View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by NORA - Norwegian Open Research Archives

UNIVERSITY OF OSLO
Department of Informatics

Improving latency for interactive, thin-stream applications by multiplexing streams over TCP

Master thesis

Chris Carlmar

February 7, 2011

Improving latency for interactive, thin-stream applications by multiplexing streams over TCP

Chris Carlmar

February 7, 2011

Abstract

Many applications use TCP on the Internet today. For applications that produce data all the time, loss is handled satisfactorily. But, for interactive applications, with low rate of data production, the loss of a single packet can mean huge delays.

We have implemented and tested a system to reduce the latency of an interactive TCP application server with many clients. This system multiplexes the streams, to clients in the same region, through a regional proxy, which then sends the streams to their destination. This increases the chance of triggering the TCP mechanism fast retransmit, when a packet is lost, thus reducing the latency caused by retransmissions.

Acknowledgements

I would like to thank my supervisors, Pål Halvorsen and Casten Griwodz, for their guidance and feedback. This thesis would not have been possible without their help.

I would also thank all the guys at the ND Lab at Simula Research Laboratory, for their inspiration and friendly talks. And a special thanks to Brendan Johan Lee, who helped with proof reading, and to Ståle Kristoffersen, who always was willing sanity check my code when something was not work correctly.

And finally, thanks to my wife, Anette, for being patient and supporting me through this entire thesis.

Contents

Abstract i

Acknowledgements iii

Contents v

List of Figures ix

List of Tables xi

1 Introduction 1

1.1 Background and motivation 1

1.2 Problem Statement 2

1.3 Main Contributions 3

1.4 Outline 3

2 Interactive applications 5

2.1 Properties and requirements of interactive applications 6

2.1.1 Games 7

2.1.2 Remote systems 8

2.1.3 Voice over IP 9

2.2 Summary 10

3 Transport **11**

3.1 Choosing a transport protocol 12

3.2 TCP 12

3.2.1 Flow control 14

3.2.2 TCP congestion control mechanisms 14

3.2.3 Retransmission timeout calculation 16

3.2.4 Nagle's algorithm 17

3.3 Thin streams 18

3.3.1 How do TCP's mechanisms affect thin streams? 18

3.4 Summary 19

4 Design **21**

4.1 Bundling of streams 22

4.2 Assumptions and abstractions 23

4.3 System design 25

4.3.1 Step 1 26

4.3.2 Step 2 26

4.3.3 Step 3 27

4.3.4 Step 4 27

4.3.5 Step 5 27

4.3.6 Step 6 27

4.4 System overview 27

4.5 Summary 28

5 Implementation, experiments and analysis **31**

5.1 First prototype 32

5.1.1 Multiplex server implementation 32

5.1.2 Multiplex proxy implementation 32

5.2 Test environment 33

5.2.1 Tools and techniques 33

5.2.2 Testbed 33

5.2.3 Test parameters 34

5.2.4 Measuring delay 34

5.2.5 Understanding the graphs 35

5.3 First tests and conclusions 36

5.4 Second prototype 38

5.5 Second tests and important observations 38

5.5.1 Imposed delays 45

5.5.2 Observations 46

5.6 Parallel connections prototype 46

5.7 Parallel connections tests 47

5.8 Summary 48

6 Conclusion **51**

6.1 Summary 52

6.2 Contributions 52

6.3 Future work 53

Bibliography **55**

A List of abbreviations 59

List of Figures

2.1 Total active subscriptions in MMO games [3] 7

3.1 TCP header 13

3.2 An example of AIMD, slow start and fast recovery 15

3.3 An example of packet transmission with and without Nagle's algorithm when there is unacknowledged data on the connection. 18

4.1 Example of retransmission with and without multiplexing 22

4.2 Multiplexing 24

4.3 A breakdown of each step in our system 25

4.4 System overview 29

5.1 An example boxplot. 35

5.2 Comparison of Baseline and TCPlex tests with 100 ms delay 37

5.3 Comparison of Baseline and TCPlex tests with 300 ms delay 37

5.4 Travel times between the server and proxy 39

5.5 Travel times between the proxy and client 39

5.6 Time it takes from a packet is captured and to it is sent 40

5.7 Comparison of Baseline and TCPlex2 tests with 100 ms delay 41

5.8 Comparison of Baseline and TCPlex2 tests with 300 ms delay 41

5.9 Time data for establishing new client connections, gathered from profiling the proxy 42

5.10 Time data for sending data, gathered from profiling the proxy 42

5.11 Time data gathered from profiling the server libpcap buffer delay 43

5.12 Time data gathered from profiling the server send delay 44

5.13 Zoomed out version of figure 5.12 44

5.14 Delays found in our system. 45

5.15 Parallel connections server 46

5.16 Comparison of Baseline and TCPlex3 tests with 100 ms delay 49

5.17 Comparison of Baseline and TCPlex3 tests with 300 ms delay 49

List of Tables

2.1 Analysis of packet traces from thin and **greedy** streams [26]. 6

Chapter 1

Introduction

1.1 Background and motivation

The Internet has the last 30 year been about bandwidth and capacity, and thus the early network models from this era were focused on fair sharing of resources. We have seen great leaps in networking technology since these early days of the Internet, and now, we have greatly improved bandwidth capacity. This rise in capacity has been followed by a trend to consume more bandwidth.

At the same time as we had this inclination to consume more bandwidth, applications needing real-time communication evolved, and today, the Internet is used for a wide range of interactive services. This has led to latency requirements; if a service has too high latency, the users do not feel it is interactive and may also suffer from bad quality. To get a high-quality experience in games, the response time for the user should be between 100 ms and 1000 ms, depending on the type of game [14]. For Voice over IP (VoIP), the International Telecommunication Union (ITU-T) recommends an end-to-end delay of 150 ms, and a maximum delay of 400 ms [19]. A solution to this problem was to try reservation in the network, but it was not generally accepted. These interactive applications usually generate a very specific traffic pattern, they have small packet sizes and large Interarrival Times (IATs) (i.e., a low packet rate). This traffic pattern is something we define as *thin streams*.

Currently, the most common end-to-end transport protocols are the Transport Control Protocol (TCP) [28] and User Datagram Protocol (UDP) [27]. There are also protocols under development, that try to add more versatility and functionality, like the Stream Control Transmission

Protocol (SCTP) [29], but there is no widespread support for these new protocols for the end-user. Thus, TCP is the most viable choice for applications that need reliable, in-order data delivery. TCP also provides congestion- and flow control, enabling the sharing of network capacity and preventing the overwhelming of the receiver. UDP does not provide any of these services, but allows the sending application to determine the transmission rate. This makes UDP suited for latency sensitive applications that do not need reliability. However, many interactive applications require reliability, forcing them to either use TCP, or implement reliability on the application layer. Still, because of it's lack of congestion control, some Internet Service Providers (ISPs) block UDP in their firewalls. Thus, many time-dependent interactive applications use TCP as the main, or fall-back, transport protocol.

Since the focus in TCP is on achieving high throughput, the mechanisms responsible for recovering after loss assume that the sending application supplies a steady stream of data. This is not the case for interactive applications, thus they suffer high delays since TCP can use up to several seconds before recovering after a packet loss in a low rate stream. Therefore, the focus in this thesis is to enable some interactive applications to recover after packet loss, without adding a high latency. By combining TCP streams that share a path through the network, we aim to reduce the latency by making the stream behave as an ordinary TCP stream over the unreliable path in the network, so that TCP's mechanisms works at peek efficiency.

1.2 Problem Statement

Traffic generated by interactive applications is treated badly by TCP congestion control. Interactive applications generally generate so little data that they are unable to trigger mechanisms like fast retransmit. Other TCP mechanisms like exponential backoff also hurt interactive applications performance, since they send data so rarely that the timeouts can become quite large.

One proposed solution to this problem, for multi-user applications, is to multiplex several streams into one stream [16, 22]. This system should reduce the latency of an interactive application, by multiplexing its many thin streams into one thicker stream in cases where the thin streams all share a path through the network. This raises the probability of triggering a fast retransmit, and thus lowers the number of times the exponential backoff is triggered. In this thesis, we try to implement this solution transparently, so it can be used without modifying the sender application.

This solution to the latency problem is competing with the solution proposed by Andreas Petlund in his PhD thesis [26]. His solution is to modify TCP itself on the server side to treat interactive applications fairly.

1.3 Main Contributions

In this thesis, we explore a solution to the latency problems that arise when using TCP with interactive thin stream applications, specifically in online gaming.

We create a system that multiplex many thin streams, over one or more TCP connections, to a proxy which then demultiplexes the stream(s) and sends the original streams to their destination. This helps to raise to probability of TCP treating the streams fairly, by triggering fast retransmit more often and reducing the retransmission delay.

We evaluate tests run with and without this system, and we break down the delays added in each step of our system. These results are then used to create new and better prototypes, and we compare the end-to-end delay of the different prototypes to the results from the baseline tests.

The end result is a system that can be used to reduces the maximum delays of a multi-user, interactive, thin-stream application in high loss scenarios, at the cost of a higher average delay time.

1.4 Outline

In this thesis, we describe some properties of interactive applications and congestion control in TCP, look at our design, implementation and experimentation, and analyse the results of these experiments. Here, we introduce each chapter.

- In chapter 2, we look at the traffic pattern of different kinds of interactive applications.
- In chapter 3, we describe how TCP works, and some of the mechanisms that contribute to interactive applications getting worse performance with normal TCP options than greedy streams. We also discuss the characteristics and behavior of *thin streams*.

- In chapter 4, we look closer at why interactive streams suffer under TCP and discuss our solution for one scenario. We also go though the design of our application and go through the assumptions taken when implementing and testing this application.

- In chapter 5, we present multiple prototypes for the programs that were written to multiplex and demultiplex the thin-streams. We thoroughly go through the testing of each prototype and describe the problems that arose and how we solved them.

- In chapter 6, we summarize what we have learn from working on this thesis, and discuss the results and possible future expansions of our work.

Chapter 2

Interactive applications

As we saw in chapter 1, the way we use the Internet has changed over the years. We are now using the Internet much more interactively, i.e., we chat, play games, use VoIP and interact with real-time systems. All these applications generate data streams that behave differently from greedy streams. In this chapter, we explain how interactive applications behave and look at what kind of traffic pattern they generate.

2.1 Properties and requirements of interactive applications

A greedy stream tries to move data between two points in the network as fast as possible, like a File Transfer Protocol (FTP) download, while an interactive application generates a small amount of data with high IAT between the packets.

Table 2.1 shows characteristics such as payload sizes, packet interarrival time and bandwidth consumption for a number of different interactive and **greedy** applications [26].

application	payload size (bytes)			packet interarrival time (ms)						avg bandwidth used	
								percentiles			
	avg	min	max	avg	med	min	max	1%	99%	(pps)	(bps)
Casa (sensor network)	175	93	572	7287	307	305	29898	305	29898	0.137	269
Windows remote desktop	111	8	1417	318	159	1	12254	2	3892	3.145	4497
VNC (from client)	8	1	106	34	8	< 1	5451	< 1	517	29.412	17K
VNC (from server)	827	2	1448	38	< 1	< 1	3557	< 1	571	26.316	187K
Skype (2 users) (UDP)	111	11	316	30	24	< 1	20015	18	44	33.333	37K
Skype (2 users) (TCP)	236	14	1267	34	40	< 1	1671	4	80	29.412	69K
SSH text session	48	16	752	323	159	< 1	76610	32	3616	3.096	2825
Anarchy Online	98	8	1333	632	449	7	17032	83	4195	1.582	2168
World of Warcraft	26	6	1228	314	133	< 1	14855	< 1	3785	3.185	2046
Age of Conan	80	5	1460	86	57	< 1	1375	24	386	11.628	12K
BZFlag	30	4	1448	24	< 1	< 1	540	< 1	151	41.667	31K
Halo 3 - high intensity (UDP)	247	32	1264	36	33	< 1	1403	32	182	27.778	60K
Halo 3 - mod. intensity (UDP)	270	32	280	67	66	32	716	64	69	14.925	36K
World in Conflict (from server)	365	4	1361	104	100	< 1	315	< 1	300	9.615	31K
World in Conflict (from client)	4	4	113	105	100	16	1022	44	299	9.524	4443
YouTube stream	1446	112	1448	9	< 1	< 1	1335	< 1	127	111.111	1278K
HTTP download	1447	64	1448	< 1	< 1	< 1	186	< 1	8	> 1000	14M
FTP download	1447	40	1448	< 1	< 1	< 1	339	< 1	< 1	> 1000	82M

Table 2.1: Analysis of packet traces from thin and **greedy** streams [26].

A greedy stream maximises the use of available bandwidth. It sends data as fast as TCP allows, sending more and more data per second until it has used all available bandwidth. When it tries to send more data than the link is able to handle, TCP starts dropping packets. This signals TCP on the sending side that it should send less data. When more than one greedy stream compete on the same link, this greedy behaviour makes them share the link fairly[1]between them since all streams sends packets as fast as they are allowed.

If interactive applications try to compete with greedy streams, the interactive streams are unable

[1] At least TCP fair.

to get their fair share of the link. This is because of the high IAT of interactive applications. Since they do not send packets all the time, they are not likely to get one through when they need to. This is because all streams on a link fight for the same buffer space in the routers. All packets have the same probability of being dropped, but the greedy streams have many more packets, and thus do not care as much if one gets lost.

2.1.1 Games

One of the popular genres of computer games is Massive Multiplayer Online Games (MMOGs), with 34% of online games falling into this category [11]. In figure 2.1, we see the estimated development of MMOG subscribers, and that in 2008, we exceeded 16 million subscribers [3]. As we can see, there is a steady growth rate to the number of people who play online games. This is one of the reasons that we focus on networked computer games in this thesis.

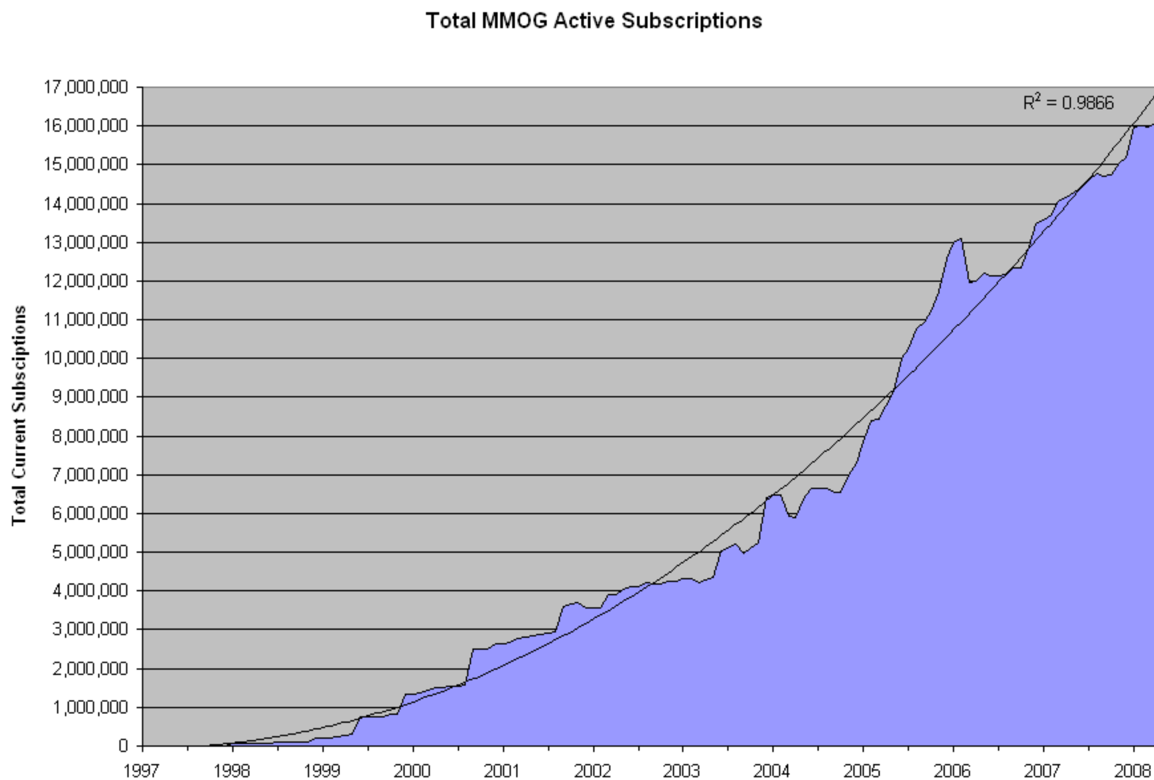


Figure 2.1: Total active subscriptions in MMO games [3]

Gaming is a popular use of the Internet that has strict latency requirements. Different kinds of games have different requirements. More fast paced and high precision games need lower latencies than games with slower interaction. A number of studies measuring player performance

with respect to latency have showed that approximately 100 ms for First-Person Shooter (FPS), 500 ms for role-playing games (RPGs) and 1000 ms for real-time strategy (RTS) games are the thresholds for players tolerance to latency [14]. These three genres can also be seen as MMOGs. We have found an analysis of several games and other interactive applications that is presented in table 2.1 taken from [26]. This table has six MMOGs, where three of them are RPGs, two are FPSs and one is a RTS game. When we look at the characteristics for these games, we see that they all have small packet sizes, and most of them have high IATs. We can see that the high intensity FPS games only have moderate IATs. This is because they need quicker position updates and such for the players to reliably be able to hit each other when aiming. We can also see that all the games have a very moderate bandwidth consumption, and this should not be any problem for a modern Internet connection.

