS: Full V5 and V4 TCP support with auto-negotiation with the proxy of which version to use.
- ✧ NetworkInterface: The java.net.NetworkInterface class allows enumeration of interfaces and addresses and can be used in applications deployed on machines with multiple network interfaces.

4.1.2 Java Performance [Mel99]

The Java VM has three generations: Interpreter (JDK 1.0 - 1.1.5), Just-In-Time (JIT) Compiler (JDK 1.1.6+, JDK 1.2) and Dynamic optimizing compiler (e.g., JDK 1.2.2 with Java HotSpot 1.01, JDK1.3). Older interpreter-style VMs are the slowest of all of these. Costs in running Java applications are byte-code execution, memory allocation/garbage collection, thread synchronization, method dispatch, and other factors. JIT compiles byte codes to native instructions on-the-fly so it has faster byte code execution and faster method dispatch. But it may use more RAM than the Interpreters and could hurt performance on low memory machines.

Using the Java 2 platform's new pluggable architecture, Java HotSpot can be seamlessly dropped into the platform, replacing both the classic Interpreter and the JIT compiler. Once this new performance engine is installed, any application or applet processed with the Java 2 runtime environment (application launcher, plug-in, or applet viewer) will by default use the Java HotSpot performance engine.

The Java HotSpot performance engine concentrates on several key areas to achieve its state-of-the-art performance enhancement. These areas include on-the-fly "adaptive" compilation, method inlining, improved and redesigned object layout, fast and fully accurate garbage collection, and ultra-fast thread synchronization. Such enhancements are typically most effective on server side applications. JDK1.3 has versions of HotSpot for client (fast execution on first pass) and server (faster execution on later pass).

Performance of an application written in the Java programming language generally depends upon four factors: the overall design of the application, the speed at which the Java bytecodes are executed, the speed at which the libraries execute (in native code), the speed of the underlying hardware and operating system. The typical client side application's performance (particularly graphics applications) is most heavily impacted by native libraries, whereas the typical server side application stresses the speed of bytecode execution—which is where the new VM shines.

For adaptive compilation, rather than compiling an entire program when it first starts, or compiling each method as it is called (as does the JIT compiler), the performance engine initially runs the program using an interpreter, and then analyzes it as it runs, looking for performance "hot spots." It then compiles and optimizes only those performance-critical areas of code. This monitoring process continues dynamically throughout the life of the program, with the performance engine adapting on-the-fly to the ongoing performance needs of the application.

4.2 UDP Socket Buffer

Since the situation where the sender overwhelms the receiver's buffer should not be interpreted as network congestion and thus curb the sending speed of the server, the receiver's buffer should be reasonably large. On SunOs 5.7, the default TCP send buffer size is 8K, the default TCP receive buffer size is 32K. The default UDP send buffer and receive buffer size are both 8K. On Windows 2000, all the four default sizes are 8K. The Java code snippet below shows how to probe the underlying operating system to get the system allowed, up to 450KB receive buffer size. The send buffer size can be set in a similar way.

```
private void setMaxReceiveBufferSize() throws IllegalArgumentException {
  int bufferSize = 450000;
  int pace = 0;
  boolean done;
  do {
      if (bufferSize <= 75000) {
        pace = 1500;
      }
      else if (bufferSize <= 150000) {
        pace = 7500;
      }
      else if (bufferSize <= 250000) {
        pace = 15000;
      }
      else {
        pace = 37500;
      }
```

```
        try {
            udpSocket.setReceiveBufferSize(bufferSize);
            done = true;
        } catch (SocketException se) {
            bufferSize -= pace;
            done = false;
        }
    } while (!done);
}
```

4.3 The Phatpackets Implementation

4.3.1 Introduction to Core Algorithm Implementation

This section outlines some of the actual implementation of the Phatpackets architecture discussed above. The implementation tool utilized is JDK J2SE 1.4.2. The control connection is implemented using TCP, and the data connection is implemented over UDP. Since the performance of the implementation is heavily affected by network I/O and disk file I/O, these I/O operations are extensively optimized. For networking I/O, some TCP socket options such as Nagling and UDP socket options such as buffer size are modified [Nag84]. For disk file I/O, random access and data buffering are optimized. To speed up I/O, the Phatpackets algorithm tries to avoid excessive method calls, accessing the disk unnecessarily, accessing the underlying operating system unnecessarily, and avoid processing bytes and characters individually [Mcc02][Zuk01].

Nagle's algorithm is enabled by default on all known OS implementations [Nag84]. It instructs the sender to buffer data if any unacknowledged data is outstanding. Any data sent subsequently is held until the outstanding data is acknowledged or until there is a full packet to send. This algorithm was developed in 1984 for heavily loaded networks such as Ford Aerospace and Communications Corporation's at that time. It is undesirable in highly interactive environments, such as this client/server application. As regards the TCP control connection, the Phatpackets algorithm disables Nagle's algorithm through the use of TCP_NODELAY sockets option.

