AN ABSTRACT OF THE THESIS OF

Sunand Tullimalli for the degree of Master of Science in Computer Science presented on September 6, 2006.

Title: Multimedia Streaming Using Multiple TCP Connections

Abstract approved: ______________________________

Thinh Nguyen

Packet loss, delay and time-varying bandwidth are three main problems facing multimedia streaming applications over the Internet. Existing techniques such as Media-aware network protocol, network adaptive source and channel coding, etc. have been proposed to either overcome or alleviate these drawbacks of the Internet. But these techniques either need specialized codecs or require significant changes in the network infrastructure. In this thesis, we propose the MultiTCP system, a receiver-driven, TCP-based application-layer transmission protocol for multimedia streaming over the Internet. The proposed algorithm aims at providing resilience against SHORT TERM insufficient bandwidth by using MULTIPLE TCP connections for the same application. Our proposed system enables the application to achieve and control the desired sending rate during congested periods, by using multiple TCP connections and dynamically changing the receiver's window size for each connection, which cannot be achieved using traditional TCP. Finally, the proposed system is implemented at the application layer. Thus no kernel modification is necessary, which ensures easy deployment. To demonstrate the performance of the proposed system, we present simulation and experimental results on the PlanetLab network to establish its advantages over the traditional single TCP based approach.

©Copyright by Sunand Tullimalli

September 6, 2006

All Rights Reserved

Multimedia Streaming Using Multiple TCP Connections

by

Sunand Tullimalli

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented September 6, 2006
Commencement June 2007

Master of Science thesis of Sunand Tullimalli presented on September 6, 2006

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Sunand Tullimalli, Author

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to my advisor Dr.Thinh Nguyen for guidance and ideas which helped me complete my thesis. Also I would like to thank my project mate Richard P.Edgecombe for his co-operation and suggestions at times when I needed them the most.

I would like to thank the Information Services Department, for keeping me funded for the 2 years of my studies and for being accommodating so that I could focus on my studies.

I would like to thank Dr.Bella Bose, Dr.Timothy Budd, Dr.Michael Quinn, Dr.Rajeev Pandey, and Dr.Toshimi Minoura whose courses and ideas have influenced me, and hence this project, a lot.

I would like to extend my gratitude to my friends Krishnan Kolazhi and Rohit Kamath for taking time off their activities and helping me solve some of the problems I encountered. I would also thank my roommates Rajat Panwar and Rajagopal for helping me complete my thesis. I also thank my other roommates Prashanth, Amar and Sawan Tilak for supporting me in various matters. I also thank my other friends who supported me in completing my work without any obstacles.

I would like to thank my family back in India for their constant support and encouragement without which this would not have been possible. Above all, thanks to the Almighty for showering his blessings on me.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

MULTIMEDIA STREAMING USING MULTIPLE TCP CONNECTIONS

CHAPTER 1

INTRODUCTION

1.1 The Problem

In recent years, there has been an explosive growth of multimedia applications over the Internet. All major news networks such as ABC and NBC now provide news with accompanying video clips. Several companies, such as MovieFlix [1], also offer video on-demand to broadband subscribers. However, the quality of videos being streamed over the Internet is often low due to insufficient bandwidth, packet loss, and delay. To view a DVD quality video from an on-demand video server, a customer must download either the entire video or a large portion of it before starting to playback in order to avoid pauses caused by insufficient bandwidth during a streaming session. To deal with these problems, many techniques have been proposed to enable efficient multimedia streaming over the Internet. The source coding community has proposed scalable video [2][3], error-resilient coding, and multiple description [4] for efficient video streaming over best-effort networks such as the Internet. A scalable video bit stream is coded in a way that enables the server to easily and efficiently adapt the video bit rate to the current available bandwidth. Error-resilient coding and multiple description aim at improving the quality of video in the presence of packet loss and long delays caused by retransmission. Channel coding techniques are also used to mitigate long delays for real-time applications such as video conferencing and IP-telephony [5].

From a network infrastructure perspective, Differentiated Services [6][7] and Integrated Services [8][7] have been proposed for improving the quality of multimedia applications by providing preferential treatments to various applications based on their bandwidth, loss, and delay requirements. More recently, path diversity architectures that combine multiple paths and either source or channel coding have been proposed for providing larger bandwidth and efficiently combating packet loss [9][10][11]. Nonetheless, these approaches cannot be easily deployed as these require significant changes in the network infrastructure.

1.2 Existing Solutions

The most straightforward approach is to transmit standard-based multimedia via existing IP protocols. The two most popular choices are Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). A single TCP connection is not suitable for multimedia transmission because its congestion control might cause large fluctuations in sending rate. Unlike TCP, an UDP-based application is able to set the desired sending rate. If the network is not congested too much, the UDP throughput at the receiver end would be approximately equal to the sending rate. Since the ability to control the sending rate is essential for interactive and live streaming applications, majority of multimedia streaming systems use UDP as a basic building block for sending packets over the Internet. However, UDP is not a congestion aware protocol since it does not reduce its sending rate in presence of network congestion, and therefore, might potentially result in a congestion collapse. Congestion collapse occurs when a router drops a large number of packets due to its inability to handle a large amount of traffic from many senders at the same time. TCP-Friendly Rate Control Protocol (TFRC) has been proposed for multimedia streaming. It transmits packets via UDP but incorporates TCP-like congestion control mechanism [12]. Another drawback of using UDP is its lack of reliable transmission and hence the application must deal with packet loss.

1.3 Our Strategy

Based on these drawbacks of single TCP and UDP, we propose a new receiver-driven, TCP-based application-layer transmission protocol for multimedia streaming over the Internet. The first version of this work was published in [13]. In particular, the proposed system, called *MultiTCP*, aims at providing resilience against *short-term* insufficient bandwidth by using *multiple* TCP connections for the same application. Additionally, this system enables the application to achieve and control the sending rate during congested period, which in many cases, cannot be achieved using a single TCP connection. Finally, the proposed system is implemented at the application layer, and hence, kernel modification to TCP is not necessary.

CHAPTER 2

OVERVIEW

Having studied the traditionally employed approaches towards multimedia streaming using single TCP and UDP and the drawbacks associated with each of these approaches, we propose a *MultiTCP* system capable of achieving the desired throughput by using multiple TCP connections.

2.1 System Goals

The proposed system and the associated algorithm are designed to achieve the following:

1. A receiver-driven, TCP-based application-layer transmission protocol for multimedia streaming using multiple TCP connections over the Internet.

2. Resilience against *short term* insufficient bandwidth by using *multiple* TCP connections.

3. The application that is able to achieve and control the desired sending rate during congested periods, which cannot be achieved using traditional TCP.

4. Capability of dynamically increasing and decreasing the number of TCP connections depending on the congestion in the network.

5. A design that can be implemented at the application layer, and hence, no kernel modification to TCP is necessary.

2.2 Thesis Contribution and Content Organization

Present thesis discusses the issues involved in multimedia streaming using single TCP connection and UDP. Further, the general client-server architecture which provides an insight into our system are discussed. This provides the overview, design, analysis and working of the *MultiTCP* system and also provides an algorithm that helps achieving all system requirements. In addition, this thesis provides the design details of the *MultiTCP* system and deeply examines the different components of the system. This thesis also focusses on working of the system. Finally, a series of experiments evaluate the performance of overall system.

Rest of the content is organized as follows. In Chapter 3, we discuss the Internet transport protocols, TCP and UDP. In addition, we discuss the drawbacks of TCP for multimedia streaming. Chapter 4 gives a broad overview of the system designed along with features, advantages and disadvantages of the system. This is followed by Chapter 5 that discusses the client-server interaction and network programming using Berkeley sockets. Next, in Chapter 6 we analyze how the proposed system works. In the next Chapter 7 we discuss design issues involved while developing the *MultiTCP* system and provide overview of the current design with a brief idea of how it works. Following this, we present the results of some NS simulations and some Internet experiment results across PlanetLab [25] nodes in Chapter 8. Finally, we discuss our conclusions and examine the related work in the area of multimedia streaming.

CHAPTER 3

TCP AND UDP

3.1 The Internet Transport Protocols

The Internet has two main transport protocols, a connectionless protocol and a connection-oriented protocol. In the following sections we elaborate both of these. The connectionless protocol is the UDP and the connection-oriented protocol is TCP.

3.2 Introduction to UDP

The Internet protocol suite supports a connectionless transport protocol, UDP (User Datagram Protocol). UDP provides a way for the applications to send encapsulated IP datagrams and send them without having to establish a connection. UDP transmits segments consisting of an 8-byte header followed by the payload. The header is shown in Figure 3.1.

The two ports serve to identify the end points within the source and destination machines. When a UDP packet arrives, its payload is handed to the process attached to the destination port. Without these port fields, the transport layer wouldn't know what to do with the packet. With them, it delivers segments correctly.

The source port is primarily needed when a reply needs to be sent back to the source. By copying the source port field from the incoming segment into

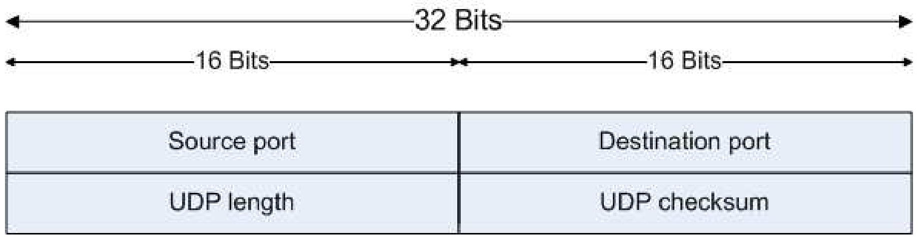


FIGURE 3.1: UDP Header

the destination port field of the outgoing segment, the process sending the reply can specify which process on the sending machine should receive it.

The UDP length field includes an 8-byte header and the data. The UDP checksum is optional and stored as 0 if not computed. Turning it off is undesirable unless the quality of the data doesn't matter.

Some of the things that UDP doesn't perform are flow control, error control, and retransmission upon receipt of a bad segment. However it provides an interface to the IP protocol with an added feature of de-multiplexing multiple processes using the ports.

One area where UDP is especially useful in the area of client-server situations. Often, the client sends a short request to the server and expects a short reply back. If either the request or reply is lost, the client can just time out and try again. In this case fewer messages are required (one in each direction) as compared to a protocol requiring an initial setup. An application that uses UDP is Domain Name System (DNS).

3.3 Introduction to TCP

UDP is a simple protocol and it has some niche uses, such as client-server interactions and multimedia. At the same time, for most Internet applications, reliable and sequenced delivery is needed. UDP can not provide this, so another protocol is needed. It is called TCP and is the main workhorse of the Internet.

TCP (Transmission Control Protocol) was specifically designed to provide a reliable end-to-end byte stream over an unreliable internetwork. An internetwork differs from a single network because different parts may have wildly different topologies, bandwidths, delays, packet sizes and other parameters. TCP was designed to dynamically adapt to properties of the internetwork and to be robust to many kinds of failures.

Each machine supporting TCP has a TCP transport entity. It could be either a library procedure, a user process, or part of kernel. In all cases, it manages TCP streams and interfaces with the IP layer. A TCP entity accepts user data streams from local processes, breaks them up into pieces, and sends each piece as a separate IP datagram. When datagrams containing TCP data arrive at a machine, these are forwarded to the TCP entity that reconstructs the original byte streams.

The IP layer doesn't ensure that datagrams will be properly delivered. Therefore, it is up to TCP to time out and retransmit the datagrams as needed. Datagrams that arrive may well do so in the wrong order. It is up to TCP to reassemble them into messages in a proper sequence. In short, TCP must furnish the reliability that most users want and that IP does not provide.

3.3.1 The TCP Protocol

A key feature of TCP that dominates the protocol design, is that every byte on a TCP connection has its own 32-bit sequence number. Separate 32-bit sequence numbers are used for acknowledgements and for the window mechanism, which will be discussed later in this chapter.