Due to the interactivity, games are very prone to high delays, and if packets have to be retransmitted because of congestion the delays can be several seconds. This manifests as in-game lag. When you are on a ”laggy” connection, objects in the game, typically other players, tend to move erratically. This is because you are missing position updates, and when you receive an update, it seems like the object instantly moved. This contributes to a bad user experience.

2.1.2 Remote systems

It can be very useful to control and run programs on a remote system. We have looked at three common ways of interacting with remote systems. These three applications were tested while editing a text document on a remote computer.

Secure Shell (SSH) is used to get a command line interface or shell on an remote Unix computer. Since the data transmitted is only text and produced by a user typing on a keyboard, it makes small packets with large IAT on the client side. On the server side, the commands typed by the user are executed, and the output is sent back to the user. This may create somewhat larger packets, but still smaller than non-interactive applications like HTTP. It can also be used to tunnel traffic securely through the Internet. After creating an SSH connection to a server, it binds a port on the local host to a server and port on the remote host. Any local connection to this port is first sent through the SSH connection before it is connected to the specified server and port. This can be used to encrypt the data from protocols that normally do not support encryption. One use of this is to forward a graphical interface from the server to the client. This behaves much like a Remote Desktop Connection. If there is high latency on an SSH connection, it manifests as a delay between when you enter something on the keyboard, and

when it appears on the screen. This kind of delay is not as critical as delays in gaming since writing is not usually time dependent, but still very annoying. In table 2.1, we can see that SSH packets have small packet sizes and a large IAT. The bandwidth consumption of SSH is thus very low.

A Remote Desktop Connection (RDC) gives access to the graphical interface of a remote computer. It sends the keyboard and mouse input to the server which sends back a display of the desktop and programs running on the remote computer. It is more vulnerable to latency than an SSH connection, as moving a mouse requires more precision than just typing on a keyboard. If the mouse pointer does not follow your directions quickly, you end up guessing where it is and what you are clicking on. This can very hurtful to the user experience. As we see in table 2.1, RDC has somewhat larger packets and about the same IAT as SSH. The bandwidth consumption is about the double, but this is still quite low.

Virtual Network Computing (VNC) works in similar ways as RDC, and is prone to the same vulnerabilities. We can see from table 2.1 that VNC sends much more data from the server to the client than the other way. We also see that both server to client and client to server IAT is moderate, but if we compare it to the IAT of applications like Hypertext Transfer Protocol (HTTP) or FTP, it can still be called large.

2.1.3 Voice over IP

VoIP telephony systems commonly use the G.7XX audio compression formats recommended by ITU-T. For the two codecs G.711 and G.729, the data rate is fixed at 64 and 8 Kbps, respectively. This means that the packet size is determined by the packet transmission cycle [17]. The ITU-T defines guidelines for acceptable end-to-end transmission times to be 150 ms delay, and a maximum delay of 400 ms [19]. If the delay becomes larger than this, the user experiences that the sound staggers.

In table 2.1, there is an analysis of two Skype [9] conferences, one with using the default UDP protocol, and one with the fall-back TCP protocol. Again, we see that both streams have small packet sizes and moderate IATs.

2.2 Summary

By looking at the traffic pattern of different interactive and latency-sensitive applications, we see that most of them produce small packets with high interarrival times. We have also seen that these kinds of applications can suffer during congestion when used over reliable transport like TCP. To explain why the performance of interactive applications suffer, we examine the way different transport protocols work and how they can affect these kinds of applications in the next chapter.

Chapter 3

Transport

In this chapter, we look at some of the different transport protocols, in particular TCP, since this protocol is used in many of the interactive applications we looked at in chapter 2. We look at how different congestion control mechanisms behave with these kinds of traffic. We also introduce a definition of thin-stream applications.

3.1 Choosing a transport protocol

We are limited to the following options when choosing a transport protocol for a time dependent application:

1. Use TCP which is reliable but may give high latency in certain conditions for our application class [28].

2. Use unreliable protocols like UDP or Datagram Congestion Control Protocol (DCCP), and implement in-order delivery and reliability on the application layer [21, 27].

3. Use an experimental protocol like SCTP that is reliable and does not give high latency for our application class [29].

One solution to the latency problems observed when packets are dropped due to congestion was to use Quality of Service (QoS) to reserve a portion of the bandwidth in the net [18]. But today, there is practically no support for QoS over the Internet.

As we saw in table 2.1, some interactive programs use UDP. But, they often have to use TCP as a fall-back since UDP is blocked by the firewalls of some ISPs. This mean that even if the application could work with UDP, a good solution for TCP is also needed in the cases where the application need to fall back to this.

Newer and experimental protocols like SCTP and DCCP are not widely supported on all the end-user systems, and therefore difficult to use in commercial software.

So, since none of the other options are viable, we have to make it work with TCP. This is also what most game companies today that need reliable transport do. Examples of this is World of Warcraft, Anarchy Online and Age of Conan.

3.2 TCP

TCP, together with IP, is one of the two core components of the original Internet Protocol suite. It is the de facto standard when reliable transport over the Internet is needed. Because of this, it is widely supported and outgoing traffic is usually not stopped by ISP firewalls as other transport protocols might be. TCP offers the following services:

Ordered data transfer The receiver application gets the data in the same order as it was sent.

Reliability If any packet is lost or erroneous, it is retransmitted.

Error detection A checksum is used to check if the packet has been altered since it was sent.

Flow control The sender does not send data faster than the receiver can handle.

Congestion control If network congestion is detected, TCP adapts the send rate to fairly share the bandwidth with other streams.

These services imply the need to keep a state. Some of this state data needs to be exchanged between the endpoints, and is embedded in each packet. This state is arranged in a header as seen in figure 3.1. Since the header must be sent with each packet, this makes the overhead large for small packets. We see that the header includes two port fields. These numbers combined with the source and destination IP address from the IP header uniquely identifies each TCP stream. The ”Sequence number” keeps track of how many bytes have been sent, and the ”Acknowledgment number” indicates how many bytes have been received. These two numbers does not count from zero, but from a random number that is exchanged during connection setup. ”Window size” tells the sender how much data it can send and is updated with each Acknowledgement (ACK). ”Data offset” specifies the size of the TCP header in 32-bit words. Since this field is 4 bit, the maximum header size is 15 word, or 60 bytes. The minimum size of a TCP header is 20 bytes, leaving 40 bytes for optional header information. The other field are used during the setup and tear-down process, and to keep track of the TCP state.

<table>
<tr><th>Bit offset</th><th>0-3</th><th>4-7</th><th>8-15</th><th>16-31</th></tr>
<tr><td>0</td><td colspan="3">Source port</td><td>Destination port</td></tr>
<tr><td>32</td><td colspan="4">Sequence number</td></tr>
<tr><td>64</td><td colspan="4">Acknowledgment number</td></tr>
<tr><td>96</td><td>Data offset</td><td>Reserved</td><td>Flags</td><td>Windows Size</td></tr>
<tr><td>128</td><td colspan="3">Checksum</td><td>Urgent pointer</td></tr>
<tr><td>160
...</td><td colspan="4">Options
...</td></tr>
</table>

Figure 3.1: TCP header

3.2.1 Flow control

Receiver's advertised Window (RWND) is a receiver-side limit on the amount of unacknowledged data [10]. It is used to avoid that the sender sends more data than the receiver is able to buffer. This window size is sent with each ACK the receiver sends, so the sender always has the updated size. If the sender receives a RWND of 0, it stops sending data and starts a persist timer. This to avoid deadlocks where the window size update is lost and the sender is unable to send data. If the timer expires, TCP tries to recover by sending a small packet.

3.2.2 TCP congestion control mechanisms

There are several different mechanisms in TCP to assure that each stream in the network gets its fair share of the bandwidth and that the streams do not overwhelm the network. We now take a closer look at the most important ones in the version of TCP that is called NewReno.

3.2.2.1 Congestion window

The Congestion Window (CWND) is a sender-side limit on the amount of data the sender can transmit into the network per Round-Trip Time (RTT). TCP must not send data with a sequence number higher then the sum of the highest acknowledged sequence number and the minimum of CWND and RWND [10]. The initial value of CWND depends on the size of the Sender Maximum Message Size (SMSS), and can hold between 2 and 4 segments.

3.2.2.2 Slow start and congestion avoidance

The task of this mechanism is to estimate the available bandwidth. It uses the "Additive Increase, Multiplicative Decrease" (AIMD)-algorithm to achieve this [10]. We can see an example of AIMD and slow start in figure 3.2. Slow start is used while the congestion window is less than the slow start threshold (*ssthresh*). The congestion window grows with up to one SMSS each time an ACK is received that covers new data, until CWND reaches or exceeds *ssthresh* [10]. Congestion avoidance is then used while the congestion window is larger then *ssthresh*. Initially, *ssthresh* may be set arbitrarily high, some implementations of TCP set it to RWND. In the congestion avoidance phase, additive increase is used, incrementing CWND by one full-sized segment each RTT until congestion is detected. Any loss of a packet is considered

a sign of congestion, and triggers multiplicative decrease. *ssthresh* is set to half the congestion windows size, and then slow start is initiated.

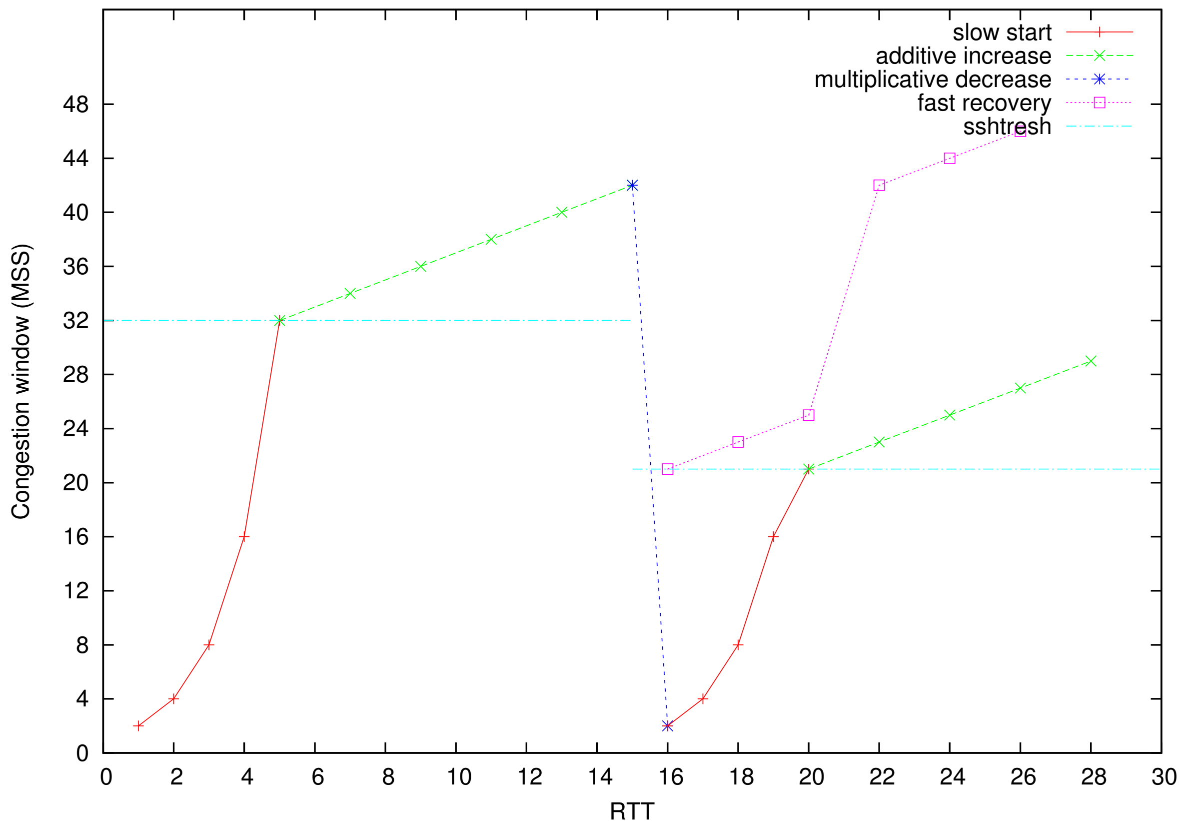


Figure 3.2: An example of AIMD, slow start and fast recovery

3.2.2.3 Fast retransmit

When a TCP receiver gets data out-of-order, for example because a packet is lost, it sends an ACK with the next sequence number it misses. Since it also sent an ACK with that sequence number when it received the last in-order segment, this is a Duplicate Acknowledgment (dupACK). The fast retransmit mechanism uses these dupACKs to detect that a packet has been lost. Three dupACKs without any intervening ACKs is used to indicate a lost packet. When a TCP sender get these three dupACKs for a segment, it sets the CWND to half it's current value and retransmits the lost segment right away, instead of waiting for the retransmission timer to expire [10].

3.2.2.4 Fast recovery

This mechanism is triggered after a fast retransmit. When fast recovery was first introduces, instead of going into slow start, TCP would set the congestion windows to *ssthresh* plus 3*MSS and continue to send data segments as normal. If an ACK for new data is received before a timeout was triggered, the congestion window would be set back to *ssthresh*.

This algorithm were updated in NewReno to instead set *ssthresh* to half the congestion windows [15]. And when an ACK is received, if it covered some but not all new data, a new fast retransmit was initiated followed by a new fast recovery period.

We can see how fast recovery works in comparison with slow start in figure 3.2. Here, we can see that the congestion window and *ssthresh* is set to half of what the congestion window was before the loss was detected. We can see that the congestion window jumps back up again after a little while, this is because an ACK covering new data was received.

3.2.2.5 Retransmission timeout and exponential backoff

Retransmission Timeout (RTO) is how long a packet has to wait before being retransmitted, if fast retransmit is not triggered. If the sender does not receive an ACK for a packet before the RTO expires, the sender retransmits the packet.