Since the situation that the sender overwhelms the receiver's buffer should not be interpreted as network congestion, the receiver's buffer should be reasonably large however the default UDP send and receive buffer sizes are no larger than 32K in all known operating systems. Otherwise the effect is that the client implicitly curbs the flow rate of the server. The socket buffer size should be one to two times the product of bandwidth and round trip delay. Automatic buffer tuning algorithm has been proposed to use system memory more effectively and prevent memory starvation [Web02][SMM98]. For the implementation discussed here, the buffers are set to a maximum of 450KB or the maximum value allowed by the underlying OS using the simple algorithm discussed in 4.2.

At this time, the Phatpackets protocol uses a CRC (Cyclic Redundancy Check) to guarantee the file received is integrity protected. The CRC also acts as a signature to ensure file version integrity in a distributed server environment.

When a PhatServer is started, the Internet addresses or names of other running PhatServers can be declared. These PhatServers then exchange directory information and store the directory information of other servers in their local databases. The directory information has fields of file name, file directory, file length and CRC.

The Phatpackets protocol uses BDP estimation at the beginning of transfer for initial load balancing. The PhatClient calculates the number of servers it wants according to a file size heuristic function. When a PhatServer receives a file transfer request with a desired number of servers larger than one, it will check its local database for this specific file. The reply message contains the PhatServer name/directory pairs. The PhatClient then tries to establish control connections with these servers and measures the BDP by calculating the time to receive the first four full size data packets. Note the link bandwidth of the server to client is utilized instead of the link bandwidth from client to server since the bandwidth may be asymmetric. The full size data packets represent a real approximation of the actual file transfer. This approach is more appropriate than ICMP based Automatic Bandwidth Delay Product Discovery, which requires the cooperation of at least one router vendor [Web02].

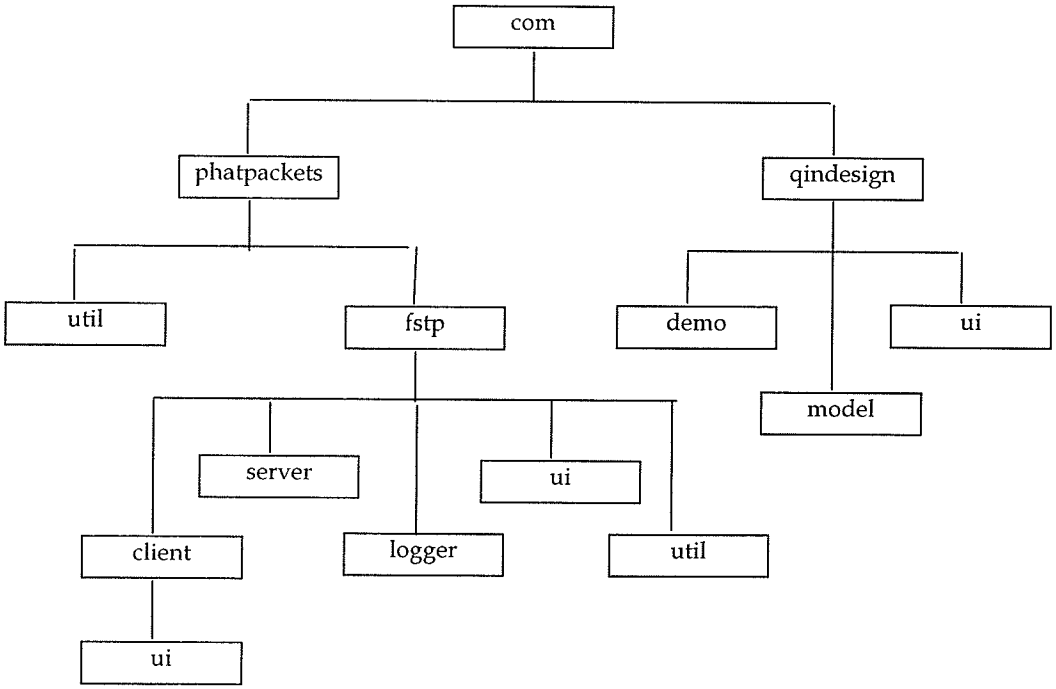
The PhatClient uses the BDP approximation to decide the number of packets to be delivered from each server. Then it sets up multiple connections with each server. Each connection is implemented with one independent Java thread. The current implementation ignores connections whose speed is slower than 1/10th of that of the fastest connection. Based on experimental results, the maximum connection number per server is set to four.

The Phatpackets protocol uses a load rebalancing approach based on round robin algorithm during the transfer. The load among the different connections from the same server tends to be balanced and normally does not need to be rebalanced since these connections share the same path and experience same network situations in general. However the load among different servers needs to be rebalanced on-the-fly during the transfer process. In the current implementation, a client controller monitors the transfer process of all downloading threads and carries out load rebalancing with round robin scheduling.

4.3.2 Main Classes and Packages

This section describes main classes in the Phatpackets protocol implementation [Sil00].