The sending and receiving TCP entities exchange data in the form of segments. A TCP segment consists of a fixed 20-byte header (plus an optional part) followed by zero or more data types. The TCP software decides how big the segments should be. It can accumlate data from several writes into one segment or split the data from one write over multiple segments. Two limits restrict the segment size.

1. Each segment including the header must fit in 65,515-byte IP payload.

2. Each network has a maximum transfer unit (MTU) and each segment must fit in MTU. In practice, the MTU is generally 1500 bytes and thus defines the upper bound on the segment size.

The basic protocol used by TCP entities is the sliding window protocol. When the sender transmits a segment, it also starts a timer. When the segment arrives at the destination, the receiving TCP entity sends back a segment bearing an acknowledgement number equal to the next sequence number it expects to receive. If the sender's timer goes off before the acknowledgement is received, the sender transmits the segment again. Segments can arrive out of order and can also be delayed so long in transit that the sender times out and retransmits them. Retransmissions may include different byte ranges from the original transmission, requiring a careful administration to keep track of which bytes have so far been correctly received. However, since each byte in the stream has its own unique offset, it can be done. The layout of a TCP segment is shown in Figure 3.2. The connections are established in TCP by means of a three-way handshake that will be discussed later in Chapter 5.

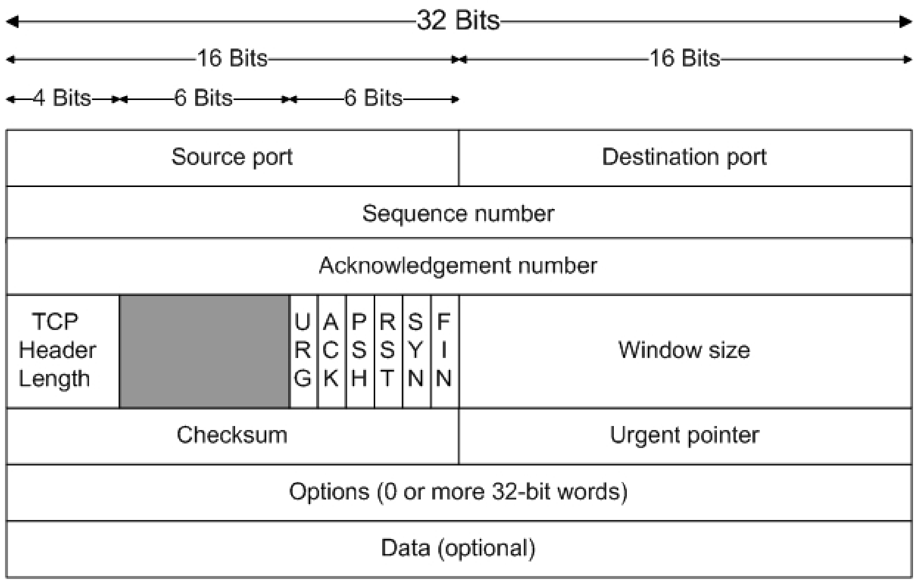


FIGURE 3.2: TCP Header

3.3.2 Drawbacks of TCP for multimedia streaming

TCP is unsuitable for multimedia streaming due partly to its fluctuating throughput and its lack of precise rate control. TCP is designed for end-to-end reliability and fast congestion avoidance. To provide end-to-end reliability, a TCP sender retransmits the lost packets based on the packet acknowledgement from a TCP receiver. To react quickly to network congestion, TCP controls the sending rate using a window-based congestion control which is discussed below.

3.3.2.1 Window-based congestion control mechanism

In this mechanism, sender keeps track of a window of maximum number of unacknowledged packets, i.e., packets that have not been acknowledged by the receiver. In the steady state, the sender increases the window size W by $1/W$ after successfully receiving an acknowledged packet, or equivalently, it increases the sending rate by one packet per round trip time. Upon encountering a loss, the window size is reduced by half, or equivalently, the sending rate is cut in half. In TCP, the receiver has the ability to set a maximum window size for unacknowledged packets, hence imposing a maximum sending rate. Thus, in a non-congestion scenario, application at the receiver can control the sending rate by appropriately setting the window size. On the other hand, during congestion, the actual throughput can be substantially low since the maximum window size may never be reached.

Based on the above discussion, we observed that a single packet loss can drop the TCP throughput abruptly and the low throughput lingers due to the slow increase of the window size. If there is a way to reduce this throughput reduction without modifying TCP, a higher throughput with proper congestion control and reliable transmission can be effectively provided. In addition, if there is a way to control the TCP sending rate during congestion, then TCP can be made suitable for multimedia streaming. Unlike non real-time applications such as file transfer and email, precise control of sending rate is essential for interactive and live streaming applications due to following reasons:

1. Sending at too high a rate can cause buffer overflow in certain receivers with limited buffer such as mobile phones and PDAs.

2. Sending at a rate lower than the coded bit rate results in pauses during a streaming session, unless a large buffer is accumulated before playback.

In Chapter 6, we propose a system that can dynamically distribute streaming data over multiple TCP connections per application to achieve higher throughput and precise rate control. The control is performed entirely at the receiver side and thus, suitable for streaming applications where a single server may simultaneously serve hundreds of receivers.

CHAPTER 4

SYSTEM ARCHITECTURE

The architecture presented in this chapter is the Client-Server architecture as it represents the MultiTCP system's architecture discussed in the present research.

4.1 Client-Server Architecture

Client-server architecture is a network architecture that separates client from the server. Each instance of the client software can send requests to a server or application server. There are many different types of servers. Some examples include, a file server, terminal server, mail server and a video streaming server. While their purpose varies somewhat, the basic architecture remains the same.

Although this idea is applied in a variety of ways on different kinds of applications, the simplest example is the current use of web pages on the internet. For example, if one reads an article on *Wikipedia*, computer and web browser being used would be considered a client. The computers, databases, and applications that make up *Wikipedia* would be considered the server. When user's web browser requests a particular article from *Wikipedia*, the Wikipedia server finds all of the information required to display the article in the Wikipedia database, assembles it into a web page, and sends it back to user's web browser to look at.

A client-server architecture is intended to provide a scalable architecture, where each computer or process on the network is either a client or a server. Server software often runs on powerful computers dedicated for exclusive use to run the business application. Client software, on the other hand, generally runs on common PCs or workstations. Clients get all or most of their information and rely on the application server for things such as configuration files, stock quotes, business application programs. Application server also helps in offloading computer-intensive application tasks back to the server in order to keep the client computer (and client computer user) free to perform other tasks. The properties of *Server* and *Client* are as follows.

Properties of a server:

1. Passive (Slave).

2. Waiting for requests.

3. On requests serves them and sends a reply.

Properties of a client:

1. Active (Master).

2. Sending requests.

3. Waits until reply arrives.

Servers can be either stateless or stateful. A stateless server does not keep any information between requests. For example, a HTTP server for static HTML pages. A stateful server can remember information between requests.

The scope of this information can be global or session. An example includes Apache Tomcat.

Another type of network architecture is known as a peer-to-peer architecture because each node or instance of the program is both a "client" and a "server" each having equivalent responsibilities.

4.2 System Overview

The proposed system follows an Object Oriented approach and its design is based on the responsibilities of entities.

4.2.1 Component Classification

Depending upon the responsibility that a component takes in the proposed MultiTCP system, the component is classified as either a Server or a Client.

1. Server: Server is a node which handles all the requests from the Clients. It listens the requests from the clients and maintains all the requests from the clients in a queue. It serves the requests from the clients one-by-one on a first come first serve basis.

2. Client: A Client is a node which requests the server for the required data. After sending a request to the server, it waits until it gets the reply from the server.

4.2.2 Component Interaction

Figure 4.1 shows the components of the system and their interaction in general. The two main components of the system are: 1) Server and 2) Client. The client-server scenario in our system is as follows.

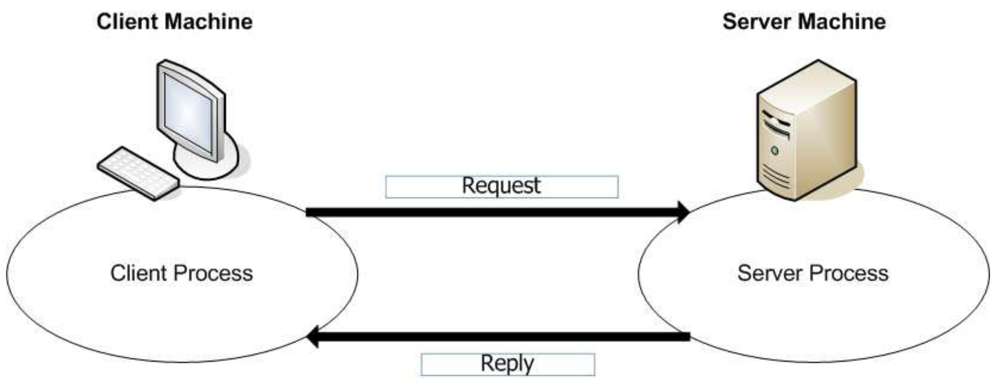


FIGURE 4.1: *Client-Server* model

1. The server process is started on some computer system. It initializes itself, and then goes into sleep, waiting for a client to contact it requesting for some data transfer.

2. The client process is started on another system that is connected to the server's system with the internetwork. The client process sends a request across the internet to the server requesting service, i.e., data transfer.

3. When the server process has finished providing its service to the client, the server goes back to sleep, waiting for the next client request to arrive.

4.3 Features of the System

The features of the system are as listed below.

1. It is a receiver-driven, TCP-based application-layer transmission protocol for multimedia streaming over the Internet.

2. It provides resilience against *short-term* insufficient bandwidth by using *multiple* TCP connections.

3. It enables the application to achieve and control the desired sending rate during congested periods, by using multiple TCP connections and dynamically changing the receiver's window size for each connection, which cannot be achieved using traditional TCP.

4. It enables the application to dynamically increase and decrease the number of TCP connections for achieving desired throughput.

4.4 Advantages of the System

There are a number of advantages of having a MultiTCP system design. These are listed below.

1. This *MultiTCP* system is a receiver-driven, TCP-based application-layer transmission protocol system. The user on the receiver side has to specify the desired throughput and it should not exceed the bandwidth limits with respect to the connection established between the server and client for data transfer.

2. This system provides resilience against *short-term* insufficient bandwidth by using multiple TCP connections. As the number of TCP connections increases, the bandwidth shared by the application will also increase and as a result, the throughput will also increase.

3. This system enables the application to achieve and control the desired sending rate during congested periods, which cannot be achieved using traditional TCP.

4. This system is implemented at the application layer, thus no kernel modification will be necessary, leading to easy deployment.

5. This system has the ability to dynamically increase and decrease the number of TCP connections depending on the congestion in the network for achieving the desired throughput.

4.5 Drawbacks of the System

1. The total throughput may exceed the desired throughput by a large amount if the sending rate of each TCP connection is too high.

2. If only a small number of TCP connections are required to exceed the desired throughput, this system may not be resilient to the sudden increase in traffic.

CHAPTER 5

SOCKET PROGRAMMING

There are two components of the system, as mentioned before, that communicate to each other in order to transfer control data and the actual data to one another. These are the *Server* and the *Client*. In this chapter we discuss these components in more details. We emphasize the utility of these components while also keeping their overall utility in perspective.

5.1 Socket Programming with TCP

The server process and client process running on different machines communicate with each other by sending messages into sockets. Each process is analogous to a house and the process's socket is analogous to a door. As shown in Figure 5.1, the socket is the door between the application process and TCP. The application developer has control of everything on the application-layer side of the socket. However, it has little control of the transport-layer side. At the very most, the application developer has the ability to fix a few TCP parameters, such as maximum buffer size and maximum segment sizes.