Exponential backoff doubles the RTO for a segment each time it is retransmitted as explained above. This is done to prevent a congestion collapse, a state where little or no useful communication gets through a router, when faced with severe congestion.

3.2.3 Retransmission timeout calculation

To be able to compute the current RTO, a TCP sender needs to maintain two state variables, Round-Trip Time Variation (RTTVAR) and Smoothed Round-Trip Time (SRTT) [25]. These variables are calculated from the RTT measurements. The RTO calculation specification says that RTT measurements must use Karn's algorithm [20]. This algorithm says that the RTT measurements of a retransmitted data segment must never be used as the basis for RTO calculations.

In the following equations, G is the granularity of the system clock in seconds. Here is the three different equations used to calculate RTO at different stages of the connection.

1. Until an RTT measurement can be made for a TCP connection, the RTO should be set to three seconds.

2. When the first RTT measurement R is made, the host can then use equation 3.1 to calculate the RTO.

$$\begin{aligned} K &= 4 \\ SRTT &= R \\ RTTVAR &= \frac{R}{2} \\ RTO &= SRTT + max(G, K \times RTTVAR) \end{aligned} \tag{3.1}$$

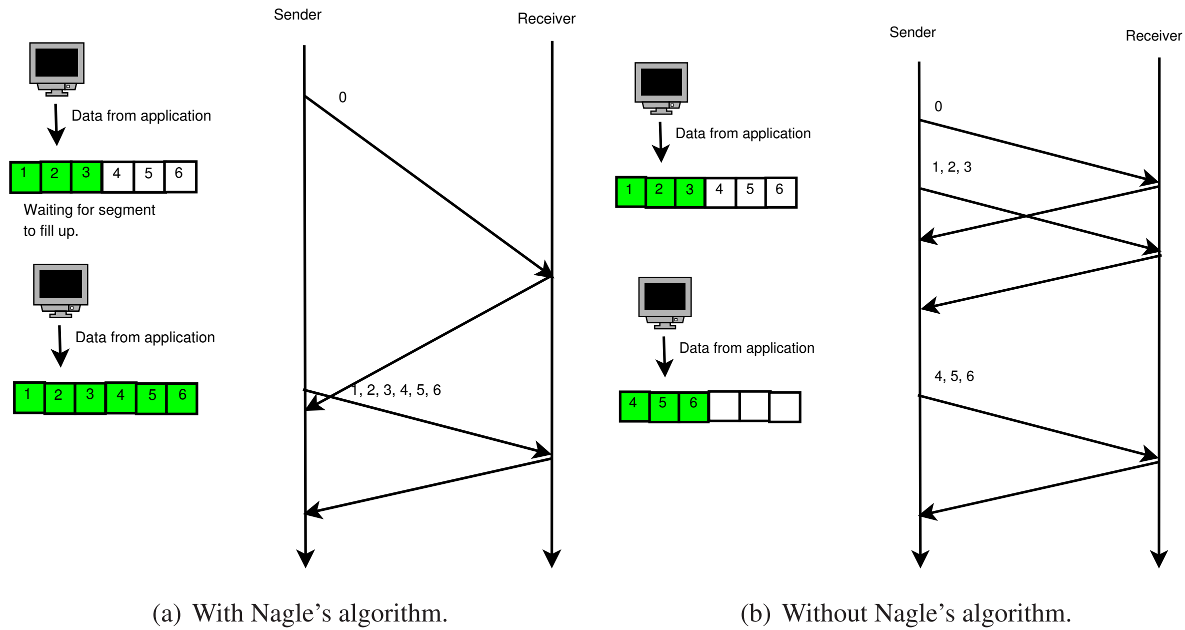
3. For each subsequent RTT measurement R', the host then uses equation 3.2 to calculate the RTO.

$$\begin{aligned} & \alpha = \frac{1}{8}, \beta = \frac{1}{4} \\ RTTVAR &= (1-\beta) \times RTTVAR + \beta \times |SRTT - R'| \\ SRTT &= (1-\alpha) \times SRTT + \alpha \times R' \\ RTO &= SRTT + max(G, K \times RTTVAR) \end{aligned} \tag{3.2}$$

If any of these equations calculate a RTO of less then one second, the RFC specifies that RTO should be rounded up to one second. This is not true in several newer operating systems, where they permit an RTO of less than one second.

3.2.4 Nagle's algorithm

This mechanism was introduced to reduce the overhead of TCP packets. It avoids sending unnecessary small packets by delaying transmission while there is unacknowledged data until the segment is full or it receives an ACK [24]. Since small packets have large overhead due to the size of the TCP header, this saves the sender a lot of unnecessary bytes. In figure 3.3(a), we see an example of how TCP works with Nagle's algorithm turned on. The sender delays the data segments, 1, 2, and 3, in the network buffer since it has not received an ACK for segment 0. When segments 4, 5 and 6 fills up the buffer, the sender sends the entire buffer. The same transmission with Nagle's algorithm turned off is shown in figure 3.3(b). Here, the data segments is transmitted without delay as soon as they are put into the buffer.



(a) With Nagle's algorithm. (b) Without Nagle's algorithm.

Figure 3.3: An example of packet transmission with and without Nagle's algorithm when there is unacknowledged data on the connection.

3.3 Thin streams

By looking at table 2.1 and comparing the average payload size and IAT of interactive applications to that of known thick-stream applications like HTTP or FTP downloads, we can see the distinct differences. The interactive applications all have high IATs, compared to the thick-streams, and where a thick-stream always fills a segment up to the Maximum Segment Size (MSS), the interactive applications on average only fill 10% of the segment.

This kind of traffic pattern is what we define as *thin-streams*. That the packet sizes are relatively small and since the IAT of these packets are so high, the streams' transmission rate is not limited by congestion control, but by the application's data production.

This means that the interactive applications we described in chapter 2 can be categorised as thin-stream applications.

3.3.1 How do TCP's mechanisms affect thin streams?

The biggest problem that greedy, or **thick**, streams experience with TCP mechanisms is slow start, since it reduces their bandwidth whenever there is a packet loss. This is not a problem for

thin streams, since they send their packets so infrequently that the restrictions put down by slow start does not affect their transmission rate.

Of the mechanisms we looked at in section 3.2, three are particularly bad for applications with specific latency requirements like the interactive thin-stream applications we looked at in chapter 2. These problems are all due to high IAT of thin streams.

Since thin streams have such a high IAT, they usually do not send enough packets to trigger fast retransmit, and thus is unable to get fast recovery. To be able to use the fast retransmit mechanism, a stream must send three packets and receive the dupACKs for these packets after a packet is lost. If we have an RTO of 2 seconds, an RTT of 0.2 seconds and the thin stream sends a packet each 0.8 seconds, then if a packet got lost, the sender would only receive two dupACKs before the RTO timed out and retransmitted the packet. Now TCP again needs three new dupACKs to trigger fast retransmit. This would also trigger exponential backoff.

If exponential backoff triggers as we saw above, it retransmits the earliest unacknowledged packet and doubles the RTO of the lost packet. This is quite hurtful to thin streams because when exponential backoff begins, the stream suffers higher and higher delays if the same packet gets lost again. This is also because of the high IAT.

Flow control is not an issue for thin streams since the packet sizes are so small that receiver does not have any problems processing them. Nagle's algorithm would also make problems for thin streams, but luckily this can be turned off by the application.

3.4 Summary

In this chapter, we defined the term *thin stream*, and showed that the interactive applications from chapter 2 qualify as thin streams. We also saw that these streams are unaffected by the bandwidth limitation of congestion control, but that the retransmission delays of TCP congestion control mechanisms can create very high latencies for thin streams.

In chapter 4, we look closer at this problem and propose a solution for some scenarios.

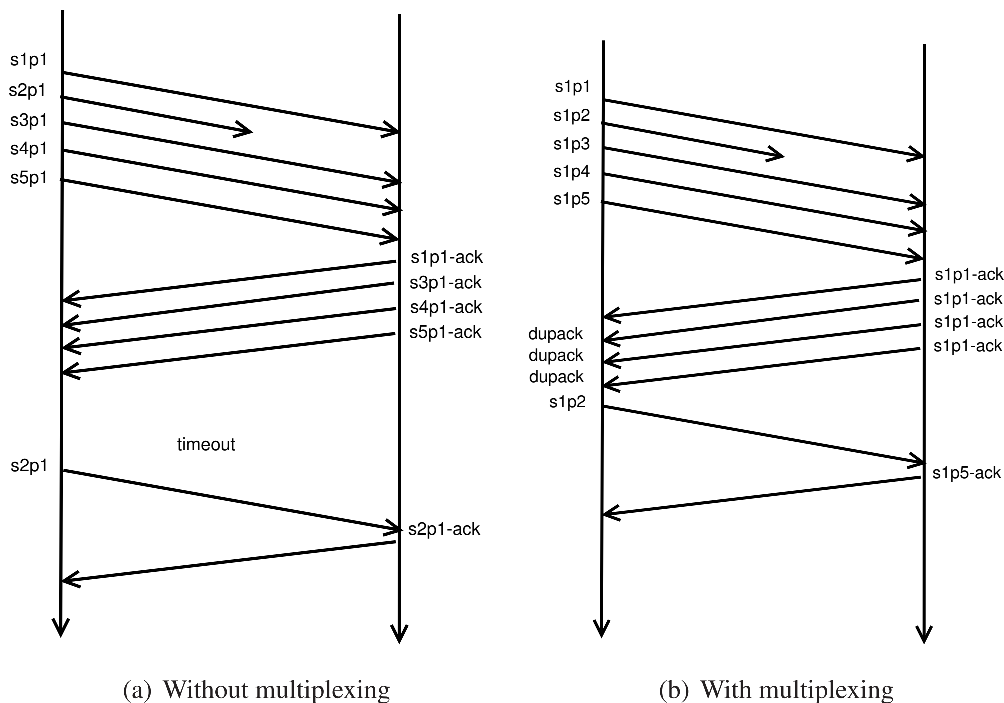
Chapter 4

Design

In chapter 3, we looked at thin streams and how they suffer from high latencies when TCP applies congestion control. We now look at many thin streams at once, and discuss how this can be used to better the performance. We present multiplexing of several thin streams as a promising idea to solve the latency problem where this is possible. We then go through the assumptions made when implementing and testing this solution, and lastly, we look at an overview of our proposed solution.

4.1 Bundling of streams

As we saw in chapter 3, there are many mechanisms in TCP that make thin streams suffer latency-wise. Since the IAT of packets in a thin stream is high, fast retransmit is unable to trigger because usually the thin stream does not send four packets before a timeout occurs. Some studies have shown that multiplexing several thin streams into one TCP connection gives a theoretical gain in performance [16, 22]. Our solution is therefore to try and implement this approach, multiplexing several thin streams into one stream. This way there is a much higher probability of receiving the three dupACKs needed to trigger fast retransmit whenever a packet is lost. We can see a comparison of how fast retransmit works with and without this kind of multiplexing of thin streams in figure 4.1.



(a) Without multiplexing (b) With multiplexing

Figure 4.1: Example of retransmission with and without multiplexing

In figure 4.1(a), we see five different TCP streams named `s1` through `s5` each sending one packet, `p1`. `s2`'s packet is lost, for example due to congestion, all other packets arrive at their destination and an ACK is sent back. `s2` must now wait for a timeout before it can send `p1` again. This is because the ACKs for the other streams do not count as dupACK for this stream. This time the packet arrives correctly and ACKs is sent back. In figure 4.1(b), we see the same scenario, just that we now use our system where all the data goes through one stream, `s1`, so the packets are now numerated in the order the they were sent in figure 4.1(a). Again, the second packet is lost, and since TCP always ACKs with the packet it wants to receive next for each

stream, we now get Duplicate Acknowledgments (dupACKs) which, as we saw in chapter 3 triggers a fast retransmit. This reduces the overall latency since we do not have to wait for timeouts if there is enough data going through the stream.

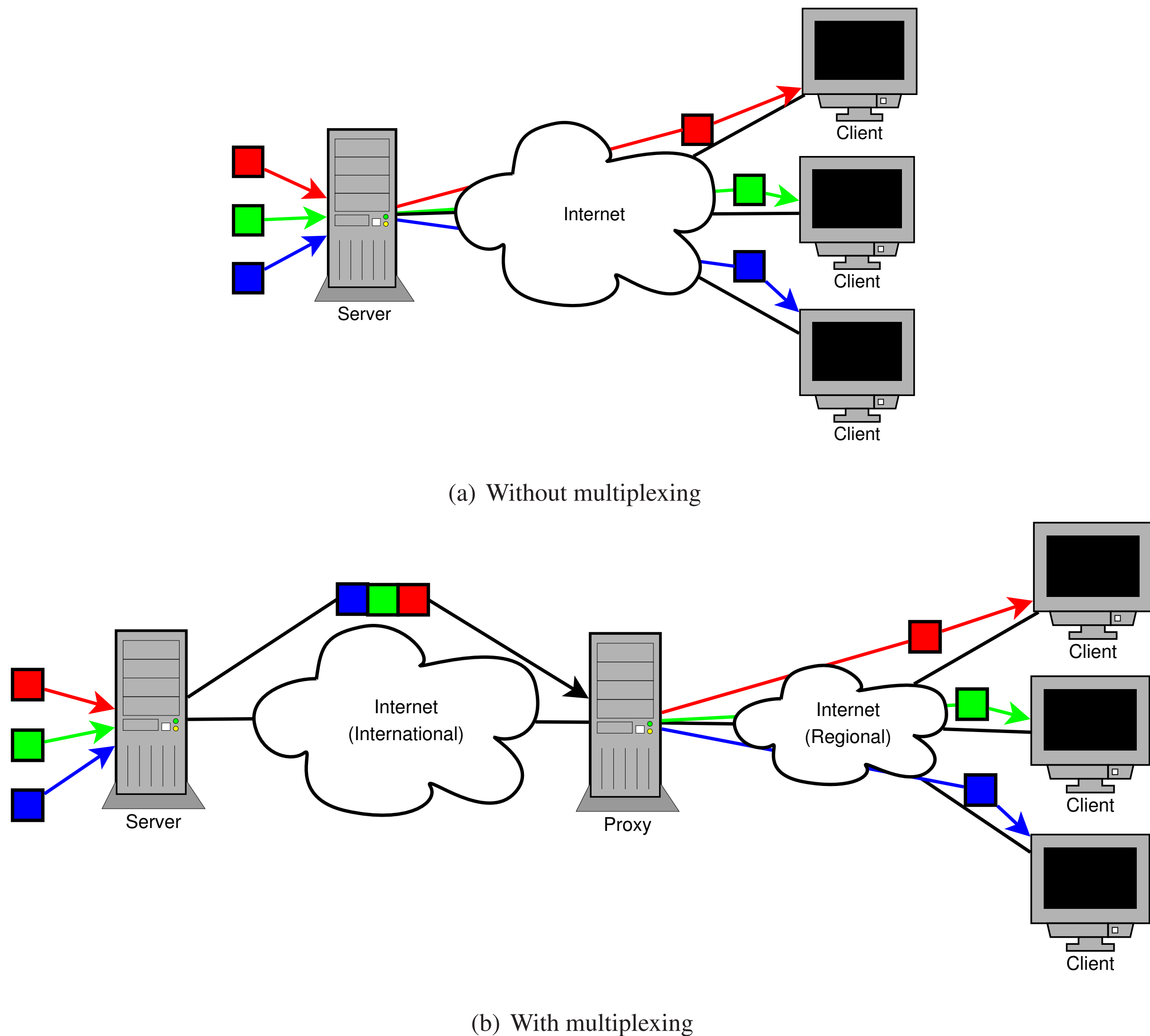
One example where this idea could be used is between an online game server with many clients in one region. Instead of having one connection to each client from the server, the game company could place regional proxies closer to the client, and then bundle the traffic between the server and the proxy. The placement of these proxies would have to take into consideration usage patterns and how many users typically were online from different regions, but one suggestion would be to place a proxy in each country. We have drawn an example of a game server with and without the use of this idea in figure 4.2. Here, we see the classical approach of one connection between the server and each of the clients in figure 4.2(a). In figure 4.2(b), we use multiplexing on the long, and maybe congested, line between the server and the proxy. We bundle each of the outgoing packets into one stream, and then unpack them on the proxy. Then, we send the data as normal between the proxy and each of the clients. This means that on the stretch were we are most likely to encounter congestion, we are better equipped to handle it.