4.3.2.1 Package Structure



4.3.2.2 Main Classes in the Phatpackets Implementation

a) phatpackets.fstp

AbstractHandler --- an abstract class that implements TCP control connection message transfer

CommandConstants --- an interface that contains a collection of control command constants

DataConsumer --- the interface that declares methods for data consumers (queue or disk write)

DataQueue --- the class that uses queue implementation of DataConsumer

DataSource --- the interface that declares methods for data provider

DataStorage --- the interface that declares methods for storing packet data into storage (such as packet data size, total size, and push method)

FSTPClient --- the client class that handles one requested file or file segment

FSTPException --- the top level class of the Phatpackets protocol specific exceptions

FSTPPeer --- the abstract superclass of FSTPClient and FSTPServer that implements a Datagram Socket and its relevant operations (packet size, stream id, port setting, connection and close)

FSTPServer --- the server class that handles one requested file or file segment

MalformedPacketException --- the subclass of FSTPException indicating UDP data packet parsing error

PacketSizeException --- the subclass of FSTPException indicating unexpected packet size error

PacketTracker --- the interface that defines methods for tracking received packets

PhatPacket --- the class that wraps UDP Datagram Packet, providing methods for inserting stream id (unique to every file request), sequence id (unique to every packet in a file request), calculating header size and data size when using dynamic header size approach for efficiency purpose, and related get methods

b) Classes in phatpackets.fstp.client.ui

ClientOption --- the class for user to set up preferred transfer related parameters (adjust size, alpha, block size, packet size, client UDP port, dynamic block size or not, logging or not)

ClientOptionWindow --- the GUI program implemented with Swing Dialog for users to set up ClientOption.

DirectoryEntry --- the class that compares two Java file object (file or directory) for listing order purpose in PhatClient's remote directory

DirectoryEntryRenderer --- the class that renders different outlook for directories (with directory icon) and files in PhatClient remote directory

DirectoryTable --- the table for remote directory display implemented with Swing Table

DirectoryTableModel --- the table model for DirectoryTable

FileQueueWindow --- the class that implements a dialog window for displaying multiple files to be received in a table

MultiReceiveWindow --- the class that implements a Swing Dialog for displaying receiving information for multiple files in a table

PhatClient --- the GUI main program for the client to view remote directory information, set up client options and transfer files.

ReceiveWindow --- the GUI class that implements the FSTPClient ClientListener interface for displaying insight of downloading process (transfer rate, timeout, block size, and total downloading time)

c) phatpackets.fstp.client

BitSetPacketTracker --- the class that implements the PacketTracker interface using Java BitSet to track received packets

RandomAccessStorage --- the class that implements DataStorage interface using Java RandomAccessFile class for single-threaded downloading

MultiRandomAccessStorage --- the class that implements DataStorage interface using Java RandomAccessFile class for multi-threaded downloading

d) The classes in phatpackets.fstp.server

AbstractCommand --- an abstract superclass that implements Command and CommandConstants interfaces for processing control commands

AdjustBlockSizeCommand --- the class that processes ADJUST_BLOCK_SIZE_CMD command

AdjustDelayCommand --- the class that executes ADJUST_DELAY_CMD command

BlockDelayCommand --- the class that executes BLOCK_DELAY_CMD command

BlockSizeCommand --- the class that executes BLOCK_SIZE_CMD command

CdUpCommand --- the class that executes CDUP_CMD command

Command --- the interface that defines operations for executing commands

CommandFactory --- the class that generates a singleton object which generates and stores command processing objects on the fly

CwdCommand --- the class that executes CWD_CMD command.

DoneReceivingCommand --- the class that executes DONE_RECEIVING_CMD command

KillServerCommand --- the class that executes KILLSERVER command

ListCommand --- the class that executes LIST_CMD command

NoOpCommand --- the class that executes NOOP_CMD command

PacketSizeCommand --- the class that executes PACKET_SIZE_CMD command

PhatServer --- the main program for the server side application

PingReplyCommand --- the class that executes PING_REPLY_CMD command

PortCommand --- the class that executes PORT_CMD command

PwdCommand --- the class that executes PWD_CMD command

QuitCommand --- the class that executes QUIT_CMD command

RTTCommand --- the class that executes RTT_CMD command

RandomDelayCommand --- the class that executes RANDOM_DELAY_CMD command

ResendPacketsCommand --- the class that executes RESEND_PACKETS_CMD command

SendCommand --- the class that executes SEND_CMD command

RandomAccessFileSource --- the class that implements DataSource interface using Java RandomAccessFile class

e) phatpackets.fstp.log

ClientLogger --- the class that implements FSTPClient ClientListener interface for logging session, timeout, exception, block size adjustment, complete and stop information

FSTPClientLogger --- the class that implements FSTPClient StatsListener for logging total expected packets, timeout, control command, and detailed FSTPClient working information during receiving process

FSTPServerLogger --- the class that implements FSTPServer ServerListener for logging timeout, exception, and detailed FSTPServer working information during sending process

ServerLogger --- the class that implements PhatServer ServerListener for logging sessions and control messages

f) phatpackets.fstp.util

CRC32 --- the class that implements 32 bits CRC checking for a file

MessageUtil --- the class that parses control message to extract file name

RandomTimeGen --- the class that generates random time interval used in PhatClient to avoid resource competition from multiple downloading threads

Summary

This chapter gave a brief introduction to Java implementation tool, UDP socket buffer, and the Phatpackets implementation. The Phatpackets related Java features and performance issues were presented. The package architecture and main classes were presented in the last section.