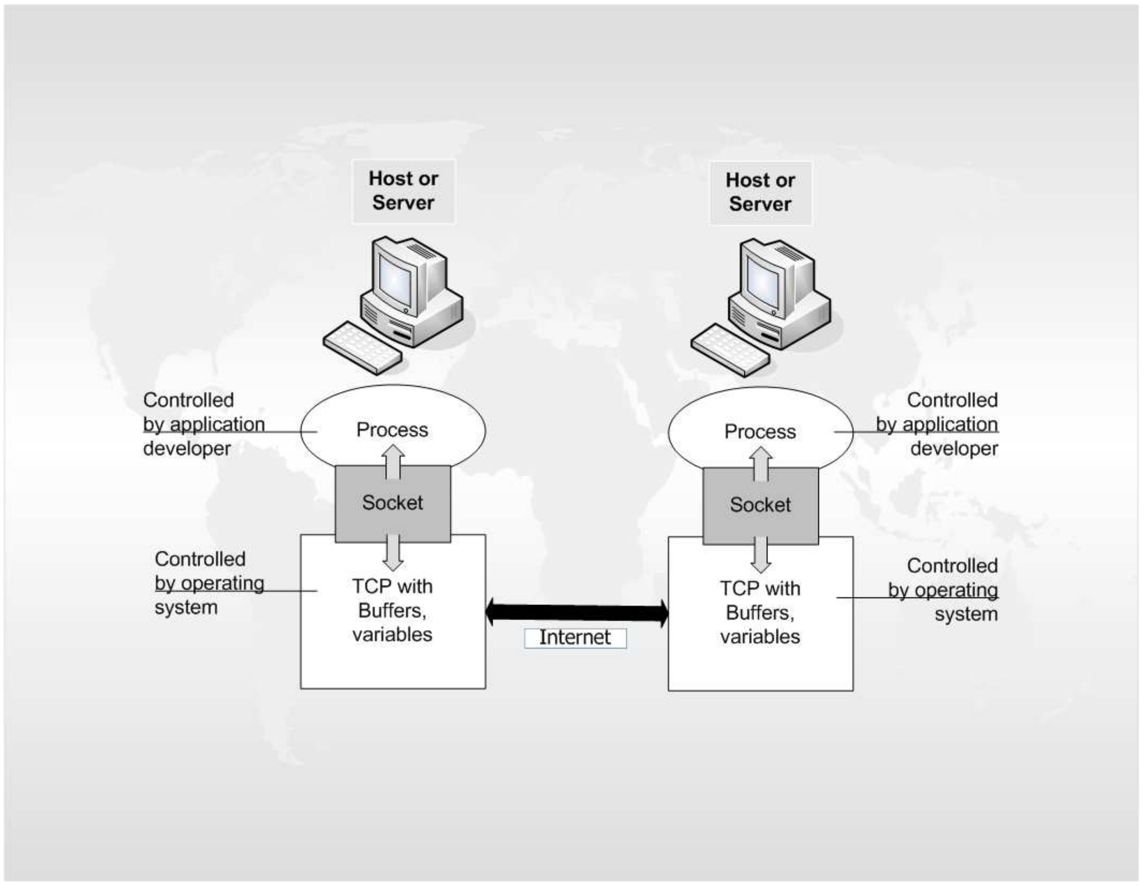

Host or Server
Host or Server
Controlled by application developer
Process
Socket
Controlled by operating system
TCP with Buffers, variables
Internet
Process
Socket
TCP with Buffers, variables
Controlled by application developer
Controlled by operating system

FIGURE 5.1: Processes Communicating through TCP sockets

5.2 Client-Server Interaction

Now let us take a closer look at the interaction of client and server programs. The client has the job of initiating contact with the server. In order for server to be able to react to the client's initial contact, the server must be ready. This implies that:

1. The server program cannot be dormant; it must be running as a process before the client attempts to initiate contact.

2. The server program must have some sort of door (that is, socket) that welcomes some initial contact from a client running on an arbitrary machine.

Using our house/door analogy for a process/socket, we will sometimes refer to the client's initial contact as "knocking on the door."

With the server process running, the client process can initiate a TCP connection to the server. This is done in the client program by creating a socket. When the client creates its socket, it specifies the address of the server process, namely, the IP address of the server (or the server name) and the port number of the process. Upon creation of the socket, TCP in the client initiates a three-way handshake, and establishes a TCP connection with the server. The TCP handshake is completely transparent to client and server processes.

5.2.1 Three-way handshake

During three-way handshake, the client process knocks on welcoming door of the server process. When the server "hears" the knocking, it creates a new door (that is, a new socket) that is dedicated to that particular client.

From the applications perspective, the TCP connection is a direct virtual pipe between the client's socket and the server's connection socket. The client process can send arbitrary bytes into its socket; TCP guarantees that server process will receive (through the connection socket) each byte in the order sent. Furthermore, just as people can go in and out of the same door, the client process can also receive bytes from its socket and the server process can also send bytes into its connection socket. This is illustrated in Figure 5.2.

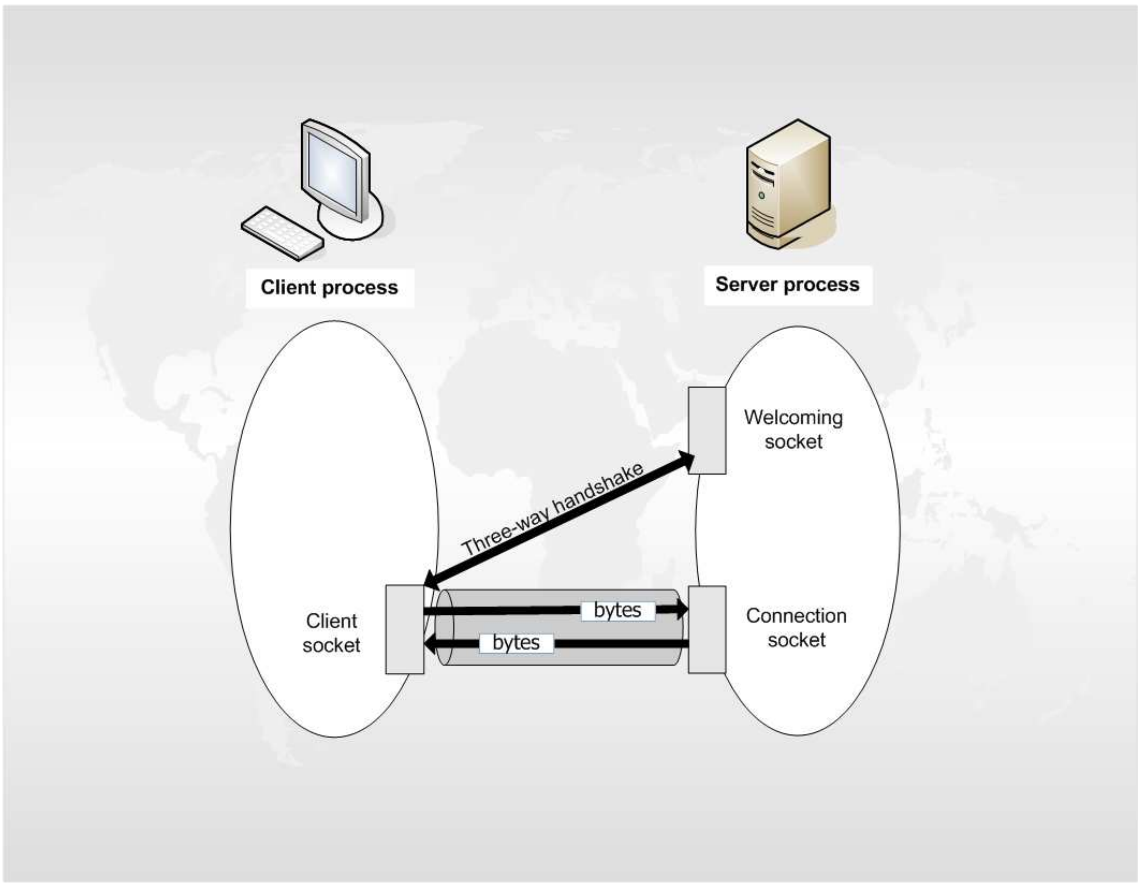


FIGURE 5.2: Client Socket, Welcoming Socket and Connection Socket

Since sockets play a central role in client/server applications, client/server application development is also referred to as socket programming. Berkeley sockets are used in developing this *MultiTCP* system. A brief discussion about these is provided below.

5.3 Berkeley Sockets

The socket primitives used in Berkeley UNIX for TCP are widely used for Internet programming. These are listed in Table 5.1.

Primitive	**Meaning**
SOCKET	Create a new communication end point
BIND	Attach a local address to a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Block the caller until a connection attempt arrives
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

TABLE 5.1: *The Socket primitives for TCP*

The first four primitives in the list are executed in the order by servers. The SOCKET primitive creates a new end point and allocates table space for it within the transport entity. The parameters of the call specify the addressing format to be used, the type of service desired (e.g., reliable byte stream), and the protocol. A successful SOCKET call returns an ordinary file descriptor for use in succeeding calls in the same way that an OPEN call does.

Newly created sockets do not have network addresses. These are assigned using BIND primitive. Once a server has bound an address to a socket, remote clients can connect to it. The reason for not having the SOCKET call create an address directly is that some processes care about their address (e.g., they have been using this address and everyone knows this address), whereas others do not care.

LISTEN call allocates space to queue incoming calls when several clients try to connect at the same time. To block waiting for an incoming connection, the server executes an ACCEPT primitive. When a data unit asking for a connection arrives, the transport entity creates a new socket with the same properties as the original one and returns a file descriptor for it. The server can then fork off a process or thread to handle the connection on the new socket and go back to waiting for the next connection on the original socket. ACCEPT returns a normal file descriptor, which can be used for reading and writing in the same standard way, as files.

Now let us look at the client side. Here, too, a socket must first be created using the SOCKET primitive, but BIND is not required since the address used is immaterial to the server. The CONNECT primitive blocks the caller and actively starts the connection process. When it completes (i.e., when the appropriate data unit is received from the server), the client process is unblocked and the connection is established. Both sides can now use SEND and RECV to transmit and receive data over the full-duplex connection. The standard

UNIX READ and WRITE system calls can also be used if none of the special options of SEND and RECV are required. Connection release with sockets is symmetric. When both sides have executed a CLOSE primitive, the connection is released.

CHAPTER 6

MULTITCP OVERVIEW AND ANALYSIS

This chapter provides theoretical background of this *MultiTCP* system. As mentioned in Section 3.3.2, the throughput reduction of TCP is attributed to the combination of (a) reduction of the sending rate by half upon detection of a loss event and (b) the slow increase of sending rate afterward or the *congestion avoidance.* To alleviate this throughput reduction, one can modify TCP to (a) reduce the sending rate by a small factor other than half upon detection of a loss, or (b) speed up the congestion avoidance process, or (c) combine both (a) and (b). There are certain disadvantages associated with these approaches.

1. These changes affect all TCP connections and must be performed by recompiling the OS kernel of the sender machine.

2. Changing the decreasing multiplicative factor and the additive term in isolated machines may potentially lead to instability of TCP in a larger scale of the network.

3. It is not clear how these factors can be changed to dynamically control the sending rate.

As such, a different approach is proposed: instead of using a traditional, single TCP connection, we use multiple TCP connections for a multimedia streaming application. This approach does not require any modification to the

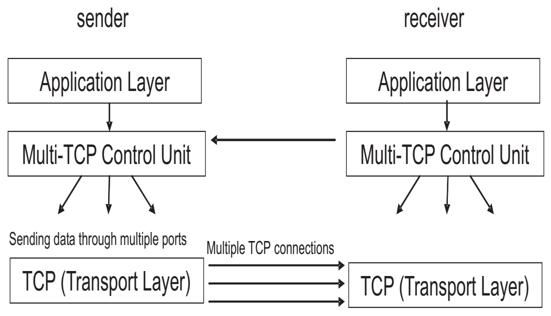


FIGURE 6.1: *MultiTCP* system diagram.

existing TCP stack or kernel. Figure 6.1 shows the proposed *MultiTCP* system. The *MultiTCP* control unit is implemented immediately below the application layer and above the transport layer at both the sender and the receiver ends. The *MultiTCP* control unit at the receiver end receives the input specifications from streaming application which include the streaming rate. The *MultiTCP* control unit at the receiver end measures the actual throughput and uses this information to control the rate precisely by using multiple TCP connections and dynamically changing receiver's window size for each connection. In the next two sections, it is elaborated as to how multiple TCP connections can mitigate the short term throughput reduction problem in a lightly loaded network. The description of the mechanism to maintain the desired throughput in a congested network is also provided.