By combining the thin streams into a thick stream, we expect to be able to trigger fast retransmit and not having to wait for a transmission timeout. Exponential backoff would also behave as normal since the stream would have a low IAT because of the bundling, and not create any larger problems. We do expect to see a slightly bigger RTT since we now have to do some additional processing on each packet. And there might also be problems if we bundle too many streams, as we could then get into bandwidth problems during congestion. These effects needs to be investigated, and we do so in chapter 5.

4.2 Assumptions and abstractions

In our tests, we used an approximately one hour long packet trace from one of Funcom's Anarchy Online servers. This gives us a more realistic traffic pattern compared to if we would have used a packet generator to simulate multiple thin streams. The program used to replay this packet trace is called `tracepump`. It was developed by Andreas Petlund in his work with thin streams [26]. It reads a trace file an recreates the sending side of several streams, but replaces the destination with one given as a parameter.

The server received no feedback from the clients. We are only interested in improving the delay from the server to client. It is more realistic to only make changes server-side, since it might



(a) Without multiplexing

(b) With multiplexing

Figure 4.2: Multiplexing

not be feasible to impose changes on the clients. Since we used a trace from a real game server communicating with many clients, we have a realistic view of how the server would respond to clients without having to actually send data to the server. A server might respond differently if packets from a client is delayed or lost, but again, since we are using a real packet trace, this is not a problem.

The testing were done without cross traffic. We instead used `netem` [5] to emulate the different network conditions. `Netem` can create random packet drop and delay packets.

4.3 System design

To design experiments, we simplified the scenario to six essential steps. These steps are shown figure 4.3. For each of these steps, we list the decisions we had to make and why we chose as we did.

1. Capturing the thin streams for multiplexing.
2. Sending the multiplexed stream to the proxy.
3. Emulating network conditions.
4. Receiving the multiplexed stream.
5. Demultiplexing and sending the individual thin streams to the clients.
6. Client receiving the data.

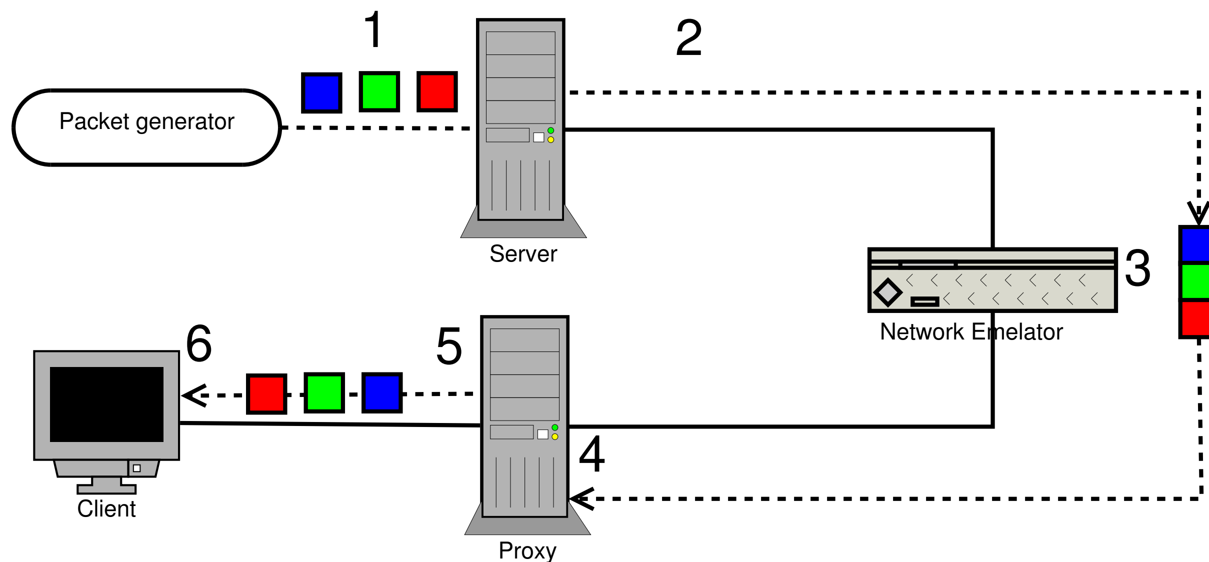


Figure 4.3: A breakdown of each step in our system

Steps 1 and 2 happen on a machine running the server software, step 3 on a separate machine used to route the network traffic between the server and proxy, step 4 and 5 happen on a machine running the proxy software, and lastly step 6 on a fourth machine. We separated the different parts of the designed system on different machines to better be able to simulate real working conditions. Each of these machines log all network traffic so we can verify the results after each test.

4.3.1 Step 1

On step 1, we looked at three different solutions:

A. Rewrite the sending application to send the streams as one thick stream.

B. Write a program that can be used as a sink for sending application.

C. Write a program that gathers the streams from the network.

We decided that understanding and rewriting the sending application would take to much time. We also wanted our system to be somewhat transparent so we might use several sending applications to generate the thin streams.

If we would write our program as a sink for the sending application, we would also loose some transparency. We would not be able to get the destination information directly from the packet headers, and a problem of finding the clients would have to be solved.

We therefore decided to write a program to gather the thin streams directly from a network interface. This way we can get all the information we need from the packet headers, and the system can be used with different packet generators.

4.3.2 Step 2

On step 2, we had the following decisions:

A. Wait until we have a full segment or a timeout has elapsed before sending.

B. Send the packets as soon as we receive them.

As we see in chapter 5, we tested both these variations thoroughly. Our first prototype used the method described in A, while the other prototypes used the method described in B. We found that sending the packets as soon as possible gave the best results.

4.3.3 Step 3

In step 3, we decided to use `netem` [5] and `tc` [2] to emulate different network conditions. We choose to use these programs since we had experience with them and knew how to make them do what we wanted to do in our tests. `Netem` and `tc` are programs found in Linux to modify the behavior of the network interfaces. They can be uses to, for example, limit the connection bandwidth, create artificial network delay and drop percentages of packets.

4.3.4 Step 4

In step 4, we decided to implement a simple socket program with a select-loop. This enables us to have more than one server per client. In most MMOGs you can choose from many servers, so we want several servers to be able to use the same regional proxy. Whenever one of the server connections has any new data, this data buffer is sent to another part of the program.

4.3.5 Step 5

In step 5, the proxy program reads out each header and payload from the multiplexed packet it got from the select-loop. For each header and payload pair, it checks if a connection to the client is open. If there is no open connection to that client, one is established. It then sends the payload.

4.3.6 Step 6

The client is just a data sink. It receives and discards incoming data and sends ACKs back to the proxy.

4.4 System overview

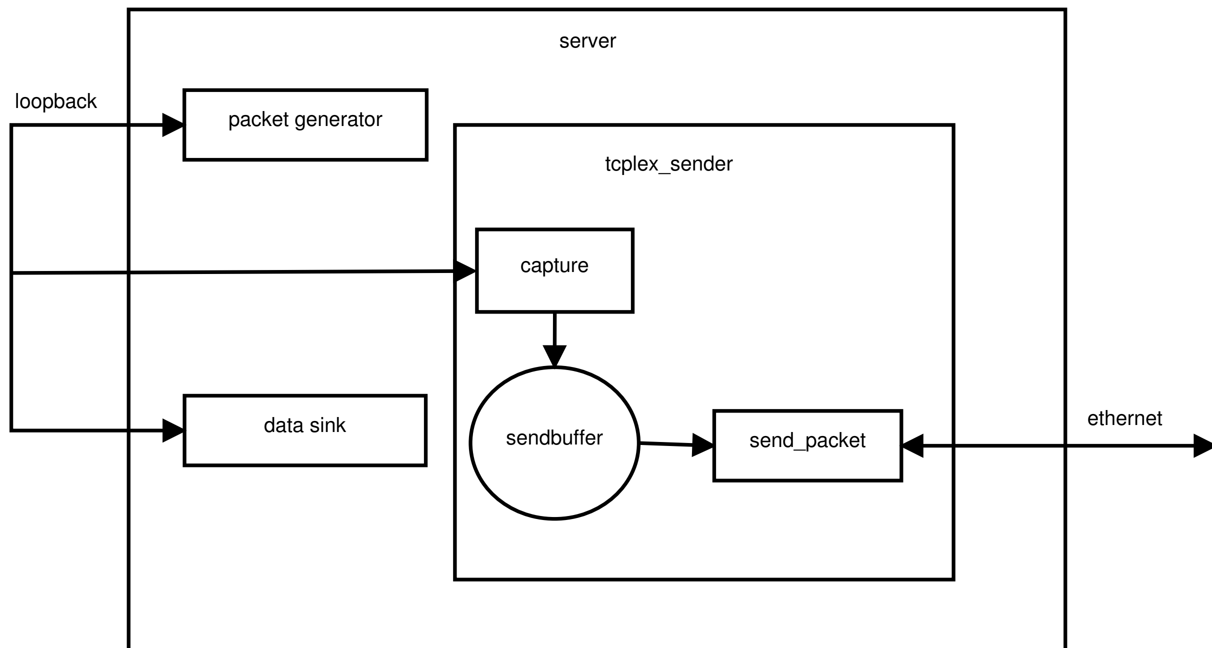
Our system consists of three parts, a server, a proxy and clients. The server and proxy are shown in figure 4.4, and the client is a simple data sink that just ACKs the packets it receives and discards the data. In figure 4.4(a), we can see that the server runs a packet generator. This

program is run locally in two modes, send and receive, and is responsible for producing the thin packet streams. It does this by ”replaying” a packet-trace file over the loopback device. On the client, we use this program in receive-mode. Our system is called `TCPlex`, and the sender side of this program is running on the server. It captures the thin streams from the loopback device, and examines each packet. It then adds a header containing the size of the data payload, the destination address, the source and destination port and the sequence number. This header and the payload is put into a send buffer and a function is called that sends this buffer to proxy via the emulated network.

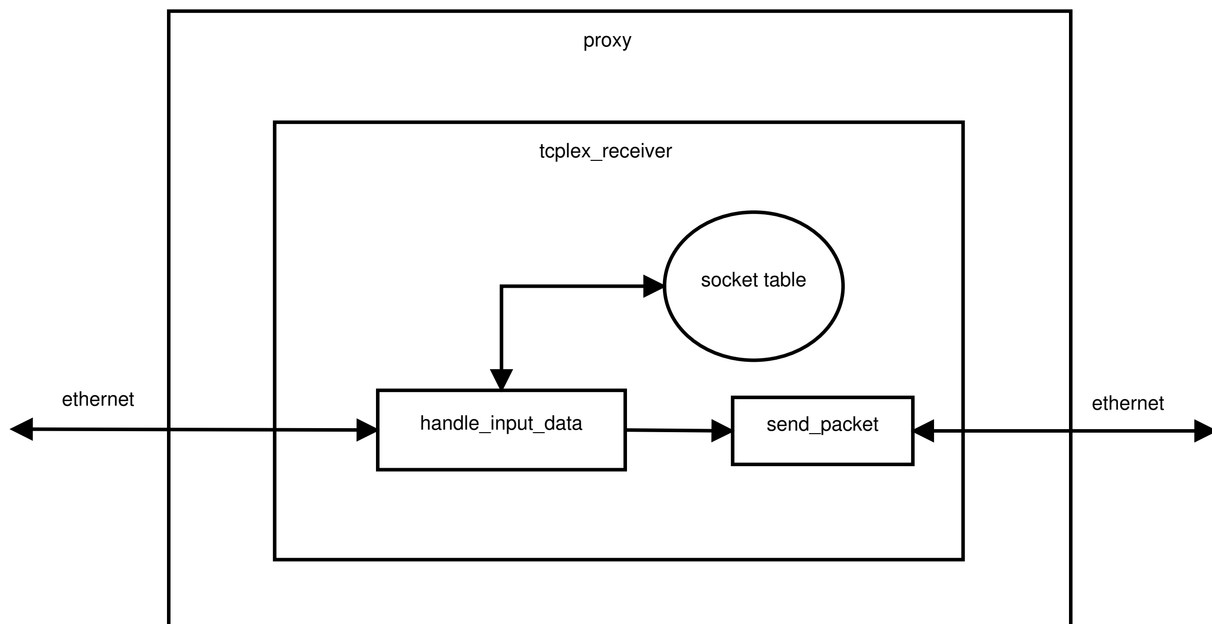
In figure 4.4(b), the proxy receives a multiplexed packet. The buffer containing this packet is sent to a function which extracts the header information put in by the server. It then checks to see if there is a connection associated with this stream in the socket table. If there is already a connection to a client for this stream, it sends the payload data to the client, if not, it creates the connection before sending the data.

4.5 Summary

In this chapter, we presented a possible solution to the latency problems caused by TCP by combining many thin streams. We discussed all decisions made during the design process and gave a complete design of the different parts of the system. In the next chapter, we show how we implemented this system through an iterative process, with thorough testing between each new prototype, and a final comparison of latency in thin-stream applications with and without our system.



(a) Server



(b) Proxy

Figure 4.4: System overview

Chapter 5

Implementation, experiments and analysis

In this chapter, we describe the implementation details of the multiplex server and proxy. First, we go through the program flow of the first prototype. Then, we look at the testing environment and explain how we did the testing, what network conditions we emulated and how to understand the results. After this, we describe what we learned from the first prototype and do an iterative process of improvements and tests to make it better.

5.1 First prototype

For the first prototype, we implemented the ideas from chapter 4. We are now going to describe how the program works and explain what program functions the server and proxy use. In both the server and the proxy, we have disabled Nagle's algorithm to minimize the latency. We decided to keep the first implementation as simple as possible, and thus, both the server and proxy run in a single thread each.

5.1.1 Multiplex server implementation

We implemented the server with the concepts found in figure 4.4(a). The program goes through an infinite loop of acquiring packets from the loopback device, this is done with `libpcap` [7]. We decided in chapter 4 that we want to capture ”in-flight” data to get the most transparency, we therefore need `libpcap` since this is one of the easiest tools to work with to capture data from network interfaces. The captured packets are put into a buffer, and a multiplexed packet is sent out when there is no more room in the buffer, or a timeout has occurred. The following functions are used by the server program:

got_packet is called by `libpcap` for each packet it sniffs from the loopback device. If the packet has any payload, `got_packet` copies destination and source information and the payload to a sending buffer. If the buffer is full, the `send_packet` function is called before copying to the buffer and updating the counter for payloads in the buffer.

send_packet is used to send a combined packet from the sending buffer. It first writes the number of payloads in the buffer and the length of the buffer to the first 4 bytes of the buffer, so the receiver knows how many payloads it must read. Then, it sends the buffer and resets the counters.