Chapter 5

EXPERIMENTS AND DATA ANALYSIS

5.1 Experiment Design

5.1.1 Experiment Goal

The experimentation should cover all typical test scenarios. The servers selected should represent a good cross section of hosts and a reasonable WAN environment. The hosts should have different resources such as CPU, memory, and access bandwidth. They are preferred to have different Operating Systems. The tests should be carried on different network load scenarios. The FTP products used for performance comparison should be typical and cover major platforms such as Solaris, Linux, and Windows. Performance, adaptability and robustness should be tested.

5.1.2 Experiment Environment

An attempt is made to cover typical test scenarios in the test. The tests were conducted on different days of the week and different times of day. Test files of different size ranged from a few hundreds bytes up to 360M bytes. An attempt is also made to experience different network loads, competing with network traffic or under light network traffic conditions. This represents a good cross section of hosts and a reasonable WAN environment.

A file named test.mov that is 15,032,348 bytes, a file named java.sh that is 35,618,816 bytes, a file named 100MB.mov that is 121,606,580 bytes, and a file named 360MB.mov that is 377,487,360 bytes are used for performance comparison illustrated here.

In these tests, four servers are used as PhatServers. They are galois.csc.uvic.ca (University of Victoria, Victoria, BC), herzberg.physics.mun.ca (Memorial University of Newfoundland, St. John's, Newfoundland), latvia.doe.carleton.ca (University of Carleton, Carleton, Ontario), and hp05.ee.ualberta.ca (University of Alberta, Edmonton, Alberta). The client machine is galliano.ee.umanitoba.ca (University of Manitoba, Winnipeg, Manitoba) which is an Ultra60 machine with SUN OS5.8.

The traditional ftp application selected is WS_FTP LE 5.08 on Windows, command line FTP on Unix and gFTP on Linux.

5.2 Performance Test Results

This section presents some performance test results as compared with traditional ftp applications. Parameters gathered are ANLB throughput, BDP throughput, and FTP throughput.

Server Name	Server Location	Distance to client (no. of routers)	Server Performance
galois.csc.uvic.ca	Victoria, BC	11	Sun Ultra5_10
herzberg.physics.mun.ca	St. John's, NF	16	28 CPU SGI ONYX
latvia.doe.carleton.ca	Carleton, ON	11	Sun Ultra5_10
hp05.ee.ualberta.ca	Edmonton, AB	7	HP Linux

Table 1. Servers Used in Performance Tests

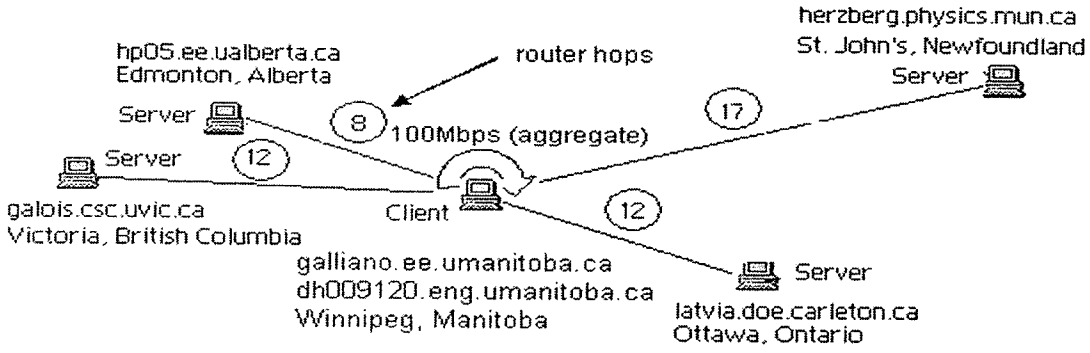


Figure 5. Server Location Topology

The tests were performed under light traffic conditions (weekend), medium traffic conditions (ordinary day during school hours), and heavy traffic conditions (Monday during school hours and with competing network traffic). Groups of data within the same test scenario were gathered under the same test condition. Table 2 clearly illustrates throughput improvements

with the ANLB approach. It seems that the galois connection is the slowest, so the download time for FTP from different servers varies considerably while Phatpackets protocol is both robust and efficient and most impressive in situations where ftp over TCP is not.

Test File	Connection	WS_FTP Throughput(s)	ANLB Throughput(s)	BDP Throughput(s)
test.mov	galios	190	5.56	6.83
test.mov	herzberg	46.03	5.94	5.40
test.mov	latvia	35	5.86	8.78
test.mov	hp05	15	5.92	6.49
java.sh	galios	420	11.50	17.81
java.sh	herzberg	115.38	13.54	14.87
java.sh	latvia	84	11.6	19.35
java.sh	hp05	36	12.4	13.20
100MB.mov	herzberg	389.56	35.29	63.47
100MB.mov	latvia	300	34.94	42.38
100MB.mov	hp05	130	29.56	59.61

Table 2. Throughput Comparison

Figure 6, Figure 7 and Figure 8 provide an example and insight into the file transfer process. Each line in the figure represents an individual connection. The red, yellow, blue and green lines represent four connections from one server. The black, pink, cyan and magenta lines represent four connections from another server.