6.1 Alleviating Throughput Reduction In Lightly Loaded Network

In this section, we analyze the throughput reduction problem in a lightly loaded network and show how it can be alleviated by using multiple TCP connections.

When there is no congestion, the receiver can control the streaming rate in a single TCP connection quite accurately by setting the maximum receiver's window size W_{max}. The effective throughput during this period is approximately equal to

$$T = \frac{W_{max} MTU}{RTT} \tag{6.1}$$

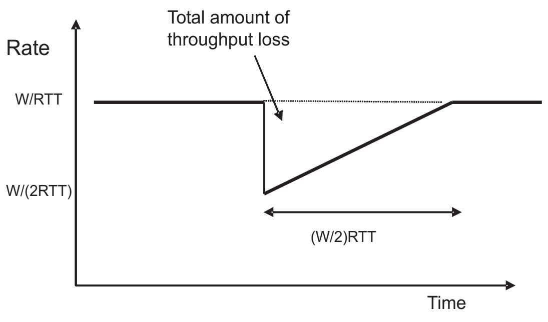
where RTT denotes the round trip time, including both propagation and queuing delay, between the sender and the receiver. MTU denotes the TCP maximum transfer unit, typically set at 1000 bytes. If a loss event occurs, the TCP sender instantly reduces its rate by half as shown in Figure 6.2(a). As a result, the area of the inverted triangular region in Figure 6.2(a) indicates the amount of data that would have been transmitted if there were no loss event. Thus, the amount of data reduction D equals to

$$D = (\frac{1}{2})(\frac{W_{max} MTU RTT}{2})(\frac{W_{max}}{2RTT}) = \frac{W_{max}^2 MTU}{8} \tag{6.2}$$

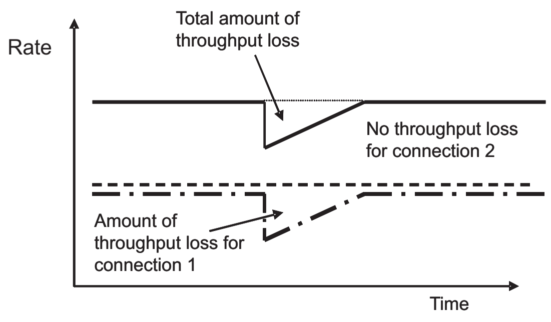
It is worth noticing that the time it takes for the TCP window to increase from $W_{max}/2$ to W_{max} equals to $W_{max} RTT/2$ since the TCP window increases by one every round trip time. Clearly, if there is a burst of loss events during a streaming session, the total throughput reduction can potentially be large enough to deplete the start up buffer, causing pauses in the playback.

Now let us consider the case where two TCP connections are used for the same application. Since we want to keep the same total streaming rate W_{max}/RTT as in the case of one TCP connection, we set $W'_{max} = W_{max}/2$ for each of the two connections as illustrated in Figure 6.2(b). Assuming that only a single loss event happens in one of the connection, the total throughput reduction would be equal to

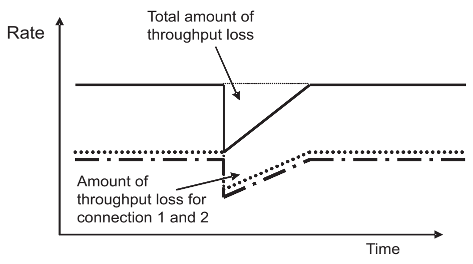
$$D' = \frac{(W'^2_{max} MTU)}{8} = \frac{(W^2_{max} MTU)}{32} = \frac{D}{4} \tag{6.3}$$



(a)



(b)



(c)

FIGURE 6.2: Illustrations of throughput reduction for (a) one TCP connections with single loss; (b) two TCP connections with single loss; (c) two TCP connections with double losses.

Equation (6.3) shows that, for a single loss event, the throughput reduction by using two TCP connections is four times less than the one using a single TCP connection. Even in the case when there are simultaneously losses on both connections as indicated in Figure 6.2(c), the throughput reduction is half of that of the single TCP. In general, let N denote the number of TCP connections for the same application and n be the number of TCP connections that suffer simultaneous losses during short congestion period, the amount of throughput reduction equals to

$$D_N = \frac{nW_{max}^2 MTU}{N^2} \tag{6.4}$$

As seen in Equation (6.4), the amount of throughput reduction is inversely proportional to the square of the number of TCP connections used. Hence, using a only small number of TCP connections can greatly improve the resilience against TCP throughput reduction in lightly loaded network.

6.2 Control Streaming Rate in a Congested Network

In the previous section, we discussed the throughput reduction problem in a lightly loaded network and showed that using multiple TCP connections can alleviate the problem. In a lightly loaded network condition, one can set the desired throughput T_d by simply setting the receiver window $W_{max} = T_d RTT/MTU$. However, in a moderately or heavily congested network, the throughput of a TCP does not depend on W_{max}, instead, it is determined by the degree of congestion. This is due to the fact that in a non-congested network, i.e. without packet loss, TCP rate would increase additively until $W_{max}MTU/RTT$ is reached, after that the rate would remain approximately constant at $W_{max}MTU/RTT$. However, in a congested network, a loss event would most likely occur before the sending rate reaches its limit and cut the rate by half, resulting in a throughput lower than $W_{max}MTU/RTT$.

A straightforward method for achieving a higher throughput than the available TCP throughput would be to use multiple TCP connections for the same application. Using multiple TCP connections results in a larger share of the fair bandwidth. Hence, one may argue that this is unfair to other TCP connections. On the other hand, one can view this approach as a way of providing higher priority for streaming applications over other non time-sensitive applications under resource constraints. We also note that one can use UDP to achieve the desired throughput. However unlike UDP, using multiple TCP connections can provide

1. Congestion control mechanism to avoid congestion collapse, and
2. Automatic retransmission of lost packets.

Assuming multiple TCP connections are used, there are still issues associated with providing the desired throughput in a congested network.

In order to maintain a constant throughput during a congested period, one possible approach is to increase the number of TCP connections until the measured throughput exceeds the desired one. This approach has a few drawbacks.

1. The total throughput may still exceed the desired throughput by a large amount since the sending rate of each additional TCP connection may be too high.
2. If only a small number of TCP connections are required to exceed the desired throughput, this technique may not be resilient to the sudden increase in traffic as analyzed in Section 6.1.

A better approach is to use a larger number of TCP connections but adjust the receiver window size of each connection to precisely control the sending rate. It is undesirable to use too many TCP connections as they use up system resources and may further aggravate an already congested network. In this research, two algorithms are considered:

1. The first(basic) algorithm maintains a fixed number of TCP connections while it varies the size of the receiver windows of each TCP connection to achieve the desired throughput.

2. The second(advanced) algorithm dynamically changes the number of TCP connections based on the network congestion.

These algorithms are discussed in the sections below.

6.3 The Basic Algorithm

The basic algorithm uses a fixed, default number of TCP connections and only varies receiver window size to obtain the desired throughput T_d. Hence, the inputs to the algorithm are the desired user's throughput T_d and the number of TCP connections. Below are the steps of the proposed algorithm.

Initializing steps:

1. Set N, the number of TCP connections to the user input.

2. Set the receiver window size $w_i = \frac{T_d RTT}{(MTU)N}$ for connection i.

Running steps: The actual throughput T_m is measured at every δ seconds and the algorithm dynamically changes the window size based on the measured T_m as follows.

3. If both of the following conditions

   (a) $T_m < T_d$, and

   (b) $W_s = \sum_i w_i \leq \frac{fT_dRTT}{MTU}$ where $f > 2$

   are true, run *AdjustWindow(T_d, T_m)*.

4. If $T_m > T_d + \lambda$, run *AdjustWindow(T_d, T_m)*.

5. Else, keep the receiver window size the same.

We now discuss each step of the algorithm in detail and show how to choose appropriate values for the parameters. In step 1, we empirically found, $N = 5$ works well in many scenarios. If user does not specify the number of TCP connections, the default value is set to $N = 5$. In step 2, we assume that the network is not congested initially, hence the total expected throughput and the total receiver window size W_s would equal to T_d and T_dRTT/MTU respectively. Worth noting is that the average RTT can be obtained easily at the receiver end.

In the running steps, δ should be chosen to be several times the round trip time since the sender cannot respond to the receiver changing window size for at least one propagation delay, or approximately half of RTT. As a result, the receiver may not observe the change in throughput until several RTTs later. In most scenarios, we found that setting the measuring interval $\delta = 8RTT$ works quite well in practice. In step 3, the algorithm tries to increase the throughput by increasing the window size of each connection via the routine *AdjustWindow*. The implementation details of *AdjustWindow* are discussed in Section 6.4. The first condition of step 3 indicates that the measured throughput is still under the desired one. The second condition limits the maximum total receiver window size. Recall that in the congestion state, the average size of the receiver window is $W_{max} = \frac{2T_m RTT}{MTU}$. Hence, increasing w_i beyond this value would not increase the TCP throughput. However, if we let w_i increase without bound, there will be a spike in throughput once the network becomes less congested. To prevent unnecessary throughput fluctuation, our algorithm limits the sum of receiver window size W_s to $f\frac{T_d RTT}{MTU}$ where $f > 2$ is used to control the height of the throughput spike. Larger and smaller values of f result in higher and lower throughput spikes respectively, as discussed later in Chapter 8.

Step 4 of the algorithm is similar to step 3 except the receiver window size now would be reduced, using the same *AdjustWindow* routine. λ in the inequality $T_m > T_d + \lambda$ is a small throughput threshold used to ensure T_m to be approximately equal to T_d, and at the same time, to prevent T_m from going below T_d. Finally, since the measured throughput can be noisy, we use the exponential average measured throughput computed recursively as $T_m = \alpha T_m + (1 - \alpha)T_n$ where T_n is the new throughput sample and $\alpha < 1$ is a smoothing parameter.

6.4 Adjusting the receiver window sizes

We now discuss *AdjustWindow* in detail. In this step, the algorithm increases/decreases the window size w_i for a subset of connections if the measured throughput is smaller/larger than the desired throughput. There exists an optimal way for choosing a subset of connections for changing the window size and the corresponding increments in order to achieve the desired throughput. If the number of chosen connections for changing the window size and the corresponding window increments are small, then the time for achieving the desired throughput may be longer than necessary. On the other hand, choosing too large a number of connections and increments may result in higher throughput than necessary. For example, assuming we have five TCP connections, each with RTT of 100 milliseconds, MTU equals to 1000 bytes, and the network is in non-congestion state, then changing the receiver window size of all the connections by one can result in a total change in throughput of $5(1000)/.1 = 50$ Kbytes per second. In a congested scenario, the change will not be that large. However, one may still want to control the throughput change to a certain granularity. To avoid these drawbacks, the algorithm chooses the number of connections for changing their window size and the corresponding increments based on the current difference between the desired and measured throughput. The pseudo code of the algorithm is shown below.

```
AdjustWindow($T_d$, $T_m$)

1. $D_s = \lceil |T_d - T_m| RTT/MTU \rceil$

2. If $T_d > T_m$

    (a) Sort the connections in the increasing order of $w_i$

                         $w_i := w_i + 1$
    (b) While $D_s > 0$  $D_s := D_s - 1$
                         $i := (i + 1) \mod N$

3. If $T_d < T_m$

    (a) Sort the connections in the decreasing order of $w_i$

                         $w_i := w_i - 1$
    (b) While $D_s > 0$  $D_s := D_s - 1$
                         $i := (i + 1) \mod N$
```

The reasoning behind the algorithm can be understood as follows. Consider the case when $T_m < T_d$. If there is no congestion, setting the sum of window size increments D_s from all the connections to $\lceil (T_d - T_m) RTT/MTU \rceil$ would result in a throughput increment of $T_d - T_m$, hence the desired throughput would be achieved. If there is a congestion, this total throughput increment would be smaller. However, subsequent rounds of window increment would allow the algorithm to reach the desired throughput. This method effectively produces a large or small total window increment at every sampled point based on a large or small difference between the measured and desired throughput, respectively. Steps 2a and 2b in the above algorithm ensure that total throughput increment is equally contributed by all the connections. On the other hand, if only one connection j is responsible for all the throughput, i.e. $w_i = 0$ for $j \neq i$, then we simply have a single connection whose throughput can be substantially reduced in a congested scenario. We note that using this algorithm, w_i's for different connections at any point of time differ from each other, at most, by one. The scenario, where $T_m > T_d$, is similar.