5.1.2 Multiplex proxy implementation

We implement the proxy with the concepts found in figure 4.4(b). The program goes through an infinite loop where it runs `select`. Select returns when there is a packet ready for processing. It then calls `handle_input_data` which splits the multiplexed packets into sets of header and payload and calls `send_packet` for each set. The following functions are used by the server program:

handle_input_data is called for each packet received by select. It reads out how many payloads it contains, then runs through the packet and calls `send_packet` for each payload and corresponding header.

send_packet gets a header and a payload. It reads the header and checks if it has a connection for this stream. If it finds one, the payload is sent. If a connection is not found, it establishes one first.

5.2 Test environment

In this section, we describe the different tools we used during testing, our test environment, explain how the different test parameters influence the tests and how to understand the results.

5.2.1 Tools and techniques

Our own program `TCPlex` is run in sender mode on the server and in receiver mode on the proxy. We use `libpcap` [7] on the server to gather the streams made by `tracepump`, which reads a packet trace from a file and generates packets with the same timing and size as the ones in the file. On each of the machines in the network we run `tcpdump` [7] to get the packet trace from the experiments. These dump files together with logs from `TCPlex` are then analysed. We wrote analyzation scripts in `python` [8] and `awk` [4], which generate data files that can be plotted by `Gnuplot` [6]. We used `ntpdate` [1] to synchronize the system clocks. Finally, as stated before, we also used `tc` and `netem` [5] to simulate different network conditions that can occur on the Internet.

5.2.2 Testbed

Our testbed consists of four computers, as seen in figure 4.3. The Server is where we run the `TCPlex` server and `tracepump` sender. Here, `tracepump` sends all the thin streams on the loopback device, and `TCPlex` reads them and multiplexes them into one thicker stream before sending. On the Netem machine, we run `netem` to simulate different network conditions. The Proxy runs the `TCPlex` client and is where we receive the multiplexed thick stream and demultiplex it into the original thin streams. On the Client, we only run `tracepump` receiver as a sink for the `TCPlex` client.

In the baseline tests, we used `tracepump` in send-mode on the Server, and in receive-mode on the Client. The packets went through the same network path, and were exposed to the same network conditions, as when running the `TCPlex` tests.

5.2.3 Test parameters

To simulate different Internet conditions on our network, we changed some of the network parameters between each run of a systems test. The following values are the same values as in the study that simulated the multiplex performance gain [16]. We have found that the chosen values are realistic to simulate Internet conditions [12, 13, 23]:

Packet loss: This parameter defines how many of the packets were dropped. The parameter is the total percentage of packets dropped from the link statistically over time. We used both 1% and 5% packet loss in our tests.

Delay: This parameter specifies how much delay is added to each packet traveling through the `Netem` machine. The delay is added in both directions, meaning that the delay is added twice to the round trip time. We used both 100 ms and 300 ms delay in our tests.

Jitter: This parameter decides how much the delay varies. We used 0% and 10% jitter in our tests. This means with 100 ms delay and 10% jitter, a packet has a random delay between 90-110 ms. With 500 ms delay and 10% jitter we would get a random delay between 450-550 ms.

In addition to these varying parameters, we had 50 ms added delay, with no jitter and no loss, between the proxy and the client. This was added to get a more realistic RTT between clients and a regional proxy.

5.2.4 Measuring delay

We decided to compare the end-to-end delay of the packets as the success metric of our system. To be able to measure this delay, we needed to know when a packet first was generated on the server, and compare this to when that same packet arrived at the client. This was not trivial, as the system clocks run at slightly different speeds and we needed precision in the millisecond range. We decided to synchronise the clocks in each machine before and after each test, and

see how much the clock had drifted from start to finish. This drift was then applied to each timestamp to correct it. When all timestamps on both the server and the client were corrected, we could compare them against each other to find out how long it took the packet to travel from the server to the client.

This seemed to work at first, as we got results in the time range we were expecting. But some of the tests we ran showed very sporadic delays that could not be explained by the system. After doing some research, we found that `ntpdate` slews the clock if it is under 500 ms wrong. This means that instead of correcting the clock, it is sped up or slowed down the clock until it was corrected. This is what caused our measurements to vary between different runs of the same test. We found that `ntpdate` could be forced to set the clock regardless of the current offset, and after running some new tests, we now had an accurate way of measuring the end-to-end delay.

5.2.5 Understanding the graphs

All the graphs we show you in this chapter are boxplots. They are used to convey statistical data. We explain how to read them in a small example in figure 5.1. Each plot has a box with a horizontal line drawn through it, and there is also a horizontal line drawn above and below, which is connected to the box. These two lines represent maximum and minimum observed values respectively. The low and high end of the box represent the lower and upper quartile, or first and third quartile respectively. The line through the box represents the median, also known as the second quartile. If a maximum value falls outside the scope of the graph, its value is shown on top of the graph.

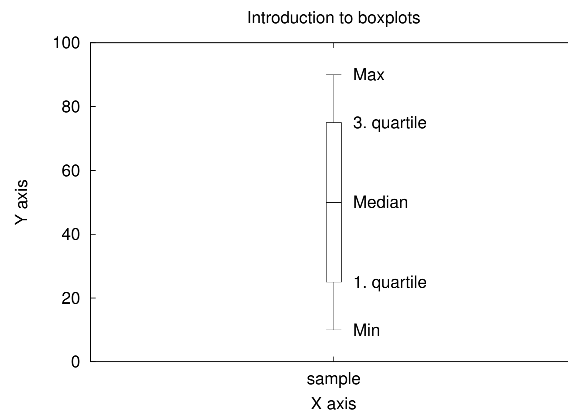


Figure 5.1: An example boxplot.

Quartiles divide a data set into four equal parts. If you sort the data set from lowest to highest value, the first quartile is the value which has one fourth of the lower values below itself. The second quartile, or median, is the middle value and have half the values before and after it. The third quartile is the value with one fourth of the values over it. The difference between the upper and lower quartiles is called the interquartile range, and is what the box in a boxplot represents.

All graphs shown have some ”cryptic” letters and numbers on the x-axis, for example ”11 100d

0j". This defines the network parameters between the server and proxy for that test. "l" stands for loss, and the number before it is the percentage of loss. "d" stands for delay and the number before it is the number of milliseconds packets are delayed each way. "j" stands for jitter and is the percentage of jitter applied to the delay. Thus, in the example, we have 1% loss, 100 ms delay and 0% jitter.

5.3 First tests and conclusions

We implemented the prototype described above, and then we did some initial testing. Figure 5.2 and figure 5.3 show statistics for packet arrival time with and without the use of our system. We observe from these tests that our system performs drastically worse than the baseline test[1]. Both maximum values and the interquartile ranges were much higher in our system than in the baseline test. We started to analyse all available data to find the cause of this. We first thought that the difference might be from CPU usage in our system and that one or more of the subroutines the packet must go through were slow.

We therefore compiled our system for profiling and ran some new tests. The profiling data showed that on average, none of the functions used much time, all functions were in the nano- and microsecond range.

We then went back to the network logs and wrote some new tools to analyse specific paths in the system. We measured the time for the following steps of our system:

1. From the time when the packet is picked up on the loopback device until it is sent.
2. Between the server and proxy.
3. Between the proxy and client.

The analysis of logs between the server and proxy can be seen in figure 5.4. Here, we see that interquartile ranges are where they should be, on 100 ms and 300 ms respectively, as the network parameters permitted. In figure 5.5, we see the travel times between the proxy and the and client are at 50 ms.

[1] In all test runs, we did not get accurate data for the 5% loss and 300 ms delay TCPlex tests. Tcpdump did not manage to capture all the packets to and from the proxy, and we are thus not sure if the bad results we see in figure 5.3 and figure 5.8 are real.

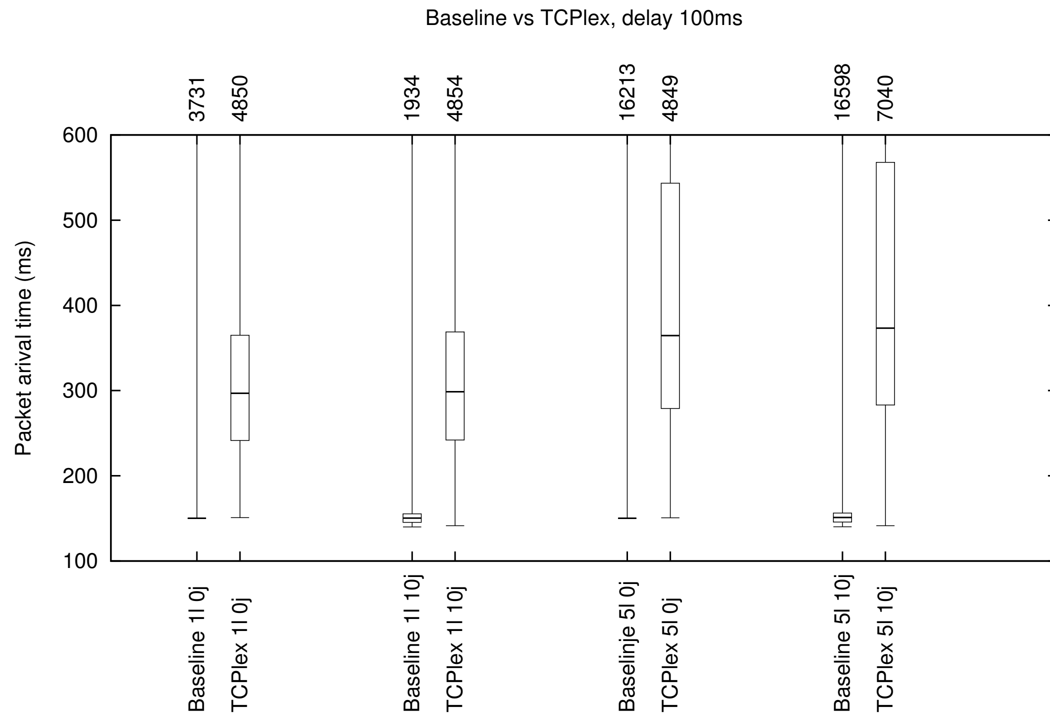


Figure 5.2: Comparison of Baseline and TCPlex tests with 100 ms delay

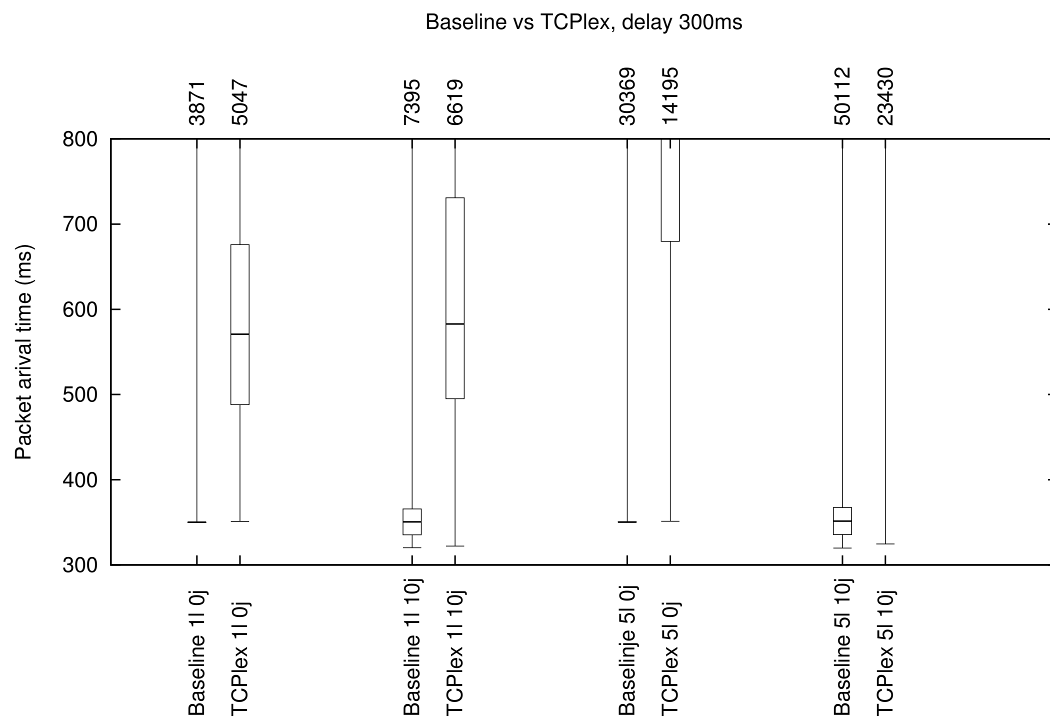


Figure 5.3: Comparison of Baseline and TCPlex tests with 300 ms delay

The maximum values seen in these two graphs are caused by retransmissions and from analysis of the data files, we find these deviant values in 1% and 5% on the server-proxy link, where the network loss is 1% and 5% loss respectively, and about 0.01% of the packets on the proxy-client link.

These two tests shows that the network emulation is working as it should and packets are not delayed in the network more then specified.

We see in figure 5.6, that on average, packets had to wait 150 ms from they were captured, until the server sent the multiplexed packet. In the worst case the delay was 350 ms. This delay was most likely caused by buffering packets until we could fill a segment. This led us to design a second prototype.

5.4 Second prototype

Since our original idea of filling up the TCP segment before sending gave bad results, we rewrote most of the server and the receiving logic on the proxy to send packets as soon as they were captured by `libpcap`.

The program does not longer buffer the packet internally, but rather sends the packets as soon as they are read. Packets may still be buffered, but it is now done by the kernel in the TCP buffer.

got_packet is called by `libpcap` for each packet it sniffs from the loopback device. If the packet has any payload, `got_packet` copies destination and source information into a header and sends this header and the payload to the `send_packet` function.

send_packet is used to send a packet through the single connection to the proxy. It first writes the length of the buffer to the first 2 bytes of the buffer, so the receiver knows how much it must read. Then, it sends the buffer and resets the counter.

5.5 Second tests and important observations

We reran our system test, and we can see the improvements in figure 5.7 and figure 5.8. The interquartile ranges were now much closer to the baseline, but there are still things that did not work as intended.