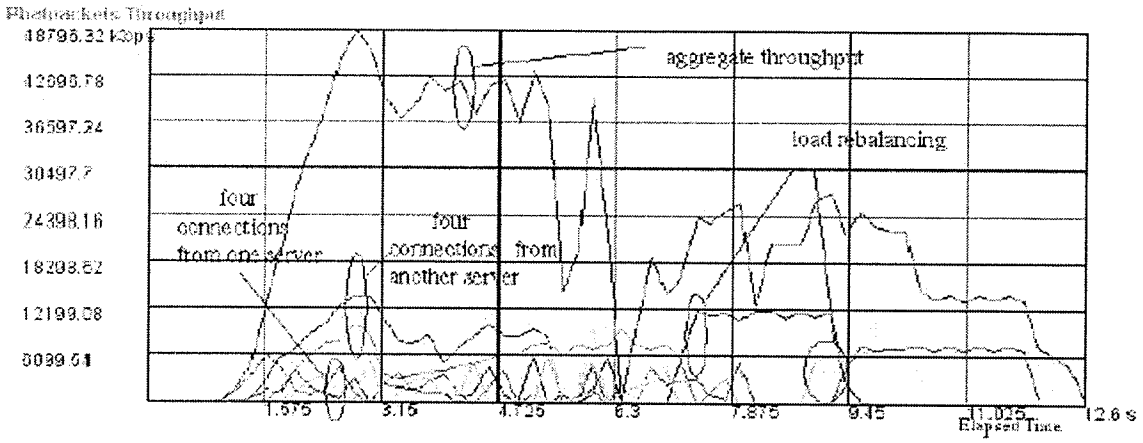


Figure 6. Throughput Graph for Downloading java.sh with ANLB

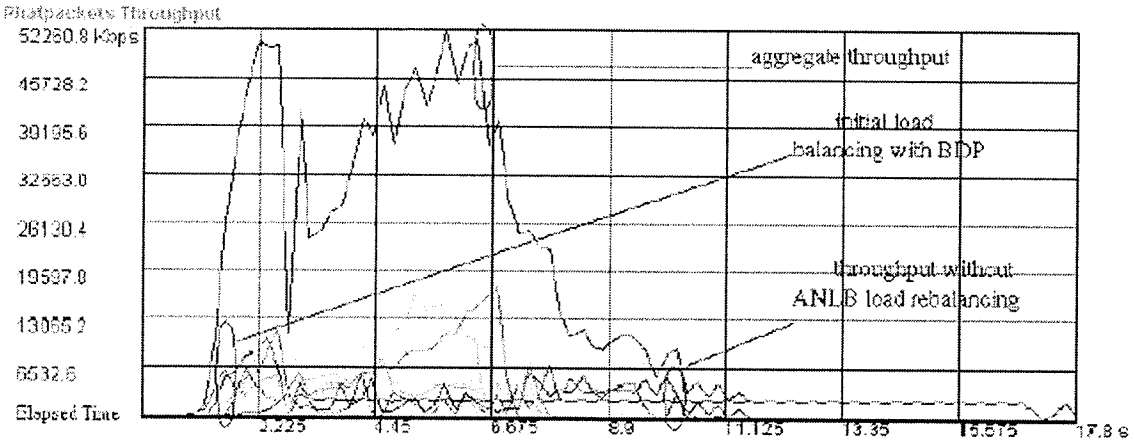


Figure 7. Throughput Graph for Downloading java.sh with BDP

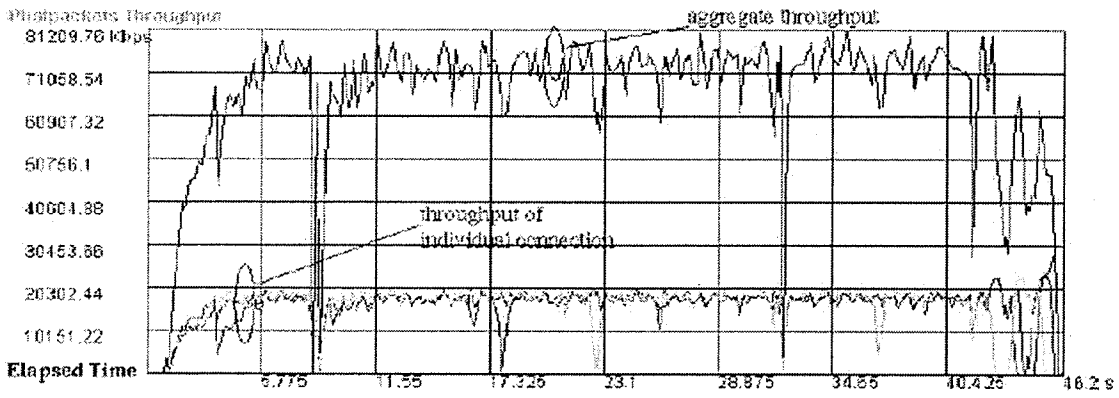


Figure 8. Throughput Graph for Downloading 360MB.mov under Light Traffic

In Figure 6, four connections 5, 7, 6, 8 took over the tasks of connections 2, 3, 4, 1. This illustrates that the ANLB approach experiences a much better load balancing than the BDP approach illustrated in Figure 7. The implication is that the ANLB approach is fairly effective. In both cases the data rate varies considerably illustrating the need for rebalancing or more sophisticated and accurate initial allocation prediction. Specially, in Figure 7 the BDP allocation schema alone clearly illustrates the bottleneck as one connection degrades from its initial value of over 7Mbps to less than 2Mbps within the initial few seconds and remains at that rate for the duration of the transfer process.