6.5 Advanced Algorithm

Disadvantage of Basic Algorithm is that the number of connections are fixed. Hence, it may not perform well under some congestion scenarios. In this section, we introduce a new algorithm that varies the number of TCP connections dynamically depending upon the congestion in the network. This algorithm is an extension of our basic algorithm which will be described shortly.

We assume that a network is congested when we cannot achieve the desired throughput. The amount of congestion in the network can be estimated by

Congestion ratio. The congestion ratio, which is given by W_s/W_n where $W_s = \sum_i w_i$ and W_n denotes the window size of single connection when there is no congestion, i.e, $W_n = T_d RTT/MTU$. If the congestion ratio is one or less, we say that there is no congestion in the network. But if it increases beyond one, we say that there is congestion in the network. Our approach for increasing and decreasing the number of connections dynamically is as follows. We specify the maximum number of connections that can be used for a particular streaming. Hence, the inputs to this algorithm are the maximum number of connections that can be used and the user's desired throughput T_d. The steps of the proposed algorithm are as follows.

We store the value of congestion ratio for every iteration in *prevcongR*, i.e., previous congestion ratio.

Initialization: $prevcongR =: \frac{W_s}{W_n}$

Running steps:

1. If all the following conditions

    (a) $\frac{W_s}{W_n} < prevcongR$

    (b) Number of connections being used for streaming ≥ 2

    (c) $T_m > T_d$

    are true, then
    $prevcongR =: \frac{W_s}{W_n}$
    Stop one connection
    Set the receiver window size for each connection

2. If all the following conditions

    (a) $\frac{W_s}{W_n} > prevcongR$

   (b) $\frac{W_s}{W_n} < cf$, congestion factor to be discussed shortly

   (c) $T_m < T_d$

   (d) Number of connections being used for streaming $\leq N$

   are true, then
   - $prevcongR =: \frac{W_s}{W_n}$
   - Start a new connection
   - Set the receiver window size for each connection

3. If all the following conditions

   (a) $\frac{W_s}{W_n} \geq cf$

   (b) Number of connections being used for streaming ≥ 2

   (c) If the time taken to receive a packet is greater than X times RTT

   are true, then
   - $prevcongR =: \frac{W_s}{W_n}$
   - Stop one connection
   - Set the receiver window size for each connection

The following are the running steps of the basic algorithm.

We now discuss each step in detail. cf is a special factor called the congestion factor, which is used to determine if the network is slightly congested or severely congested. If the network is slightly congested, by increasing the number of TCP connections, we will be able to achieve the desired throughput. But if the network is severely congested, even increasing the TCP connections will not increase the throughput and doing that would further increase the load on the network. So it would be better if we decrease the number of connections being used for streaming. So if the congestion ratio is equal to or exceeds cf, then we reduce the number of TCP connections. Let us consider step 1. When the previous congestion ratio, $prevcongR$, decreases by one we can say that the congestion in the network reduces. If the number of connections being used for streaming is greater than or equal to two and also if the measured throughput, T_m, is greater than the desired throughput, T_d, we reduce the number of connections being used for streaming. The average window size of all connections are also updated accordingly. This step 1 is used when the congestion in the network reduces.

Now consider step 2. When the previous congestion ratio, *prevcongR*, increases by one, we can say that the congestion in the network increased. Here, if the current congestion ratio, W_s/W_n, is less than the congestion factor, cf, and if the number of connections being used for streaming are fewer than the maximum number of connections that are allowed and also if the measured throughput, T_m, is less than the desired throughput, T_d, we increase the number of connections being used for streaming by one. The average window size of all connections is also set accordingly by the basic algorithm. This step 2 is used when the congestion in the network increases. When the congestion ratio increases by one, we start streaming through an additional connection along with the connections which are already being used. Now let us consider step 3. If the current congestion ratio, W_s/W_n, is greater than or equal to the congestion factor, cf, and if the number of connections being used for streaming is greater than or equal to two and also if the time taken to receive a packet is greater than X times RTT, we decrease the number of connections being used for streaming. The average window size of all connections is set accordingly. This step 3 is used when the congestion in the network increases to a level that even by increasing the number of connections we cannot achieve the desired throughput, T_d. This means that the network is severely congested. So we have to reduce the network congestion by stopping as many connections as possible.

CHAPTER 7

MULTITCP DESIGN

In this chapter we provide the details of the *MultiTCP* design and discuss how the *MultiTCP* system works.

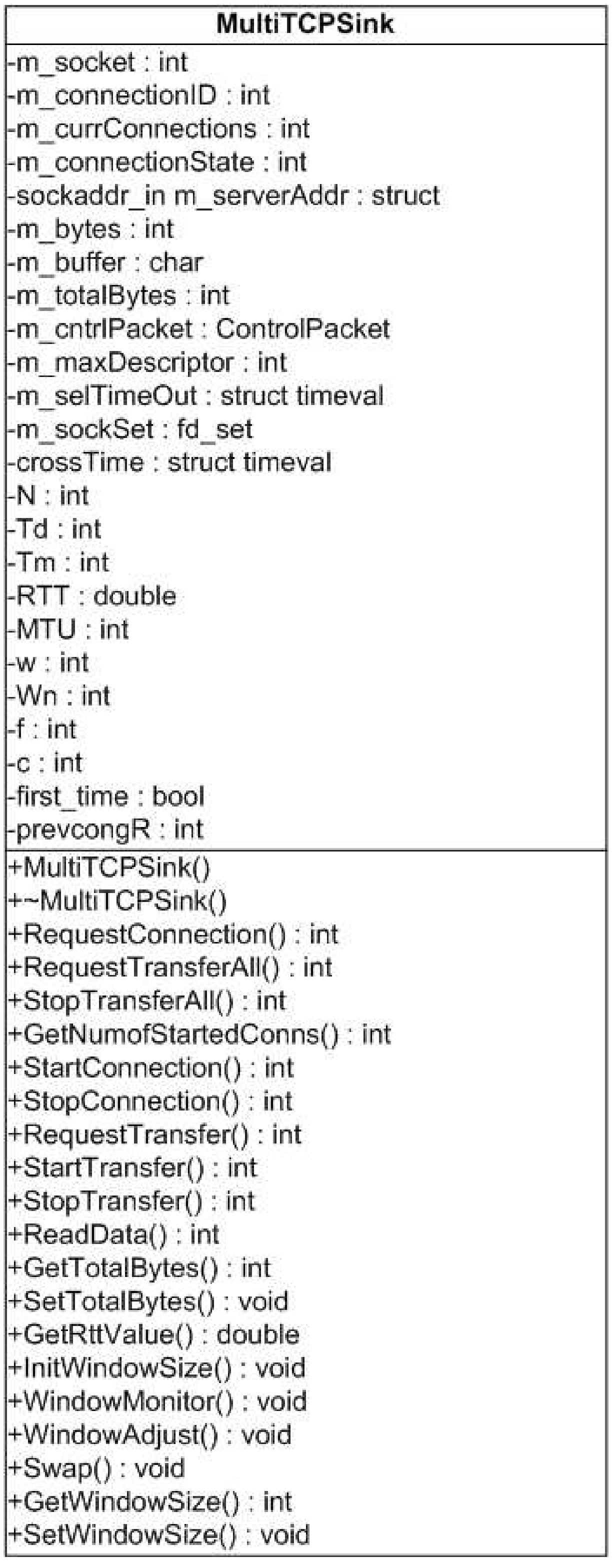
7.1 MultiTCP Design

This *MultiTCP* system is a receiver-driven, TCP based application-layer transmission protocol for multimedia streaming over the Internet. So, the algorithm which is responsible for achieving the desired throughput is implemented on the client side. The class which comprises the main algorithm component of this system is shown in Figure 7.1.

The function that implements the algorithm which controls this system and is responsible for achieving the desired throughput is *WindowMonitor()* which in turn uses the function *WindowAdjust()* to set the window size appropriately depending on the throughput received. The pseudo code for these algorithms is given in Chapter 6.

7.2 MultiTCP Working

This section shows the working of the *MultiTCP* system using a sequence diagram. The sequence diagram can be seen in Figure 7.2.



MultiTCPSink

```
-m_socket : int
-m_connectionID : int
-m_currConnections : int
-m_connectionState : int
-sockaddr_in m_serverAddr : struct
-m_bytes : int
-m_buffer : char
-m_totalBytes : int
-m_cntrlPacket : ControlPacket
-m_maxDescriptor : int
-m_selTimeOut : struct timeval
-m_sockSet : fd_set
-crossTime : struct timeval
-N : int
-Td : int
-Tm : int
-RTT : double
-MTU : int
-w : int
-Wn : int
-f : int
-c : int
-first_time : bool
-prevcongR : int
```

```
+MultiTCPSink()
+~MultiTCPSink()
+RequestConnection() : int
+RequestTransferAll() : int
+StopTransferAll() : int
+GetNumofStartedConns() : int
+StartConnection() : int
+StopConnection() : int
+RequestTransfer() : int
+StartTransfer() : int
+StopTransfer() : int
+ReadData() : int
+GetTotalBytes() : int
+SetTotalBytes() : void
+GetRttValue() : double
+InitWindowSize() : void
+WindowMonitor() : void
+WindowAdjust() : void
+Swap() : void
+GetWindowSize() : int
+SetWindowSize() : void
```