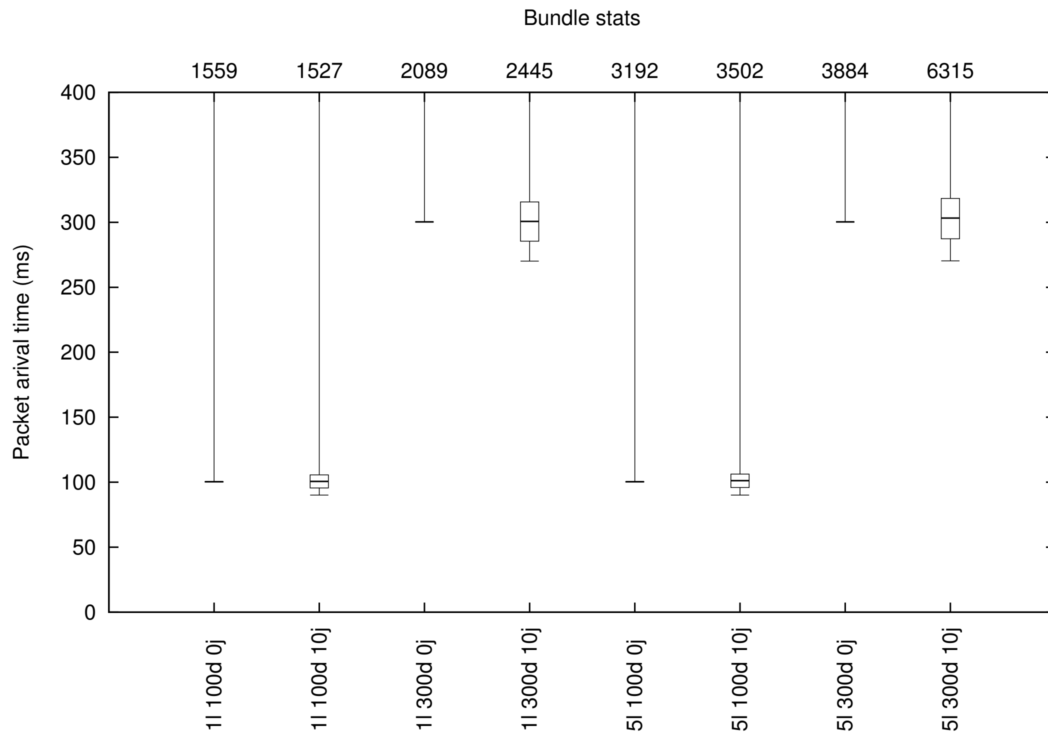


Figure 5.4: Travel times between the server and proxy

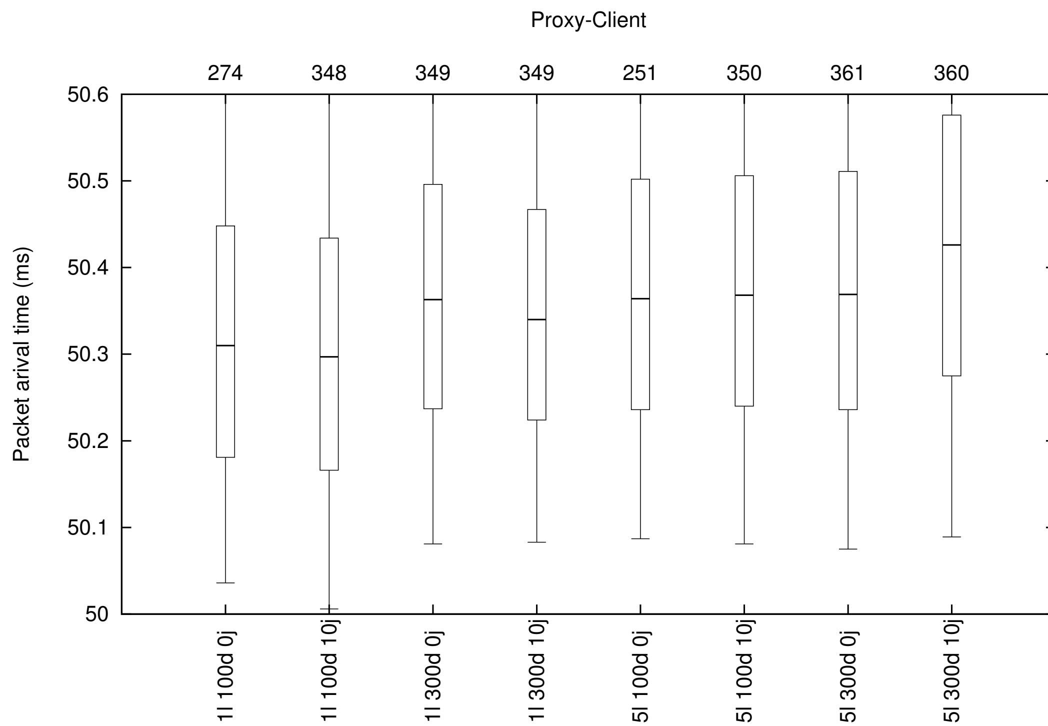


Figure 5.5: Travel times between the proxy and client

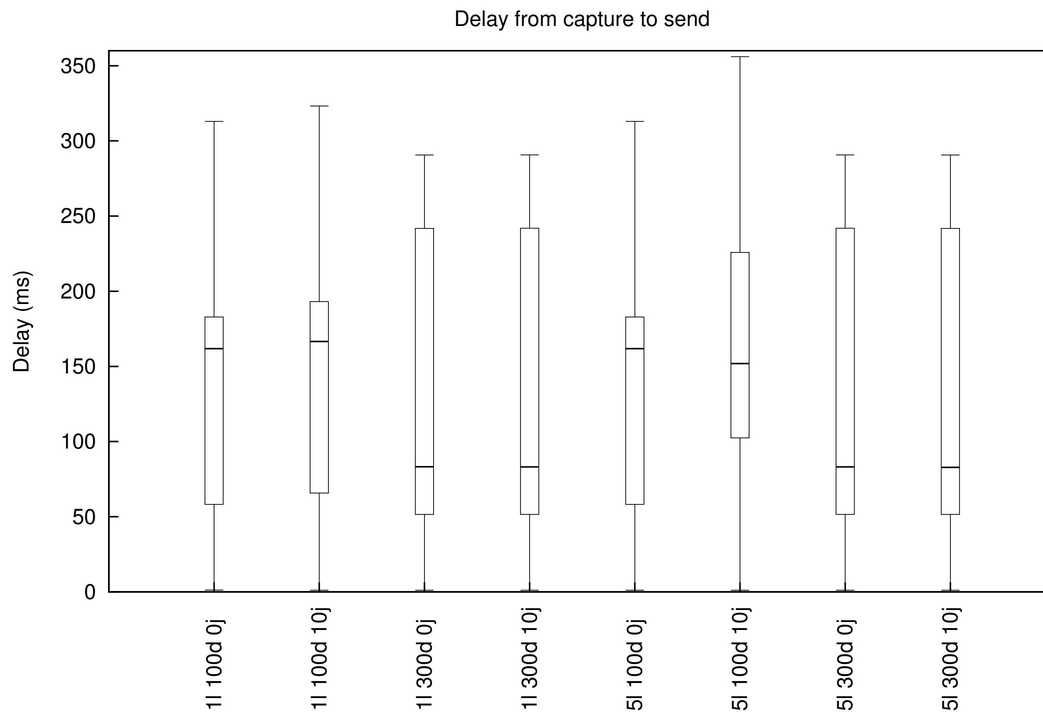


Figure 5.6: Time it takes from a packet is captured and to it is sent

We found from the comparison between the baseline test and our system test that there was a large difference in max values. We believed these were caused by setting up the connections to the client and changed the proxy code to move the code responsible for creating the client connections into its own subroutine so that we could get accurate profiling data and ran new tests. These tests showed that the ”create_connection” function used 0 time. The reason for this is that `gprof`, the profiling tool we used, cannot measure time outside of user space. We thus went for a simpler approach and made the program output the time difference between before and after calling the function, using the system’s gettimeofday function. The results of this test can be found in figure 5.9. Here, we see that although there is a noticeable delay, it does not explain the huge maximum numbers we were getting, thus we had to look elsewhere. We also measured the time it takes the proxy to send a packet to the client, and the results of this test can be seen in figure 5.10. We can see here that it uses almost no time. The maximum value seen here is the first packets for each connection, these packets have to wait for the connection to be established before they can be sent.

We also wanted to measure more accurately the time used by the server to process and send out the packets. Since each packet that `libpcap` gives us comes with the timestamp when it was captured from the link, we compared this with the current time and output it. Furthermore, we

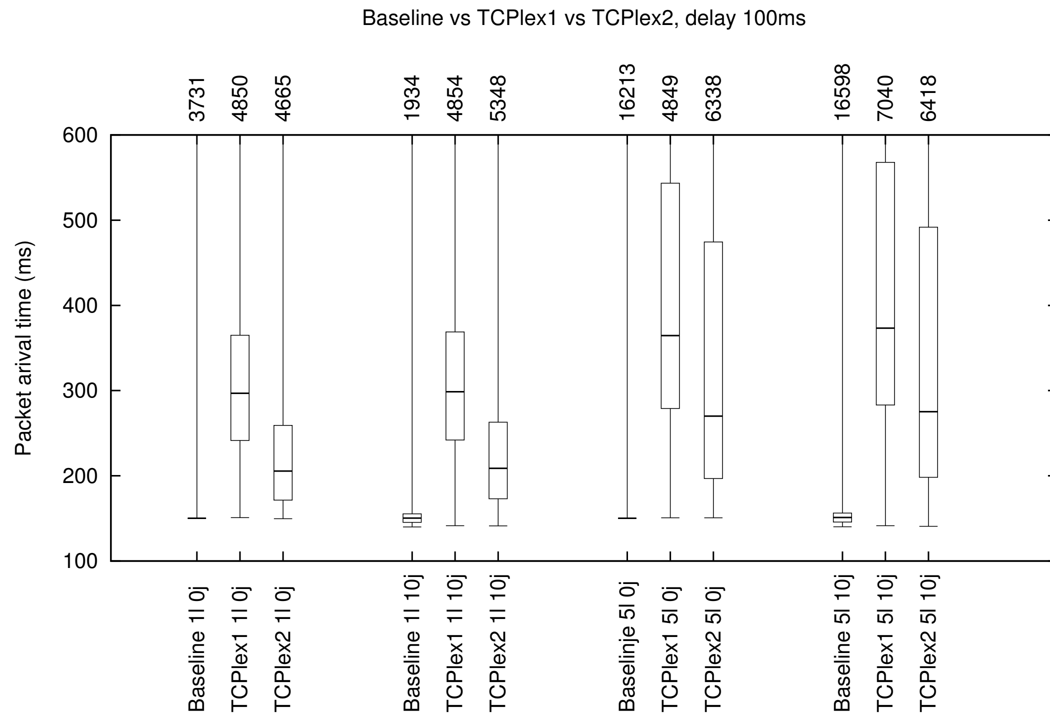


Figure 5.7: Comparison of Baseline and TCPlex2 tests with 100 ms delay

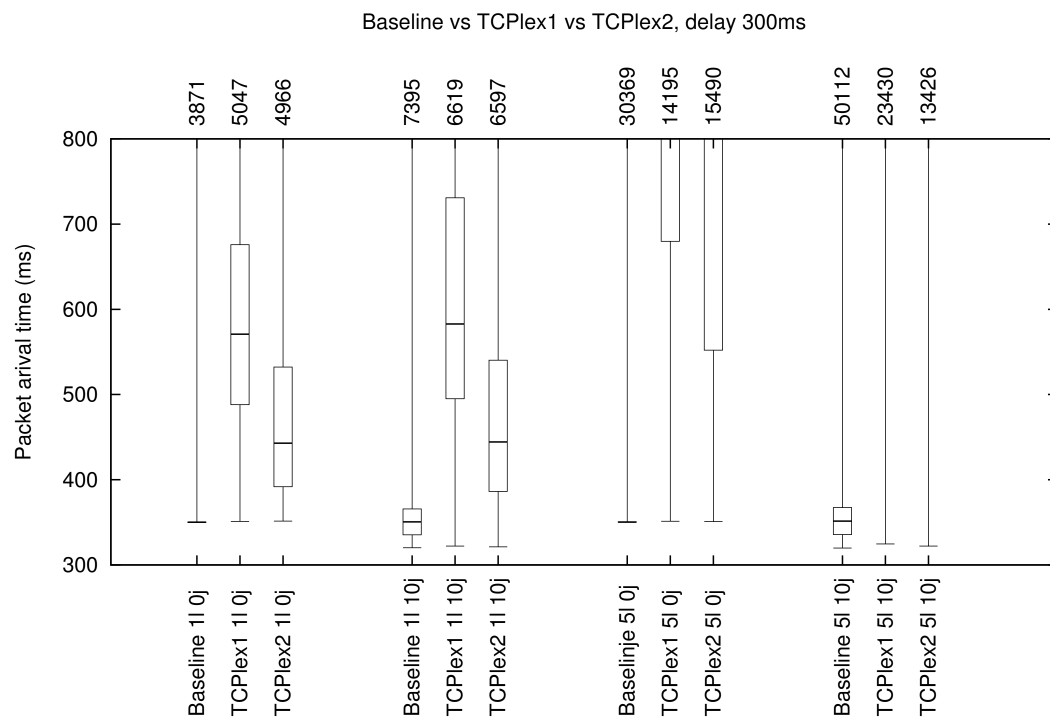


Figure 5.8: Comparison of Baseline and TCPlex2 tests with 300 ms delay

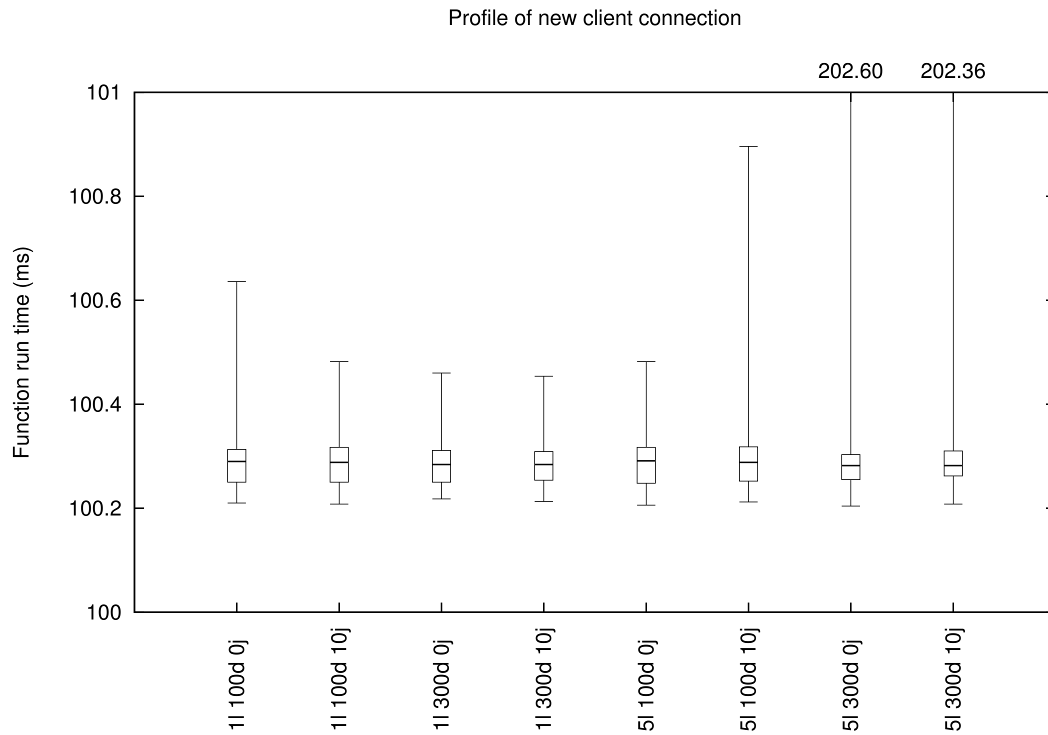


Figure 5.9: Time data for establishing new client connections, gathered from profiling the proxy

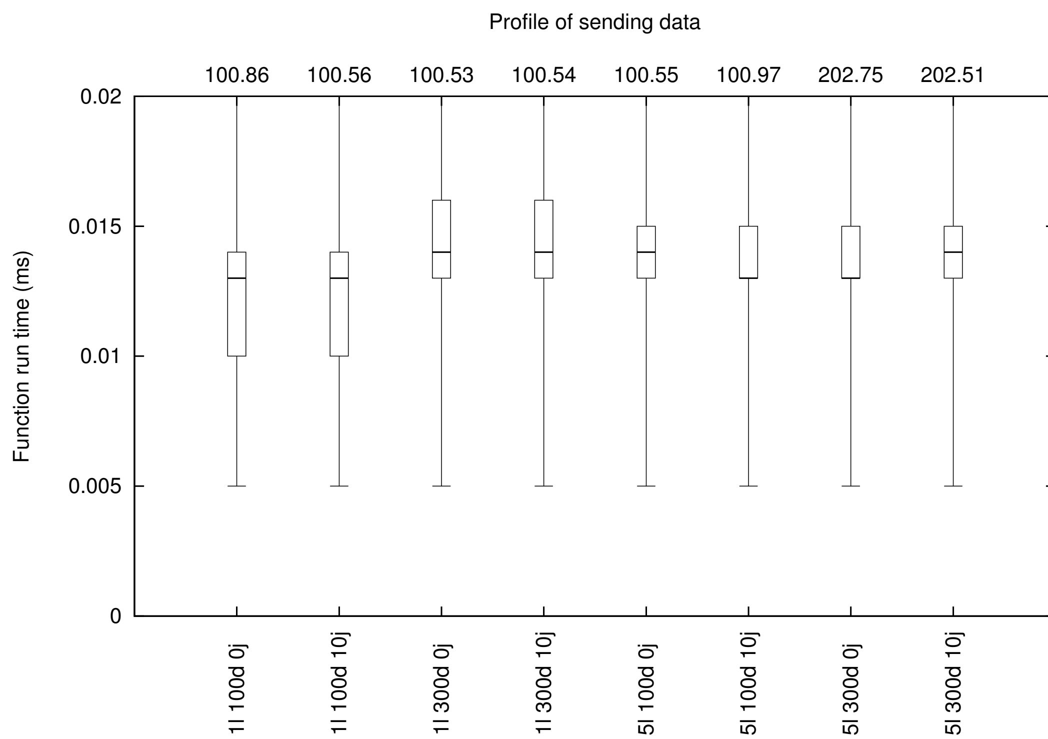


Figure 5.10: Time data for sending data, gathered from profiling the proxy

wished to see if the send call created any problems and therefore also output the time difference between when the function receives a packet from `libpcap` and after the send call is done.