Figure 8 gives an example of downloading the 360MB.mov file with four connections from a single server under light network traffic. It indicates that load among connections from the same server tend to be well balanced.

From the test, it can be seen that FTP over TCP is more sensitive to distance and traffic load while the Phatpackets protocol adapts well to both network traffic and distance. The download

time for FTP from different servers varies considerably while the Phatpackets throughputs for all PhatServers are almost the same. Irrespective of which server the PhatClient chooses to connect to, it takes almost the same time to download with the Phatpackets implementation. The server selection and task assignment are transparent to client. From the test, it can be seen that the Phatpackets protocol is both robust and efficient and more impressive in situations where FTP over TCP is not.

There are additional implementations of FTP supporting multiple connections that would improve the throughput of FTP as compared to those presented here. The FTP comparison presented here however represents an effective baseline under the most common scenario. Enhanced FTP servers were not available as they are required to run as daemon processes and would not represent scenarios available to the typical user.

Summary

An architecture for improving the waiting time to download files from the Internet and a set of methods were presented. The basic model effectively represents an alternative "transport" layer in the TCP/IP protocol stack that better captures end-to-end performance improvements.

Reasonably impressive results have been obtained under a variety of experimental conditions providing additional evidence for research into improving transport layer protocols or application layers that directly influence transport layer processes. Experimental results

presented here show that the Phatpackets protocol is capable of sustaining an 80 Mbps connection across a WAN from one campus area network to another.

Chapter 6

CONCLUSIONS

An architecture for improving the waiting time to download files from the Internet and a set of methods are presented. The basic model effectively represents an alternative "transport" layer in the TCP/IP protocol stack that better captures end-to-end performance improvements.

Reasonably impressive results have been obtained under a variety of experimental conditions providing additional evidence for research into improving transport layer protocols or application layers that directly influence transport layer processes. Experimental results presented show that the Phatpackets protocol is capable of sustaining an 80 Mbps connection across a WAN from one campus area network to another.

REFERENCES

[AFP98] M.Allman, S. Floyd, C.Partridge, Increasing TCP's Initial Window, RFC 2414, Network Working Group, September 1998

[Aga01] P. Agarwal, A Test-bed for Performance Evaluation of Load Balancing Strategies for Web Server Systems, obtained via http://www.cse.iitk.ac.in/research/mtech1999/9911128/9911128.html, May 2001

[AOM98] M. Allman, S. Ostermann and C. Metz, FTP Extensions for IPv6 and NATs, RFC 2428, Network Working Group, September 1998

[APS99] M.Allman, V.Paxson, W.Stevens, TCP Congestion Control, RFC 2581, Network Working Group, April 1999

[BBM97] M. Baentsch, L. Baum, G. Molter, S. Rothkugel, P. Sturm, Enhancing the Web's Infrastructure: From Caching to Replication, University of Kaiserslautern, 1997

[CJR89] D. D. Clark, V. Jacobson, J. Romkey, H. Salwen, An analysis of TCP processing overhead, IEEE Communications Magazine, June 1989, pp. 23-29.

[CLZ87] D. D. Clark, M. L. Lambert, L. Zhang, NETBLT: A Bulk Data Transfer Protocol, Request for Comments 998, March 1987

[DD97] W. Dabbous, C. Diot, High Performance Protocol Architecture, obtained via http://cite-seer.nj.nec.com/correct/50220

[Dee90] J. Mogul, S. Deering, Path MTU Discovery, Request For Comments: 1191, November 1990

[Eng98] R. S. Engelschall, Practical Approaches for Distributing HTTP Traffic, New Architect Magazine, obtained via http://www.webtechniques.com/archives/1998/05/engelschall/, May 1998

[Fan00] X. Fang, Reliable File Transfer on the Internet using Distributed File Transfer (DFT), Master Thesis of University of Manitoba, 2000

[Flo00] S. Floyd, Congestion Control Principles, Request For Comments: 2914, September 2000

[Gri01] GridFTP: Protocol Extensions to FTP for the Grid, the Globus Project, obtained via http://www.globus.org/datagrid/gridftp.html, March 2001

[HL97] M.Horowitz and S. Lunt, FTP Security Extensions, RFC 2228, Network Working Group, October 1997

[HKC96] Internet Registry IP Allocation Guidelines, K. Hubbard, M. Kosters, D. Conrad, D. Karrenberg, and J. Postel, Request for Comments 896, November 1996

[Hoe96] J. C. Hoe, Improving the Startup Behavior of a Congestion Control Scheme for TCP, ACM SIGCOMM, 1996