FIGURE 7.1: *MultiTCPSink* Class diagram

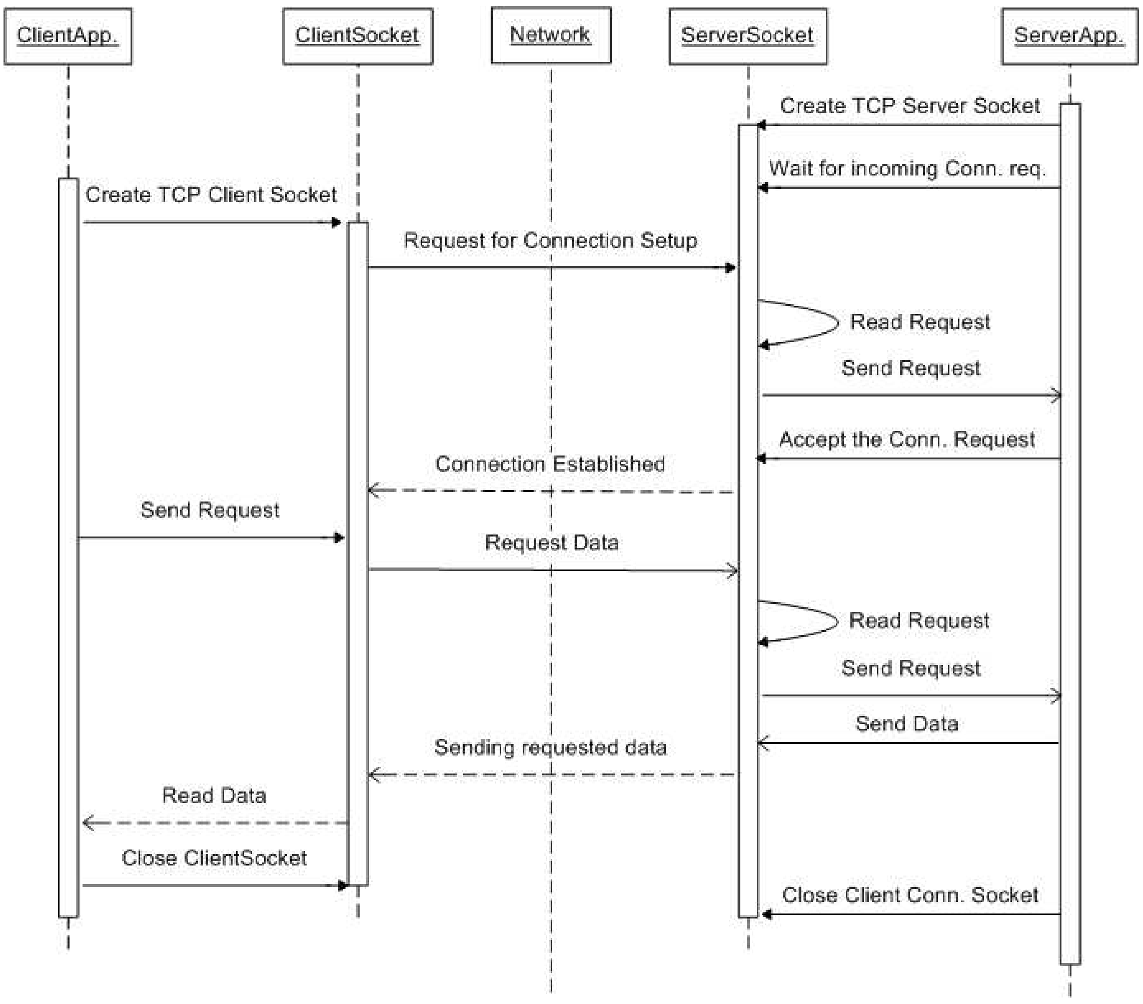


FIGURE 7.2: *Client-Server* Sequence diagram

The interaction between the client and server for establishing connections and transferring data will be the same as seen before in Section 5.2. But while initiating the connection establishment between client and server, the user will specify the number of TCP connections that the system must use.

If we consider the basic algorithm, the user has to specify the desired throughput and the number of TCP connections to be used (which is constant). But if we consider the advanced algorithm, the user has to specify the desired throughput and the maximum number of connections that the system must use (which may vary).

7.3 Remarks on Sender

At the sender, data is divided into packets of equal size. These packets are always sent in order. The *MultiTCP* system chooses the TCP connection to send the next packet in a round robin fashion. If a particular TCP connection is chosen to send the next packet, but it is blocked due to TCP congestion mechanism, the *MultiTCP* system chooses the first available TCP connection in a round robin manner. For example, suppose there are 5 connections, denoted by TCP1 to TCP5. If none of TCP connection is blocked, packet 1 would be sent by TCP1, packet 2 by TCP2, and so on. If TCP1 is blocked, then TCP2 would send packet 1 and TCP3 would send packet 2, and so on. When it is TCP1's turn again and if TCP1 is not blocked, it would send packet 5. This is similar to socket striping technique in [24].

7.4 Object Oriented Features used in Design

The Object Oriented features used in the design of the *MultiTCP* system are listed below.

(a) Object: An Object/Instance is an individual representative of a class. Instances of all the classes that are part of the system are created to perform required functions.

(b) Class: A class is a collection of objects of similar type. Once a class is defined, any number of objects can be created which belong to that class. Example of a class in this system is *MultiTCPSink*.

(c) Behavior and State: The behavior of a component is the set of actions that a component can perform. A Server can handle several client requests. Its state would be described by the set of values of the attributes of a particular object.

(d) Encapsulation: Storing data and functions in a single unit (class) is encapsulation. This helps to club together the data and all the methods that operate on that data.

(e) Constructors: It is a procedure that is invoked implicitly during the creation of a new object value and guarantees that the newly created object is properly initialized. *MultiTCPSink* class of the system has a constructor associated with it.

(f) Destructors: Like a constructor, a destructor is a method invoked implicitly when an object goes out of scope or is destroyed. Therefore, just before the object is reclaimed, all resources held by the object need to be released. *MultiTCPSink* class of the system has a destructor associated with it.

CHAPTER 8

PERFORMANCE EVALUATION

In this chapter, we show simulation results using NS and the results produced using actual network to demonstrate the effectiveness of this *MultiTCP* system in achieving the required throughput as compared to the traditional single TCP approach.

8.1 Results for the Basic Algorithm

8.1.1 NS Simulation

The simulation setup consists of a sender, a receiver, and a traffic generator connected together through a router to form a dumb bell topology as shown in Figure 8.1. The bandwidth and propagation delay of each link in the topology are identical, and are set to 6 Mbps and 20 milliseconds, respectively. The sender streams 800 kbps video to the receiver continuously for a duration of 1000s, while the traffic generator generates cross traffic at different times by sending packets to the receiver using either long term TCPs or short bursts of UDPs. In particular, from time $t = 0$ to $t = 200$s, there is no cross traffic. From $t = 200$s to $t = 220$s and $t = 300$s to $t = 340$s, bursts of UDPs with rate of 5.5 Mbps are generated from the traffic generator node to the receiver. At $t = 500$s the traffic generator opens 15 TCPs connections to the receiver, and 5 additional TCP connections at $t = 750$s. We now consider this setup under

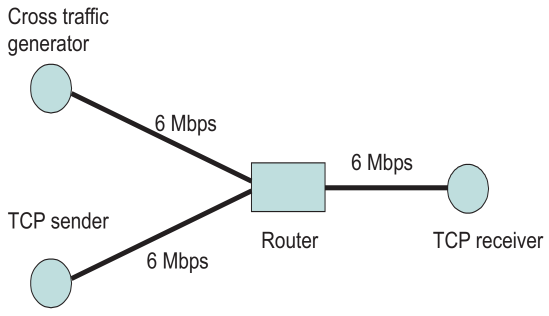


FIGURE 8.1: *Simulation topology.*

three different scenarios: (a) the sender uses only one TCP connection to stream the video, while the receiver sets the receiver window size to 8, targeting at 800 kbps throughput, (b) the sender and the receiver use the *MultiTCP* system to stream the video with the number TCP connections limited to two, and (c) the sender and the receiver also use the proposed *MultiTCP* system, except the number of TCP connections are now set to five. Table 8.1 shows the parameters used in the *MultiTCP* system.

Sampling interval δ	300 ms
Throughput smoothing factor α	0.9
Guarding threshold λ	7000 bytes
Throughput spike factor f	6

TABLE 8.1: *Parameters used in MultiTCP system*

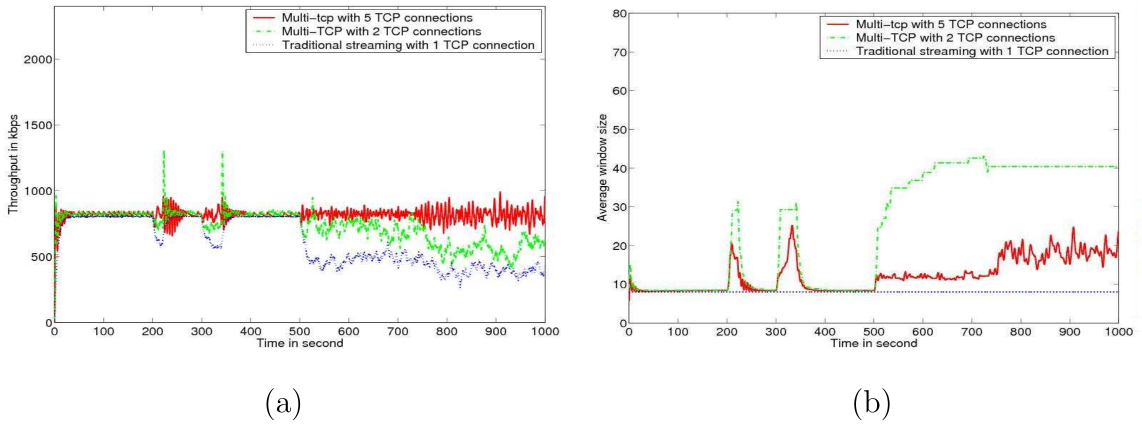


FIGURE 8.2: (a) Resulted throughput and (b) average receiver window size when using 1, 2 and 5 TCP connections.

Figure 8.2(a) shows the throughput of three described scenarios. As seen, initially without congestion, using the traditional single TCP connection can control the throughput very well since setting the size of the receiver window to 8 achieves the desired throughput. However, when traffic bursts occur during the intervals $t = 200s$ to $t = 220s$ and $t = 300s$ to $t = 340s$, the throughput of using a single TCP connection reduces substantially to only about 600 kbps. For the same congested period, using two TCP connections results in higher throughput, approximately 730 kbps. On the other hand, using five TCP connections produces approximately the desired throughput, demonstrating that a larger number of TCP connections results in higher throughput resilience in the presence of misbehaved traffic such as UDP flows. These results agree with the analysis in Section 6.1. It is interesting to note that when using two TCP connections, there are spikes in the throughput immediately after the network is no longer congested at t = 221s and t = 341s. This phenomenon relates to the maximum receiver window size set during the congestion period. Recall that the algorithm keeps increasing the w_i until either (a) the measured throughput exceeds the desired throughput or (b) the sum of receiver window size $W_s = \sum_i w_i$ reaches $f\frac{T_d RTT}{MTU}$. In the simulation, using two TCP connections never achieves the desired throughput during the congested periods, hence the algorithm keeps increasing the w_i. When network is no longer congested, the W_s already accumulates to a large value. This causes the sender to send a large amount of data until the receiver reduces the window size to the correct value a few RTTs later. On the other hand, when using 5 TCP connections, the algorithm achieves the desired throughput during the congestion periods, as such W_s does not increase to a large value, resulting in a smaller throughput spike after the congestion vanishes.

Next, when 15 cross traffic TCP connections start at $t = 500$s, the resulting throughput when using one and two TCP connections reduce to 700 kbps and 350 kbps, respectively. However, throughput when using 5 TCP connections stays approximately constant at 800 kbps. At $t = 750$s, 5 additional TCP connections start, throughput are further reduced for the one and two connection cases, but it remains constant for the five-connection case.

Figure 8.2(b) shows the average of the sum of window size W_s as a function of time. As seen, W_s increases and decreases appropriately to respond to network conditions. Note that using two connections, W_s increases to a larger value than when using 5 TCP connections during the intervals of active UDP traffic. This results in throughput spikes discussed earlier. Also, the average window size in the interval $t = 500$s to $t = 750$s is smaller than that of the interval $t = 750$s to $t = 1000$s, indicating that the algorithm responds appropriately by increasing the window size under a heavier load.

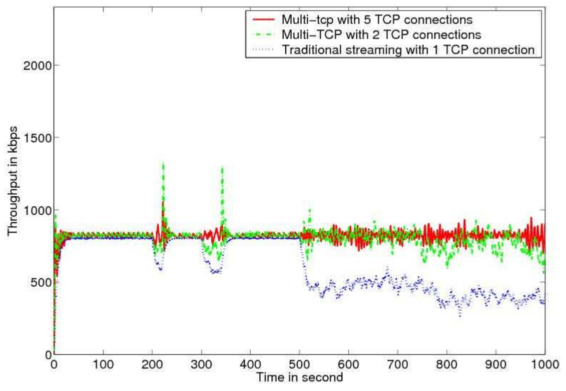


FIGURE 8.3: Resulted throughput when using 1, 2 and 5 TCP connections with cross traffic having larger RTT than that of video traffic.

We now show the results when the cross traffic has different round trip time from that of the video traffic. In particular, the propagation delay between the router and traffic generator node is now set to 40 milliseconds. All cross traffic patterns stay the same as before. The new simulation shows the same basic results. As seen in Figure 8.1.1, the throughput of 5 connections is still higher than that of two connections which, in turn is higher than that of one connection during the congestion periods. The throughput of two connections in this new scenario is slightly higher than that of the previous scenario during the congested period from $t = 500$s onward. This is due to a well known phenomena where TCP connection with shorter round trip times gets a larger share of bandwidth for the same loss rate. Since the round trip time of the video traffic is now shorter than that of the TCP cross traffic, using only two connections, the desired throughput of 800 kbps can be approximately achieved during the period from $t = 500$s to $t = 750$s, which is not achievable in previous scenario. So clearly, the number of connections to achieve the desired throughput depends on the competing traffic.