We see from figure 5.11 that the delay is mostly between 0.6 ms and 0.8 ms. If we look at figure 5.12 and figure 5.13, we find the same maximum numbers that are present in figure 5.11, and by examining the profiling data, we found that the reason we get the high processing delays is that the whole program waits for the `send` call and can not process new packets from the `libpcap` buffer. Each time there is a long send delay, the next couple of packets gets the same delay while the system goes through its backlog of packets.

The reason we have to wait on `send` is that the TCP buffer in the kernel is full. This means that combined bandwidth of the multiplexed thin streams is higher than the bandwidth allowed by TCP's congestion control. We have actually created a too thick stream.

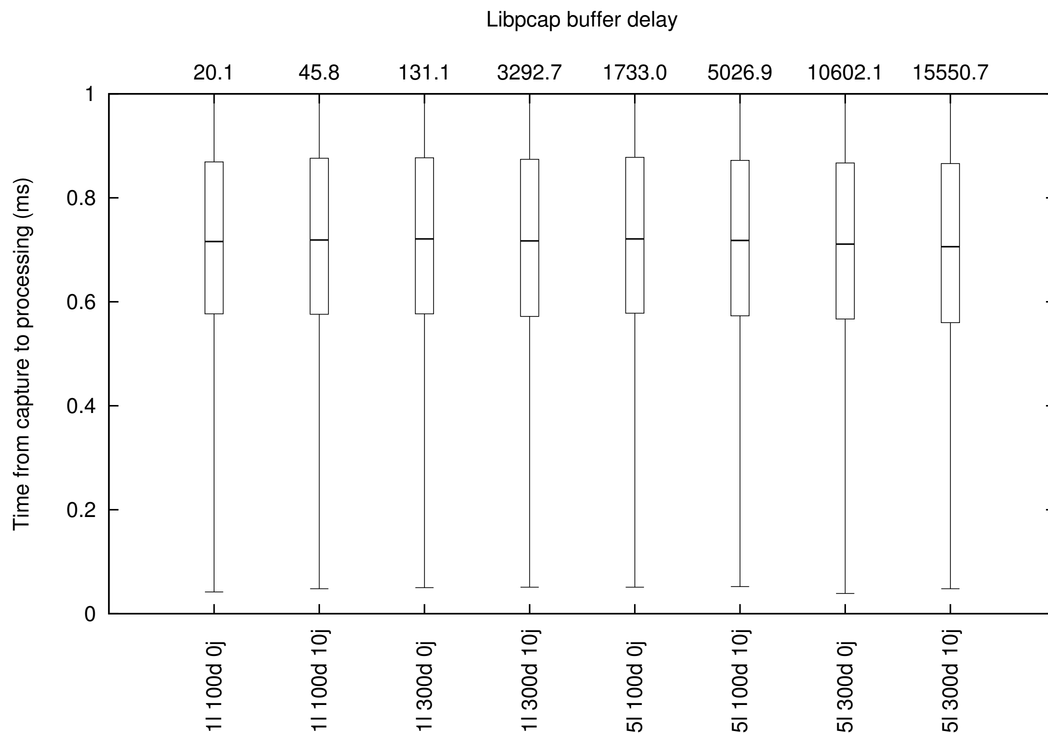


Figure 5.11: Time data gathered from profiling the server libpcap buffer delay

In figure 5.13, we can clearly see that this problem arises when the network conditions worsen, as the TCP buffer gets filled up while we wait for retransmissions. In section 5.6, we talk about a possible solution to this problem.

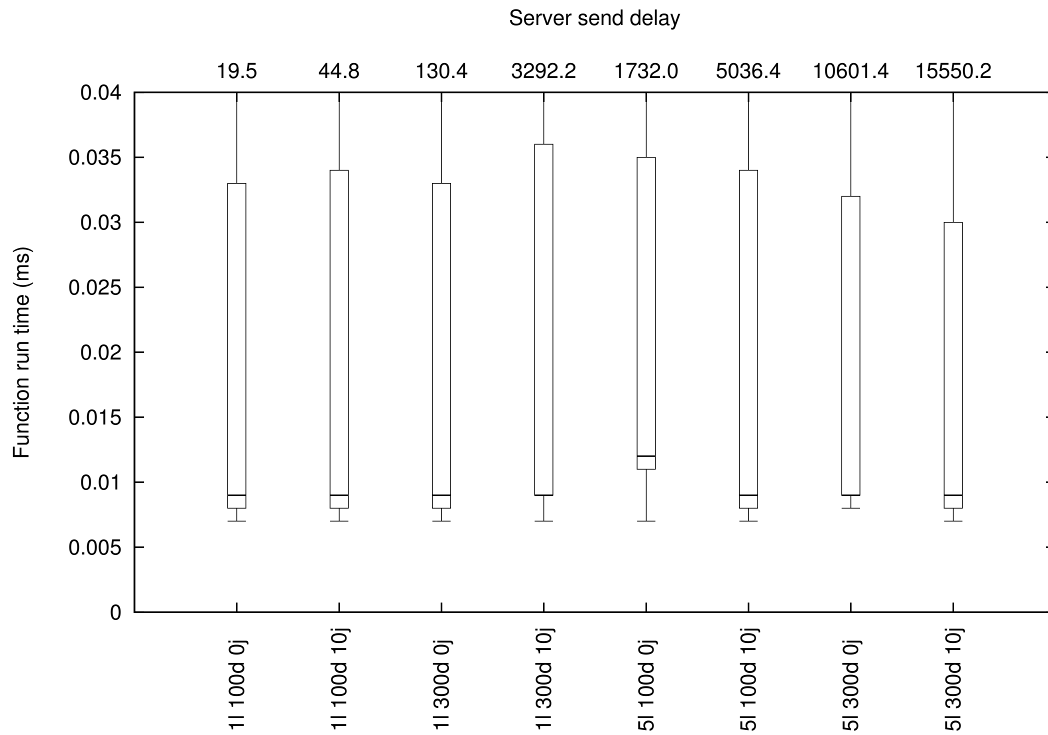


Figure 5.12: Time data gathered from profiling the server send delay

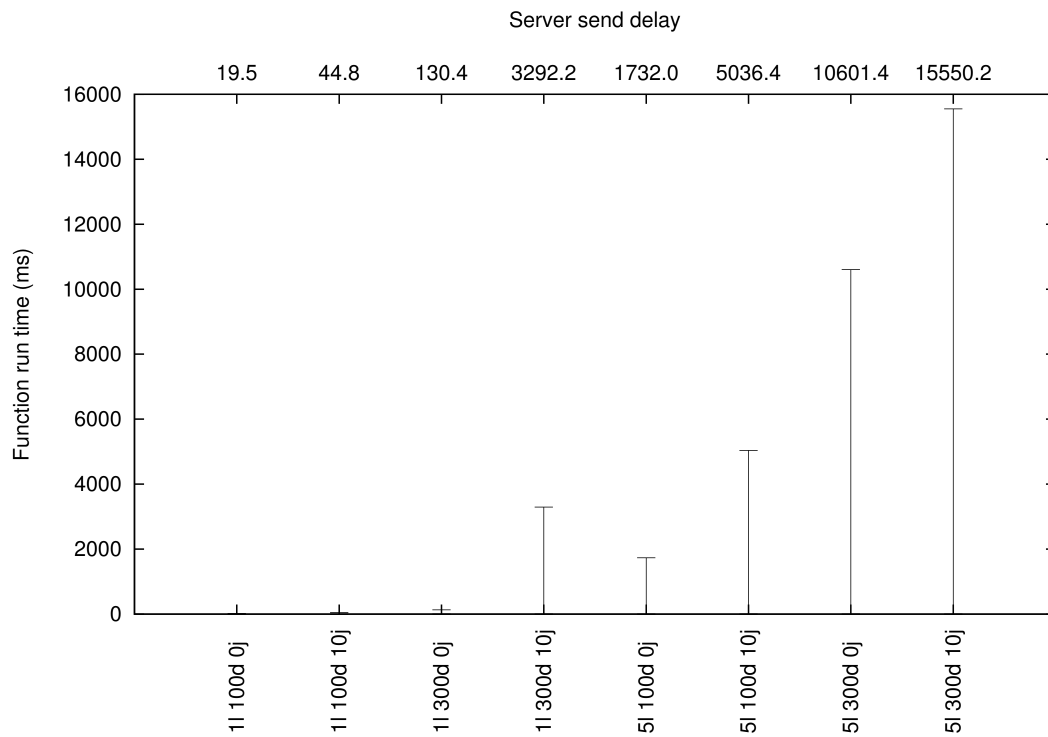


Figure 5.13: Zoomed out version of figure 5.12

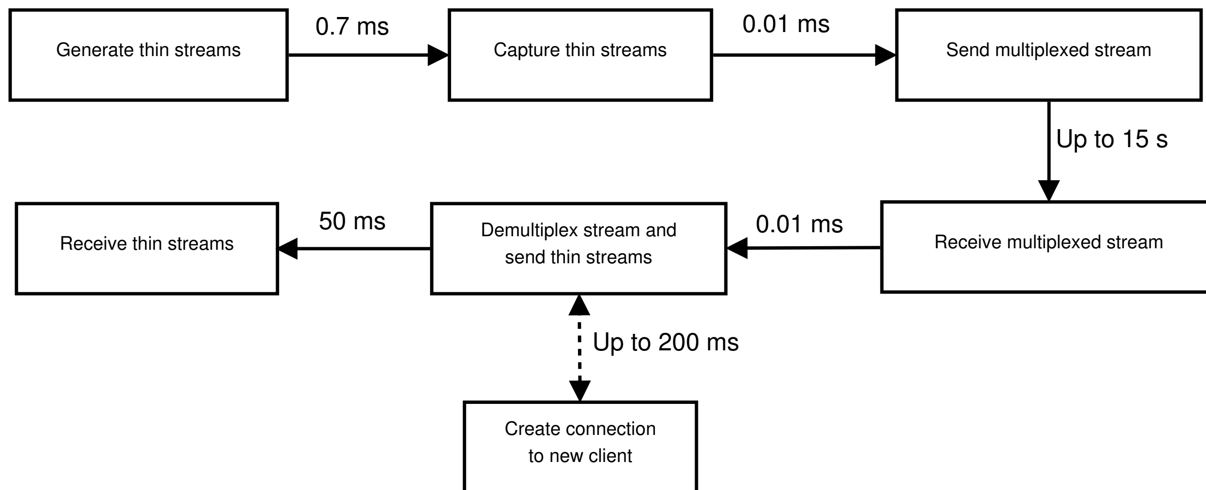


Figure 5.14: Delays found in our system.

5.5.1 Imposed delays

We have now seen that in some steps, some delay is added to a packet going through our system. These delays are shown in figure 5.14. The most notable that always applies are:

0.7 ms Libpcap buffer delay on the server, seen in figure 5.11

0.01 ms Send delay on server, seen in figure 5.12

0.01 ms Send delay on proxy, seen in figure 5.10

These delays are acceptable and also not something we can improve upon easily. There are also larger delays added under certain conditions, these are:

Up to 15 s Full TCP buffer on the server due to network conditions, as seen in both the system tests and the server profiling.

Up to 200 ms New client connections on the proxy, seen in figure 5.9

These delays are due to limitations in our system, and we therefore try to remedy the worst of these delays in the next prototype.

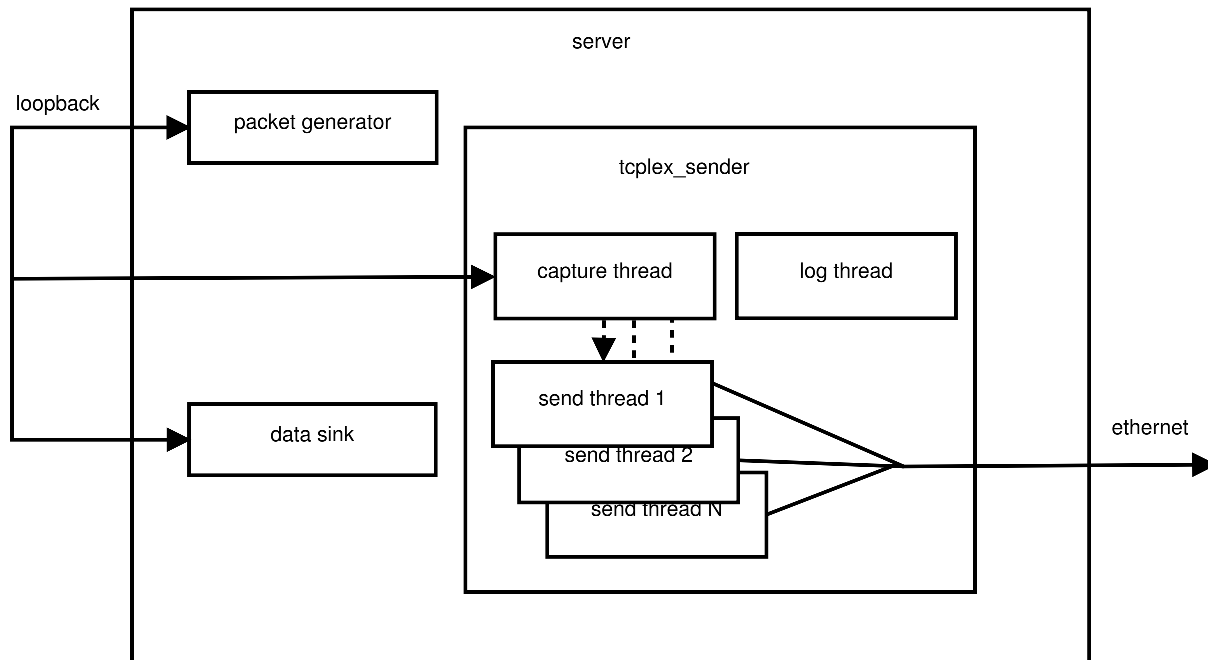


Figure 5.15: Parallel connections server

5.5.2 Observations

We have observed that large delays occur in our system due to congestion on the network and inside our program. In the next section, we try to remedy the highest of these delays. In the next prototype, we introduce parallel connections between the server and the proxy. This spreads the load on the link. We also introduce threading to minimize idle time on the server.

5.6 Parallel connections prototype

Since we found, from analysis of the second prototype, that a single TCP connection actually becomes too thick to transfer all the thin streams without congestion problems, we designed a new server program. We can see this new design in figure 5.15. Here, we see that there are now multiple connections between the server and the proxy, each running in its own thread.