[HM102] S. Huang, and R.D. McLeod, Phatpackets For Data Transport Within An HPC Network, IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS), 2002

[HM202] S. Huang, and R.D. McLeod, Adaptive Network Load Balancing in Phatpackets, IASTED International Conference Communications, Internet and Information Technology (CIIT), 2002

[HSM01] S. Huang, S. Silverman, and R.D. McLeod, Implementation and Experimentation, Internet Innovation Centre, University of Manitoba, 2001

[IBM02] IBM WebSphere Application Server, IBM, obtained via http://www-3.ibm.com/software/webservers/appserv/doc/v35/ae/infocenter/was/01040202.html

[ISC05] Internet Software Consortium, http://www.isc.org/, Internet Software Consortium, 2005

[JBB92] V. Jacobson, R. Braden, D. Borman, TCP Extensions for High Performance, Request For Comments: 1323, May 1992

[Jse99] Apache JServ Features, Java Apache Project, obtained via http://java.apache.org/jserv/, 1999

[KR99] J. F. Kurose and K. W. Ross, Computer Networking: A Top-Down Approach Featuring the Internet, Addison Wesley, 1999

[Mah02] Q. Mahmoud, New and Enhanced Networking Features in the JavaTM 2 Platform, Standard Edition 1.4, obtained via http://developer.java.sun.com/developer/technicalArticles/Net-working/newfeatures14.html, April 2002

[Mcc02] G. McCluskey, Tuning Java I/O Performance, Java Developer Connection, obtained via http://developer.java.sun.com, January 2002

[Mel99] S. Meloan, The Java HotSpot™ Performance Engine: An In-Depth Look, obtained via http://java.sun.com/developer/technicalArticles/Networking/HotSpot/, June 1999

[Mon00] A. S. Mondal, An intelligent load balancing tool, obtained via www.infy.com/knowledge_capital/thought-papers/CRN-0001-01.pdf, SysLab, January 1, 2000

[Nag84] J. Nagle, Congestion Control in IP/TCP Internetworks, Request for Comments 896, January 1984

[NLB00] Network Load Balancing Technical Overview, Windows 2000 White Paper, http://www.microsoft.com/windows2000/techinfo/howitworks/cluster/nlb.asp

[PHI01] S. Philopoulos, Improving the Performance of TCP over Satellite Channels, Master Thesis of University of Manitoba, 2001

[PR85] J. Postel, J. Reynolds, File Transfer Protocol, RFC 959, Network Working Group, October 1985

[Sch97] U. Schwantag, An Analysis of the Applicability of RSVP, http://ns.uoregon.edu/ursula/thesis/thesis.html

[Sil00] S. Silverman, First version of Phatpackets design and implementation, the University of Manitoba.

[SMM98] J. Semke, J. Mahdavi, M. Mathis, Automatic TCP Buffer Tuning, obtained via http://www.psc.edu/networking/projects/auto/, Pittsburgh Supercomputing Center, September 1998

[Sol92] K. Sollins, THE TFTP PROTOCOL (REVISION 2, Request For Comments: 1350, July 1992

[Ste97] W. Stevens, TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, Request For Comments: 2001, Network Working Group, January 1997

[Tec02] TechWeb: The Business Technology Network, obtained via http://www.techweb.com/encyclopedia/defineterm?term=internet

[Web02] Web100 Concept Paper, obtained via http://www.web100.org/docs/concept_paper.php

[Wid00] J. Widmer, Equation-Based Congestion Control, Diploma Thesis, Department of

Mathematics and Computer Science, University of Mannheim, February 2000, http://www.icsi.berkeley.edu/~widmer/tfrc/thesis/thesis.html

[YCE97] C. Yoshikawa, B. Chun, P. Eastham, A. Vabdat, T. Anderson, and D. Culler, Using Smart Clients to Build Scalable Services, University of California, 1997

[Zuk01] J. Zukowski, New I/O Functionality for Java 2 Standard Edition 1.4, Java Developer Connection, obtained via http://developer.java.sun.com, December 2001

Appendix A

THE PHATPACKETS PROTOCOL CONTROL COMMAND

(Using BNF notation)

ADJUST_BLOCK_SIZE_CMD

ADJUSTBLOCKSIZE<SP><streamid><SP><blocksize><CRLF>

Adjust the block size to new block size in packets.

ADJUST_DELAY_CMD

ADJUSTDELAY<SP><streamid><SP><blockdelay><CRLF>

Adjust the block delay to new block delay in milliseconds.

BLOCK_SIZE_CMD

BLOCKSIZE<SP><blocksize><CRLF>

Set the block size to block size in packets.

BLOCK_DELAY_CMD

BLOCKDELAY<SP><blockdelay><CRLF>

Set the block delay to block delay in milliseconds.

BW_CMD

TESTBANDWIDTH<CRLF>

The client requests the server to send four full size test packets to calculate downlink bandwidth.

CDUP_CMD

CDUP<CRLF>

Change to parent directory.

CWD_CMD

CWD<SP><pathname><CRLF>

Change working directory to pathname.

DIR_CMD

DIR<CRLF>

Request directory information of other servers.