8.1.2 Internet Experiments

We now show the results from actual Internet experiments using Planet-Lab nodes [25]. In this experiment our setup consisted of a sender, planetlab1.netlab.-uky.edu at University of Kentucky, and a receiver, planetlab2.een.orst.edu at Oregon State University. The sender streams the video to the receiver continuously for a duration of nearly 1000s, while the FTP connections generate cross traffic at the client at different times. From time $t = 0$ to $t = 400$s, there is no cross traffic. From around $t = 401$s to $t = 600$s cross traffic is generated using two FTP connections and after $t = 601$s two additional FTP connections are opened for generating more traffic at the receiver. We now consider performances under three different scenarios: (a) only one TCP connection is used to stream the video, while the receiver sets the receiver window size targeting at 480 kbps throughput, and (b) the sender and the receiver use the proposed MultiTCP system, using the basic algorithm with the number of TCP connections set to four (c) the sender and the receiver use the proposed MultiTCP system, using the basic algorithm with the number of TCP connections now set to eight. Table 8.2 shows the parameters used in our MultiTCP system.

Figure 8.4(a) shows the throughput of three described scenarios. As seen, initially without congestion, using the traditional single TCP connection can very well control the throughput since setting the size of the receiver window achieves the desired throughput. So using either one, four or eight connections does not make any difference during the interval $t = 0s$ to $t = 400s$ as there is no cross traffic. However, when traffic bursts occur during the interval $t = 401s$

Sampling interval δ	6*RTT ms
Throughput smoothing factor α	0.9
Guarding threshold λ	3000 bytes
Throughput spike factor f	10

TABLE 8.2: *Parameters used in MultiTCP system*

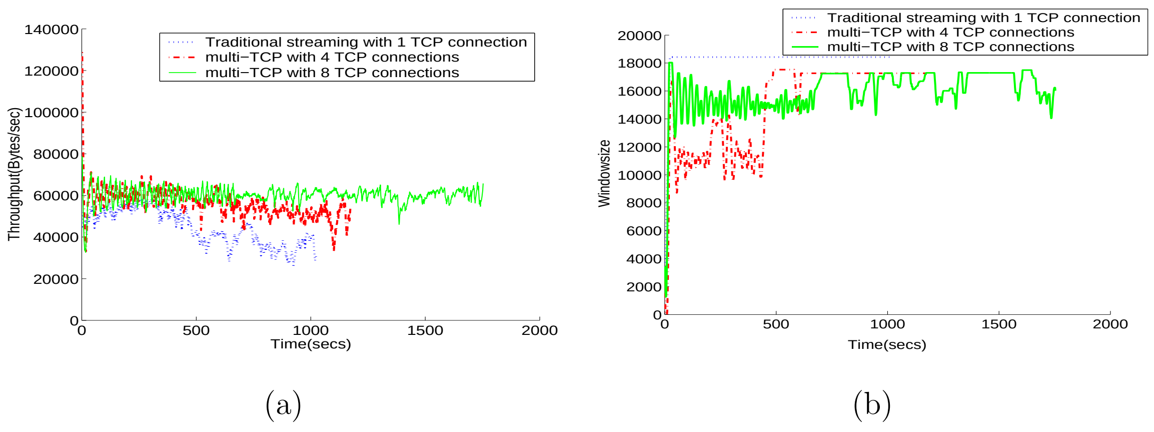


FIGURE 8.4: (a) Resulted throughput, (b) receiver window size when using 1, 4 and 8 TCP connections with cross traffic.

to $t = 600s$ and after $t = 601s$, the throughput of using a single TCP connection reduces substantially from 480 kbps to 280 kbps. Using four TCP connections we were able to achieve the desired throughput during the interval $t = 401s$ to $t = 600s$. But after $t = 601$, the throughput reduced from 480 kbps to 400kbps. For the same congested period, using eight TCP connections results in higher throughput, which is the desired throughput.

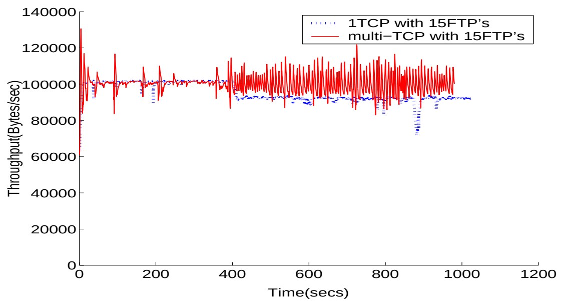
Figure 8.4(b) shows the average of the sum of window size W_s as a function of time. As seen, W_s increases and decreases appropriately to respond to network conditions. Note that using single connection, W_s increases to a larger value than when using multiple TCP connections during the intervals of active FTP traffic. This results in throughput spikes discussed earlier. Also, the average window size in the interval t = 300s to t = 600s is smaller than that of the interval t = 601s to t = 1000s for multiple TCP connections, indicating that the algorithm responds appropriately by increasing the window size under a heavier load.

8.2 Results for the Advanced Algorithm

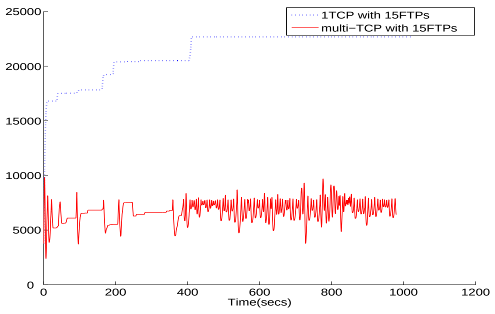
8.2.1 Internet Experiments

We now show the results of our system using our modified algorithm for dynamic change in number of connections. For this experiment our setup consisted of a sender, a receiver, and fifteen FTP connections to generate cross traffic. The machine used as server is planetlab1.csres.utexas.edu which is located at University of Texas. The setup under two different scenarios is as follows: (a) the sender uses only one TCP connection to stream the video, while the receiver sets the receiver window size targeting at 800 kbps throughput, and (b) the sender and receiver use our MultiTCP system to stream the video with the maximum number of TCP connections that can be used for streaming set to five. Table 8.3 shows the parameters used in our MultiTCP system.

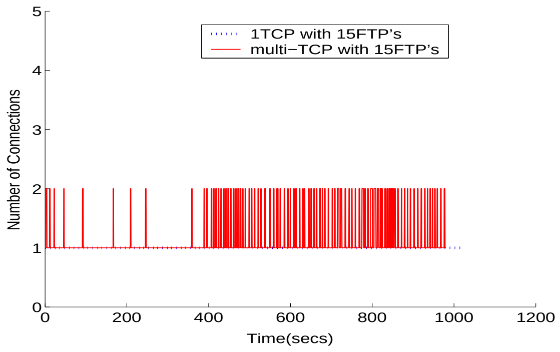
Figure 8.5(a) shows the throughput of two described scenarios with cross traffic. During the interval $t = 0s$ to $t = 400s$ there is no cross traffic. However, when traffic bursts occur due to six FTP connections during the interval $t = 401s$ to $t = 600s$ and nine more FTP's after $t = 601s$, the throughput obtained



(a)



(b)



(c)

FIGURE 8.5: (a) Resulted throughput, (b) receiver window size when using 1 and 5 TCP connections with cross traffic (c) Number of connections used for streaming.

Sampling interval δ	6*RTT ms
Throughput smoothing factor α	0 - 0.9
Guarding threshold λ	3000 bytes
Throughput spike factor f	6
Congestion factor cf	5

TABLE 8.3: *Parameters used in MultiTCP system*

using a single TCP connection reduces from 800 kbps to 720 kbps. For the same congested period, using the maximum number of connections as five TCP connections, even though it used only two TCP connections, resulted in higher throughput which is the desired throughput. These results demonstrate that a larger number of TCP connections results in higher throughput resilience in the presence of misbehaved traffic.

Figure 8.5(b) shows the average of the sum of window size W_s as a function of time. As explained before, W_s increases and decreases appropriately to respond to network conditions.

Figure 8.5(c) shows how the number of connections vary depending upon the congestion ratio for that particular scenario. When the window size increased the number of connections used for streaming also increased. This implies that when the congestion in the network increases and if we are not able to receive the desired throughput, the number of connections to be used for streaming are increasing. These results agree with the analysis in Section 6.1.

Now let us consider another experiment. For this experiment our setup consists of a sender, a receiver, and eight FTP connections to generate cross

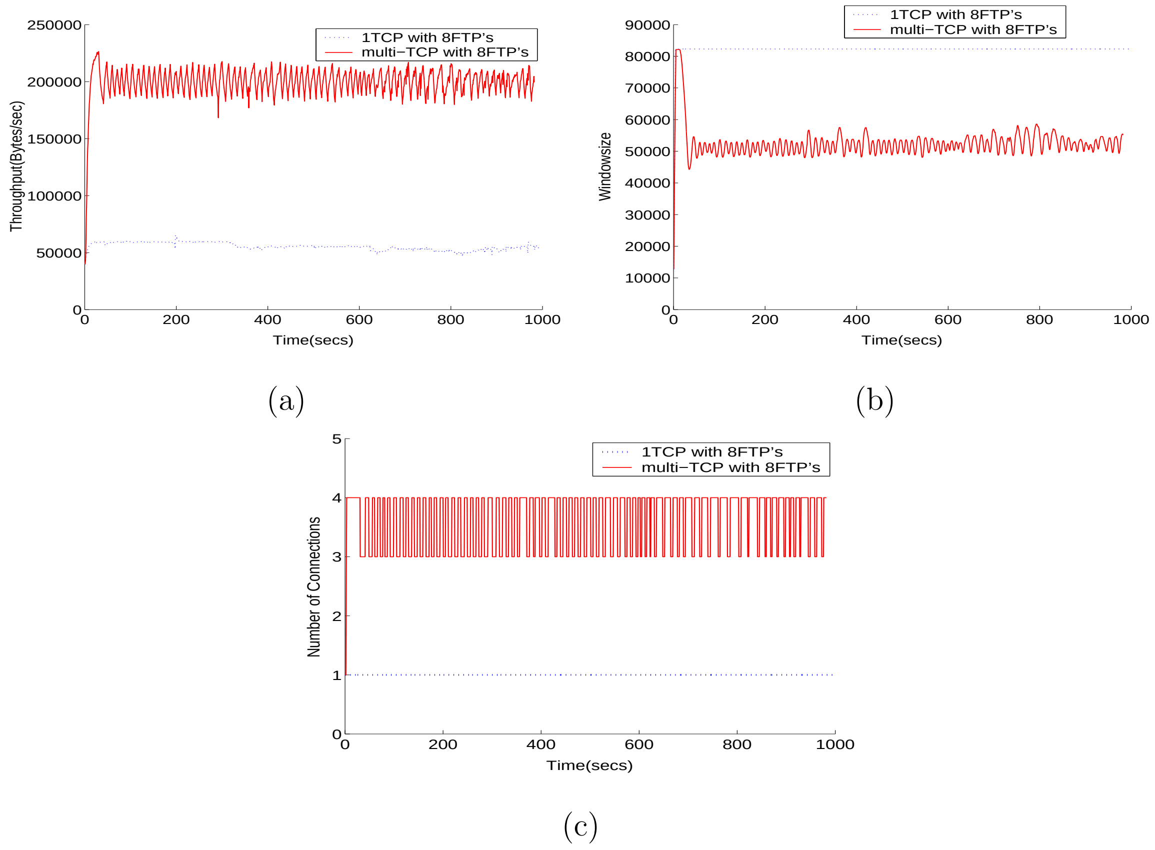


FIGURE 8.6: (a) Resulted throughput, (b) receiver window size when using 1 and 5 TCP connections with cross traffic (c) Number of connections used for streaming.

traffic. The machine used as server is planetlab1.cs.pitt.edu which is located at University of Pittsburgh. The setup under two different scenarios is as follows: (a) the sender uses only one TCP connection to stream the video, while the receiver sets the receiver window size targeting at 1.6 Mbps throughput, and (b) the sender and receiver use our MultiTCP system to stream the video with the maximum number of TCP connections that can be used for streaming is set to five. We used the same parameters as in the previous experiment.