The server can now be divided into three parts; The capture thread, which is responsible for capturing the thin streams from the loop back device. The send threads, which are responsible for sending all packets in their queues to the proxy. And the log thread, which is responsible for writing all log information to a designated log file. The log thread and each of the send threads are created when the program starts. The number of send threads is determined by a

parameter given to the program at startup. The number of connections to the proxy is static, and is established when the thread is created. The following functions are used by the server program:

got_packet is called by `libpcap` for each packet it captures from the loopback device. If the packet has any payload, `got_packet` copies destination and source information into a header, and a temporary buffer is assigned to this packet, where the header, the payload and the length of the payload is inserted. This buffer comes either from a list of free buffers, or if there are no free buffers, one is created for the packet. This buffer is then sent to the `send_packet` function. The capture timestamp and header is also sent to the `write_log` function.

send_packet goes through all the send queues and finds the shortest available queue, and places the buffer it got from `got_packet` into this queue. These queues are First In First Out (FIFO) queues. After the buffer is placed in a queue, `send_packet` signals the `send_packet_thread` that owns that queue.

write_log puts the capture timestamp and destination and source information into a log buffer. Then, if this buffer has reached a certain threshold, it signals the `log_thread`.

send_packet_thread constantly loops, taking out the first buffer in its queue. If the queue is empty, the thread sleeps until it is woken by a signal from `send_packet`, indicating the arrival of a new buffer in the queue. The length, header, and payload in the buffer is then sent to the proxy. The buffer the packet came in is then put back into the list of free buffers.

log_thread sleeps until it is awaken by the signal from `write_log`, it then writes the entire log buffer to a file and sleeps again.

5.7 Parallel connections tests

The number of connections and send threads is not dynamic, but specified as a parameter when starting the server. We ran a series of tests while changing this parameter each time. These test did not give the expected results. We saw that the delay actually went up when we added more connections. We assumed this is because of the hardware limitations on the server. It only has one Central Processing Unit (CPU) core, and thus cannot run more than one thread at a time.

This means that we only get the overhead from running with more than one thread, without getting any performance gain, since the threads cannot run concurrently and we have to switch back and forth between the threads all the time.

We therefore got a new machine to run the server program, this time with four CPU cores. We did not have time to run through every test on the new machine, but reran the prototype two tests, and ran two tests with the threaded server, with two connections and four connections. The results from these tests can be seen in figure 5.16 and figure 5.17. Here, TCPlex2 means the second prototype, and TCPlex3 means the third, parallel connections prototype. The ”2c” and ”4c” means, with two and four parallel connections, respectively. We see that in all the 1% loss cases, TCPlex3 has a slightly larger interquartile range and about the same maximum values as TCPlex2. If we look at the 5% loss cases, we can see a minor improvement in the interquartile ranges, and that the maximum values are about the same.

This is not what we expected. We hoped to see a improvement in the maximum values between the two prototypes. What we do see, is that the waiting we observed in TCPlex2 is due to multiple retransmissions of a single packet, and that this is not solved by adding more connections. We can also see a trend towards overall lower delays, in high loss scenarios, when we add more connections.

5.8 Summary

In this chapter, we implemented and tested many prototypes. We did a thorough examination of all the delays added throughout our system, and found ways of reducing many of them. In the next chapter, we look back at what we have learned in this thesis, and talk about the results and what still needs to be researched.

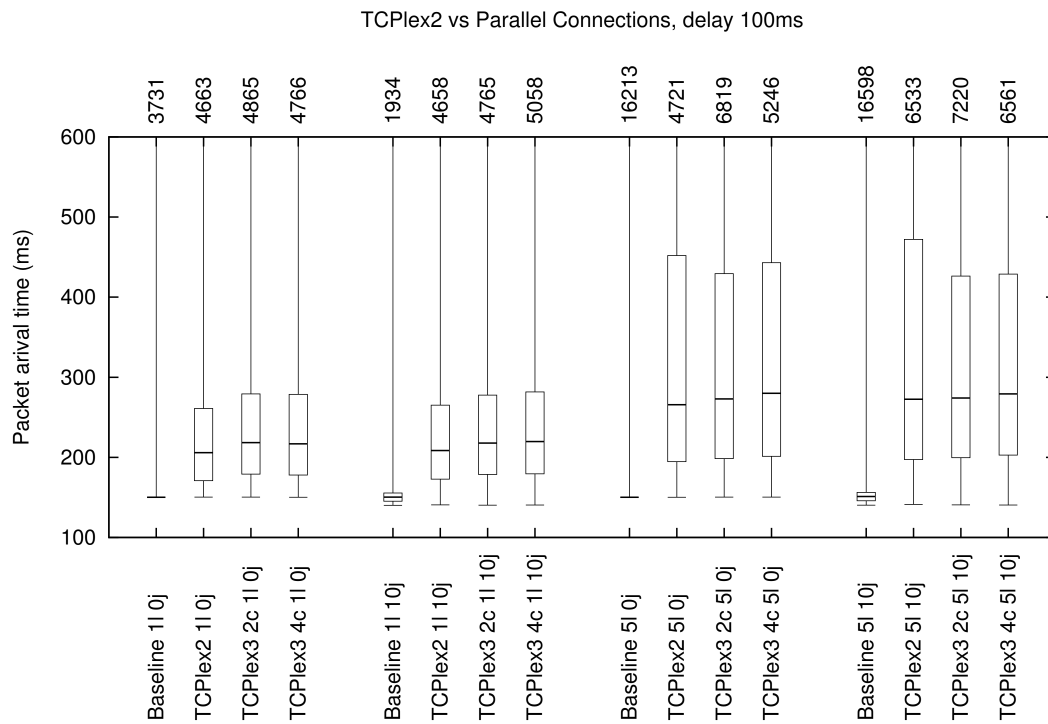


Figure 5.16: Comparison of Baseline and TCPlex3 tests with 100 ms delay

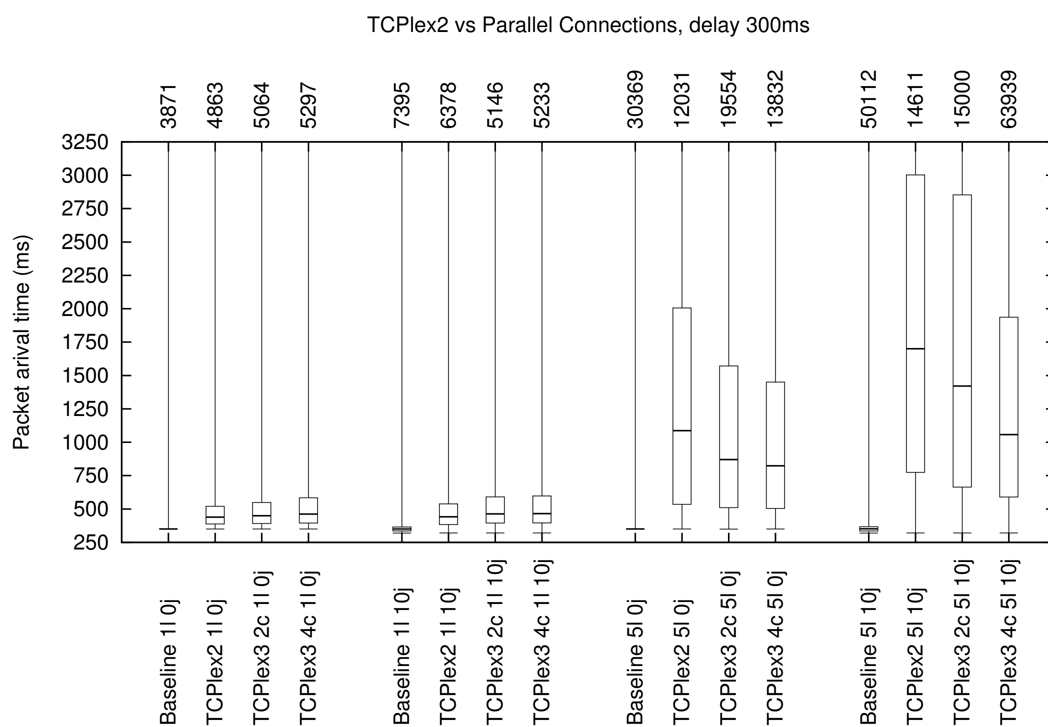


Figure 5.17: Comparison of Baseline and TCPlex3 tests with 300 ms delay

Chapter 6

Conclusion

In this thesis, we have presented our work on a transparent way to reduce the latency of a multi-user interactive thin-stream TCP application. We now summarize the work done in this thesis, look at the results, and provide an outline of possible future work.

6.1 Summary

Interactive applications often produce something we call thin streams. These data streams have small packet sizes with high IATs. When streams like this are transported over TCP, they succumb to high latencies during loss, due to retransmissions. To overcome this problem, we implemented a way to multiplex many thin streams over the unreliable network. When many thin streams all go through one TCP connection, there is a much higher probability of triggering a fast retransmit whenever a packet is lost. This reduces the delays caused by retransmissions.

Our system captures the packets of thin streams from the network, and send all these packets through one, or more, TCP connection. These streams are received by a proxy, who sends the original streams to their destination. We designed and implemented this system through an iterative process. Between each prototype, we did thorough tests and tried to reduce the delays our system imposed in the next prototype.

6.2 Contributions

We created a system for transparently multiplexing and demultiplexing packets from several thin streams. This system was intended to reduce the latency of a interactive application with many outgoing streams, by making the streams behave more like a normal TCP stream, and thus effectively using the mechanisms in TCP that help prevent high latencies.

We used a program that simulated the traffic generated by a game server with many clients. We measured statistical values for delay under several different network conditions for this program. We then did the same tests again, but with the streams created by this program going through our system.

We saw that our system imposed many different delays on the packets going through it. Several of these delays were reduced in new prototypes. If we look at the comparison of the baseline tests and the two last prototypes in figure 5.16 and figure 5.17, we can see that our approach, a multiplexing system that is transparent for the sending application, is not as successful as we had hoped. The interquartile ranges of delay are between 50% and 70% higher in our prototypes than in the baseline tests. Still, what is interesting, is that the maximum delays in our system are significantly lower than the baseline in the high loss scenarios. This can be explained by the lower IAT the streams have when multiplexed together, as this triggers fast retransmit more often.

Maximum delays are what reduces the quality and user experience the most in online gaming, thus this system has some potential. If the added delay can be some what reduced, this system can be very helpful in improving the user experience in online gaming. We talk about possible ways to improve the overall efficiency of the system in the next section.

6.3 Future work

We found that some of the delays in our system are due to the fact that we tried to make the system as transparent as possible to the sending application. A possible future expansion of our work, would be to integrate the multiplexing functions into the sending application. This will remove the capturing delay, and the sending application can also optimize transmission for a multiplexed link.

Another improvement to take a closer look at, is to make the proxy threaded. This would remove the delay when the proxy has to wait on new client connections before it can handle any other packets.

Bibliography

[1] ntpdate. man ntpdate, January 2000. 33

[2] tc. man tc, December 2001. 27

[3] An analysis of mmog subscription growth. `http://www.mmogchart.com/`, 2008. ix, 7

[4] gawk. `http://www.gnu.org/software/gawk/`, July 2009. 33

[5] netem. `http://www.linuxfoundation.org/collaborate/workgroups/networking/ne` November 2009. 24, 27, 33

[6] Gnuplot. `http://www.gnuplot.info`, September 2010. 33

[7] libpcap. `http://tcpdump.org`, April 2010. 32, 33

[8] python. `http://www.python.org`, 2010. 33

[9] Skype. `http://www.skype.com`, 2010. 9

[10] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681 (Draft Standard), September 2009. 14, 15

[11] The Entertainment Software Association. Essential facts about the computer and video game industry. `http://www.theesa.com/facts/pdfs/ESA_Essential_Facts_2010.PDF`, 2010. 7

[12] Jean-Chrysostome Bolot, Hugues CrÃ©pin, and Andres Garcia. Analysis of audio packet loss in the internet. In Thomas Little and Riccardo Gusella, editors, *Network and Operating Systems Support for Digital Audio and Video*, volume 1018 of *Lecture Notes in Computer Science*, pages 154–165. Springer Berlin / Heidelberg, 1995. 10.1007/BFb0019264. 34

[13] Jean-Chrysotome Bolot. End-to-end packet delay and loss behavior in the internet. *SIGCOMM Comput. Commun. Rev.*, 23:289–298, October 1993. 34

[14] Mark Claypool and Kajal Claypool. Latency and player actions in online games. *Commun. ACM*, 49(11):40–45, 2006. 1, 8

[15] S. Floyd, T. Henderson, and A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782 (Proposed Standard), April 2004. 16

[16] Carsten Griwodz and Pål Halvorsen. The fun of using TCP for an MMORPG. pages 1–7, May 2006. 2, 22, 34

[17] Mahbub Hassan and Danilkin Fiodor Alekseevich. Variable packet size of ip packets for voip transmission. In *IMSA'06: Proceedings of the 24th IASTED international conference on Internet and multimedia systems and applications*, pages 136–141, Anaheim, CA, USA, 2006. ACTA Press. 9

[18] G. Huston. Next Steps for the IP QoS Architecture. RFC 2990 (Informational), November 2000. 12

[19] International Telecommunication Union (ITU-T). One-way transmission time, itu-t recommendation g.114. `www.itu.int/itudoc/itu-t/aap/sg12aap/history/g.114/g114.html`, 2003. 1, 9

[20] P. Karn and C. Partridge. Improving round-trip time estimates in reliable transport protocols. *SIGCOMM Comput. Commun. Rev.*, 17:2–7, August 1987. 16

[21] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), March 2006. Updated by RFCs 5595, 5596. 12

[22] Chris Majewski, Carsten Griwodz, and Pål Halvorsen. Translating latency requirements into resource requirements for game traffic. July 2006. 2, 22

[23] Athina Markopoulou, Fouad Tobagi, and Mansour Karam. Loss and delay measurements of internet backbones. *Comput. Commun.*, 29:1590–1604, June 2006. 34

[24] J. Nagle. Congestion Control in IP/TCP Internetworks. RFC 896, January 1984. 17

[25] V. Paxson and M. Allman. Computing TCP's Retransmission Timer. RFC 2988 (Proposed Standard), November 2000. 16

[26] Andreas Petlund. *Improving latency for interactive, thin-stream applications over reliable transport*. PhD thesis, University of Oslo, 2009. xi, 3, 6, 8, 23

[27] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980. 1, 12

[28] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168. 1, 12

[29] R. Stewart. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), September 2007. 2, 12

Appendix A

List of abbreviations

ACK	Acknowledgement
AIMD	"Additive Increase, Multiplicative Decrease"
CPU	Central Processing Unit
CWND	Congestion Window
DCCP	Datagram Congestion Control Protocol
dupACK	Duplicate Acknowledgment
FIFO	First In First Out
FPS	First-Person Shooter
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
IAT	Interarrival Time
IP	Internet Protocol
ISP	Internet Service Provider
ITU-T	International Telecommunication Union
MMOG	Massive Multiplayer Online Game
MSS	Maximum Segment Size

MTU Maximum Transmission Unit

QoS Quality of Service

RWND Receiver's advertised Window

RDC Remote Desktop Connection

RPG role-playing game

RTO Retransmission Timeout

RTS real-time strategy

RTT Round-Trip Time

RTTVAR Round-Trip Time Variation

SCTP Stream Control Transmission Protocol

SMSS Sender Maximum Message Size

SRTT Smoothed Round-Trip Time

SSH Secure Shell

ssthresh slow start threshold

TCP Transport Control Protocol

UDP User Datagram Protocol

VNC Virtual Network Computing

VoIP Voice over IP