DONE_RECEIVING_CMD

DONERECEIVING<SP><streamid><CRLF>

Send to indicate done receiving with the specific steam id.

LIST_CMD

LIST<CRLF>

Request the server to send a list of files in current directory.

MORE_CMD

MORE<SP><filename><SP><filesize><SP><CRLF>

The client requests the server to send the specific file with specific length and suggests the server to send using the specific number of connections.

NOOP_CMD

NOOP<CRLF>

The client sends this command and keeps the sending time, and calculates the Round Trip Delay Time (RTT) after receiving NOOP_RPLY.

PACKET_SIZE_CMD

PACKETSIZE<SP><packetsize><CRLF>

The client sends the preferred packet size in bytes.

PING_CMD

PING<SP><streamid><CRLF>

The server pings client to see if the client is still alive.

PORT_CMD

PORT<SP><localport><CRLF>

The client sends its local UDP port number.

QUIT_CMD

QUIT<CRLF>

The client sends this command to terminate the control connection.

RANDOM_DELAY_CMD

RANDOMDELAY<SP><delay><CRLF>

The client sends this command to request the server to delay the specific milliseconds before sending next batch.

RESEND_PACKETS_CMD

RESENDPACKETS<SP><streamid>{<SP><sequenceid>}<CRLF>

The client requests the loss packets with sequence numbers of the specific stream id.

RTT_CMD

ROUNDTRIPDELAY<SP><roundtripdelay><CRLF>

The client sends its RTT in milliseconds to the server so that the server can adjust its retransmission timeout timer

SEND_CMD

SEND<SP><filename><SP><startposition><SP><expectedpackets><SP>[newStreamId][oldStreamId]<CRLF>

The client requests a file segment of specific file name that starts from start position in packets, and has expected number of packets. If this SEND_CMD is the first SEND_CMD for multiple connections from same server, then use newStreamId to request for a new stream id, or else use oldStreamId.

BW_RPLY

228<SP><testdata><CRLF>

The server sends four full packet size packets in response to BW_CMD. The client then calculates the time interval from receiving first packet to the fourth packet, and uses this time as downlink bandwidth.

CONNECTION_ERROR_RPLY

420<SP>Control connection error.<CRLF>

The server sends this message if there are some errors during control connection operations. The server then terminates all related data connections and this control connection.

COMMAND_UNRECOGNISED_RPLY

500<SP><command><SP>is an unrecognized command.<CRLF>

The server replies with the message if the command is not a valid command.

COMMAND_OKAY_RPLY

200<SP><message><CRLF>

The server replies with the suitable message after executing the requested command.

DIR_RPLY

229<SP>{<LF><filepath/filename><HT><filesize><HT><lastmodified>}<CRLF>

The server replies with the information of all the files in directory and subdirectory managed by the PhatServer.

FILE_ERROR_RPLY

550<SP>file<SP><filename><SP>not found.<CRLF>

The server replies with this message if the requested file with file name is not a valid file in current directory.

FILE_INFO_RPLY

251<SP>FILESIZE<SP><filesize><SP>CRC<SP><crc><SP>STREAMID<SP><streamid><SP>SERVERPORT<SP><localport>

The server replies client's SEND_CMD command with the specific file's length, 32 bits Cyclic Redundancy Check (CRC), stream id, and server's UDP port number.

FILE_UNAVAILABLE_RPLY

450< SP>Thre is a problem with preparing the local file.<CRLF>

The server replies with this message if the requested file of file name can't be found in current directory.

GOODBYE_RPLY

221<SP>Thank you for using this server. Be phat.<CRLF>

The server sends this message when receiving client's QUIT_CMD and terminates the related control connection.

LOGGED_IN_RPLY

230<SP>User logged in, proceed with style.<CRLF>

The server sends this message when the client logs in.

MORE_RPLY

225{<SP><servername><SP><directory><filename>}<CRLF>

The server checks its database for known servers' directories about the specific file of the specific version (same file size), then returns server name/directory/file name pairs.

NOOP_RPLY

227<CRLF>

The server sends NOOP_RPLY immediately after receiving NOOP_CMD.

NOT_IMPLEMENTED_RPLY

502<SP><message><CRLF>

The server sends the message if the requested function has not been implemented yet.

NOT_SENDING_ERROR_RPLY

520<SP>Not currently sending data.<CRLF>

Reply the client RESEND_PACKETS_CMD request with this reply if the data connection with this stream id is not sending.

SENDING_DIR_RPLY

152<SP>LISTING<SP><pathname>[<LF><filename><HT><length><HT><lastmodified>][{<LF>[/]<filename><HT><length><HT><lastmodified>}]

The server replies with LIST_CMD. If the requested name is a file, then replies with file information. If the requested name is a directory, list the contents in this directory. Note, to distinguish file with directory, a "/" is put in front of file name to indicate it is a directory.

SENDING_FILE_RPLY

150<SP>Sending<SP><filename><SP><clientport><CRLF>

The server replies the SEND_CMD with the requested file name, and the client UDP port.

SYNTAX_ERROR_PARAMS_RPLY

501<SP><message>

The server replies with the message if the client's command is not in correct format (missing parameters, wrong data types, etc.).