Figure 8.6(a) shows the throughput of two described scenarios with cross traffic. During the interval $t = 0s$ to $t = 300s$ there is no cross traffic. Even though there is no traffic during that period, as the socket buffer size for each socket supports only 480 kbps, the client was able to receive only 480 kbps. The socket buffer size might vary from system to system. As the desired throughput is 1.6 Mbps, using the maximum number of connections as five, the throughput increased to 1.6 Mbps and stabilized at that throughput. However, when traffic bursts occur due to four FTP connections during the interval $t = 301s$ to $t = 600s$ and four more FTP's after $t = 601s$, the throughput of using a single TCP connection reduces further from 480 kbps to 400 kbps using single TCP connection. For the same congested periods, using the maximum number of connections as five TCP connections, even though it never used all the five TCP connections, resulted in higher throughput which is the desired throughput. These results demonstrate that a larger number of TCP connections results in higher throughput resilience in the presence of misbehaved traffic. Figure 8.6(c) shows how the number of connections vary depending upon the congestion ratio for that particular scenario.

Figure 8.6(b) shows the average of the sum of window size W_s as a function of time. As explained before, W_s increases and decreases appropriately to respond to network conditions. But initially, as it has to get to a high throughput, the window size went up to maximum and then got adjusted slowly.

Figure 8.6(c) shows how the number of connections vary depending upon the congestion in the network as discussed before.

These results also agree with the analysis in Section 6.1. Recall that the algorithm keeps increasing the w_i until either (a) the measured throughput exceeds the desired throughput or (b) the sum of receiver window size $W_s = \sum_i w_i$

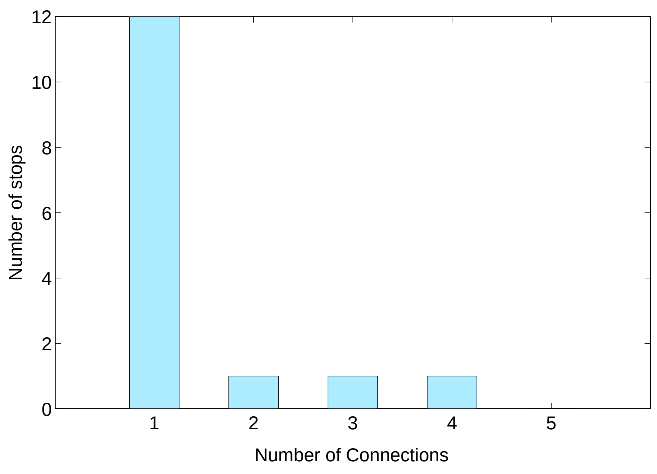


FIGURE 8.7: Resulted number of stops when using 1, 2, 3, 4 and 5 TCP connections with cross traffic having same RTT as that of video traffic.

reaches $f\frac{T_d RTT}{MTU}$. In the results we have shown, using single TCP connection never achieves the desired throughput during the congested periods, hence the algorithm keeps increasing the w_i and reaches maximum. When network is no longer congested, the W_s already accumulates to a large value. This causes the sender to send a large amount of data until the receiver reduces the window size to the correct value a few RTTs later. On the other hand, when using eight or five TCP connections, the algorithm achieves the desired throughput during the congestion periods, as such W_s does not increase to a large value, which results in a smaller throughput spike after the congestion vanishes.

8.3 Buffering

In this section we show the results of buffering using our MultiTCP system. The Figure 8.7, shows the number of connections on the x-axis and the number of stops on the y-axis. The video is streamed for a duration of 200 seconds using our basic algorithm. The required throughput for streaming the video without stopping is 720 kbps. The setup for this is as follows. Initially we will have 10 seconds for buffering. Once it starts playing the video, for every second we will have to receive 720kbits to play the video without stopping. If the total number of bits received till that particular second is greater than or equal to the total number of bits required, the streaming will continue without any stops. If not then the video will stop for 5 seconds to buffer the data and starts streaming again. For this experiment using our basic algorithm, there are 12 stops when using single TCP connection, 1 stop for each of the 2, 3, 4 TCP connections and no stops when 5 TCP connections are used. But when we used the advanced algorithm, there were no stops since it can achieve the desired throughput automatically by increasing and decreasing the number of connections.

These results demonstrate that our algorithm is able to achieve the desired throughput and maintain precise rate control under varying congestion scenarios with competing UDP and TCP traffic for simulation results and FTP traffic for actual results. It is to reemphasize that, the applications based on our system indeed obtain a larger share of the fair bandwidth. However, we believe that under limited network resources, time-sensitive applications like multimedia streaming should be treated preferentially as long as the performance of all other applications do not degrade significantly. Since our system uses TCP, congestion collapse is not likely to happen as in the case of using UDP when network is highly congested. In fact, DiffServ architecture uses the same principle by providing preferential treatment to high priority packets.

CHAPTER 9

RELATED WORK

There has been previous work on using multiple network connections to transfer data. For example, *path diversity* multimedia streaming framework [10][11][9] provide multiple connections on different path for the same application. These work focus on either efficient source or channel coding techniques in conjunction with sending packets over multiple approximately independent paths. On the other hand, our work aims to increase and maintain the available throughput using multiple TCP connections on a single path. There is also a related work using multiple connections on a single path to improve throughput of a wired-to-wireless streaming video session [15][16]. This work focuses on obtaining maximum possible throughput and is based on TFRC rather than TCP. On the other hand, our work focuses on eliminating *short term* throughput reduction of TCP due to burst traffic and providing precise rate control for the application. As such, the analysis and rate control mechanism in our paper are different from those of [15]. Another related work is Streaming Control Transmission Protocol (SCTP)[17], designed to transport PSTN signaling messages over IP networks. SCTP allows user's messages to be delivered within multiple streams, but it is not clear how it can achieve the desired throughput in a congestion scenario. In addition, SCTP is a completely new protocol, as such the kernel of the end systems need to be modified. There are also other work related to controlling TCP bandwidth. For example, the work in [18][19] focuses on allocating bandwidth among flows with different priorities. This work assumes that the bottleneck is at the *last-mile* and that the required throughput for the desired application is achievable using a single TCP connection. On the other hand, our work does not assume the *last-mile* bottleneck, and also the proposed *MultiTCP* system can achieve the desired throughput in a variety of scenarios. Further, the authors in [20], use weighted proportional fair sharing web flows to provide end-to-end differentiated services. The work in [21] uses the receiver advertised window to limit the TCP video bandwidth in VPN link between video and proxy servers. The authors in [22] proposed an approach that leads to real-time applications that are responsive to network congestion, sharing the network resources fairly with other TCP applications. Finally, the authors in [23] propose a technique for automatic tuning of receiver window size in order to increase the throughput of TCP.

CHAPTER 10

CONCLUSION

This thesis is concluded with a summary of contributions. First, we proposed and implemented a receiver-driven, TCP-based application-layer transmission protocol for multimedia streaming over the Internet using multiple TCP connections. Second, our proposed system is able to provide resilience against short-term insufficient bandwidth due to traffic bursts. Third, the proposed system enables the application to control the sending rate in a congested scenario, by using multiple TCP connections and dynamically changing the receiver's window size for each connection, which cannot be achieved using traditional TCP. Finally, the proposed system is implemented at the application layer, and hence, no kernel modification to TCP is necessary. The simulation and experimental results using PlanetLab machines demonstrate that using this proposed system, the application can achieve the desired throughput in many scenarios, which cannot be achieved by traditional single TCP approach.

BIBLIOGRAPHY

[1] MovieFlix, http://www.movieflix.com/.

[2] W. Tan and A. Zakhor, "Real-time internet video using error resilient scalable compression and tcp-friendly transport protocol," *IEEE Transactions on Multimedia*, vol. 1, pp. 172–186, june 1999.

[3] G. De Los Reyes, A. Reibman, S. Chang, and J. Chuang, "Error-resilient transcoding for video over wireless channels," *IEEE Transactions on Multimedia*, vol. 18, pp. 1063–1074, june 2000.

[4] A. Reibman, "Optimizing multiple description video coders in a packet loss environment," in *Packet Video Workshop*, April 2002.

[5] H. Ma and M. El Zarki, "Broadcast/multicast mpeg-2 video over wireless channels using header redundancy fec strategies," in *Proceedings of The International Society for Optical Engineering (SPIE)*, November 1998, vol. 3528, pp. 69–80.

[6] S. Blake, D. Black, M. Carson, E. Davis, Z. Wang, and W. Weiss, "An architecture for differentiated services," in *RFC2475*, December 1998.

[7] Z. Wang, *Internet QoS, Architecture and Mechanism for Quality of Service*, Morgan Kaufmann Publishers, 2001.

[8] P. White, "Rsvp and integrated services in the internet: A tutorial," *IEEE Communication Magazine*, pp. 100–106, May 1997.

[9] T. Nguyen and A. Zakhor, "Multiple sender distributed video streaming," *IEEETransactions on Multimedia and Networking*, vol. 6, no. 2, pp. 315–326, April 2004.

[10] J. Apostolopoulos, "Reliable video communication over lossy packet networks using multiple state encoding and path diversity," in *Proceeding of The International Society for Optical Engineering (SPIE)*, January 2001, vol. 4310, pp. 392–409.

[11] J. Apostolopoulos, "On multiple description streaming with content delivery networks," in *InfoComm*, June 2002, vol. 4310.

[12] S. Floyd, M. Handley, J. Padhye, and J. Widmer, "Equation-based congestion control for unicast application," in *Architectures and Protocols for Computer Communication*, October 2000, pp. 43–56.

[13] T. Nguyen and S. Cheung, "Multimedia streaming using multiple tcp connections," in *IPCCC*, April 2005.

[14] Information Sciences Institute, http://www.isi.edu/nsnam/ns, *Network simulator*.

[15] M. Chen and A. Zakhor, "Rate control for streaming over wireless," in *INFOCOM*, July 2004.

[16] M. Chen and A. Zakhor, "Rate control for streaming video over wireless," *IEEE Wireless Communications*, vol. 12, no. 4, August 2005.

[17] Internet Engineering Task Force, RFC 1771, *Stream Control Transmission Protocol*, October 2000.

[18] P. Mehra and A. Zakhor, "Receiver-driven bandwidth sharing for tcp," in *INFOCOM*, San Francisco, April 2003.

[19] C.De Vleeschouwer P.Mehra and A.Zakhor, "Receiver-driven bandwidth sharing for tcp and its application to video streaming," *IEEE Transactions on Multimedia*, vol. 7, no. 4, August 2005.

[20] J. Crowcroft and P.Oeschlin, "Differentiated end-to-end internet services using weighted proportional fair sharing tcp," 1998.

[21] Y. Dong, R. Rohit, and Z. Zhang, "A practical technique for supporting controlled quality assurance in video streaming across the internet," in *Packet Video*, 2002.

[22] Y.J. Liang, E. Setton, and B. Girod, "Channel adaptive video streaming using packet path diversity and rate-distortion optimized reference picture selection," in *IEEE Fifth Workshop on Multimedia Signal Processing*, December 2002.

[23] J. Semke, J. Mahdavi, and M. Mathis, "Automatic tcp buffer tuning," in *SIGCOMM*, 1998.

[24] J. Leigh, O. Yu and D. Schonfeld, and R. Ansari, "Adaptive networking for tele-immersion," in *Immersive Projection Techonology/Eurographics Virtual Environments Workshop(IPT/EGVE)*, May 2001.

[25] PlanetLab, http://www.planet-lab.org.

[26] Wikipedia, http://www.wikipedia.org.

[27] Computer Networking: A Top-Down Approach Featuring the Internet by James F. Kurose and Keith W. Ross.

[28] Computer Networks by Andrew S. Tanenbaum.