UNIVERSITY OF OSLO
Department of Informatics

Improving TCP for time-dependent applications

Master thesis

Kristian R. Evensen

Contents

List of Figures

List of Tables

Acknowledgement

I would like to thank my supervisors, Andreas Petlund, Pål Halvorsen, Carsten Griwodz, for a lot of help, comments and useful discussions. I would also like to thank Paul Beskow, Tonje Fredrikson, Håkon Kvale Stensland and Håvard Espeland for taking the time to read through the thesis and provide feedback. Finally, to the guys at the lab, thank you for the moral support.

Abstract

In the last couple of years, developers have started implementing more complex features utilizing computer networks in interactive applications, for example, advanced and large scale multiplayer modes in games. In addition, we have seen applications using computer communication in new ways, like IP telephony and video conference systems. Due to their interactive nature, these applications have strict latency requirements. Also, several of them have *thin stream* characteristics, meaning that they have very small bandwidth requirements.

For applications to communicate over a computer network, they have to use a transport protocol. The dominant ones today are TCP and UDP, with TCP being preferred by the network because of the lack of fairness mechanisms in UDP. Also, UDP is more likely to be blocked by firewalls, and developers have to implement mechanisms that enforces e.g., reliability when using UDP. Unfortunately, TCP and its retransmission mechanisms are tuned for high-throughput streams without any timeliness requirements, for example streams generated by applications doing bulk data transfer. Interactive applications will often suffer from unnecessary high latencies due to these mechanisms, which can be devastating for the user experience.

To address the latency issues, we have developed a sender side TCP modification that bundles potentially lost data into packets that are to be sent or retransmitted. By doing this we hope to preempt the experience of packet loss and improve the user experience. We have implemented and tested this modification in the Linux kernel, and our results show that we reduce the application latency by trading it against bandwidth.

Chapter 1

Introduction

1.1 Background

In the last couple of years, developers have started implementing more complex features utilizing computer networks in interactive applications, for example, advanced and large scale multiplayer modes in games. We have also seen applications using computer communication in new ways, like IP telephony and video conference systems. Due to their interactive nature, these applications have strict latency requirements. In [3], it is shown that the required latency for a first person shooter is approximately 100 ms, 500 ms for role playing games, and 1000 ms for real time strategy games. Audio conferencing and IP-telephony require the latency to stay below 150-200 ms to achieve a satisfactory user experience, and below 400 ms to remain usable [4].

Many interactive applications have *thin stream* characteristics. In this context, a stream is considered *thin* if the application generates data in such a way that: a) The packet interarrival times are so high that the transport protocol's fast retransmission mechanisms are ineffective, or b) the size of most packets is well below the Maximum Segment Size (the largest packet a network can transport, MSS). An example of a time-dependent thin stream application (because of the latency requirements), is the traffic generated by the massive multiplayer online role-playing game (MMORPG) Anarchy Online. When we captured network traffic from this game, we saw an average payload size of 93 bytes and an average interarrival time of 580 ms (IAT, the time that passes between every sent packet). This type of game usually consists of thousands of players interacting in real time. As such, high latency can be devastating for the user experience. The player could be killed because the data arrives too late for him to react in time, or lose out on a big reward because one other player received the data earlier and got a head start. Another example of a time-dependent thin stream-application is IP telephony, where a delay in the delivery of data will reduce the perceived quality of the conversation. When we had a conversation using the popular IP telephone Skype over the internet, we got an average payload of 110 bytes and IAT of 24 ms.

For these applications to communicate over a computer network, they have to use a transport protocol. The currently dominant ones are the Transport Control Protocol (TCP) [5] and the User Datagram Protocol (UDP) [6], and interactive applications may prefer UDP because of it's simplicity. UDP's lack of reliability and congestion control suits interactive applications well, there are no mechanisms that will reduce the send rate (thus increasing latency) and the application itself decides what to do when data is lost or delayed. For example, an application that streams audio can choose to drop packets that arrive too late, thus ensuring smooth playback with some audio corruption. Despite this, UDP has several drawbacks. It can send at an uncontrolled data rate that pushes other traffic out of the network. Additionally, it is often blocked by firewalls and also UDP's lack of reliability is not always positive. Several interactive applications require reliability - for example, if you are playing a strategy game, you want your orders to arrive in the correct order. The application developers will have to implement reliability themselves, which will take time and make the application more complex. UDP-middleware like UDT [7] tries to solve some of the issues related to fairness and reliability, but they are not able to overcome the firewall issue.

For these reasons, many interactive applications use TCP; fewer firewalls block the protocol, and it is fair[1] to other streams. Also, TCP offers reliable communication, so the developers do not have to implement this as application layer functionality. However, TCP's flow and congestion control mechanisms are tuned for non time-critical and high-throughput streams, and will often contribute to an increased latency. Most TCP variations assume that packet loss is due to congestion, and will reduce the send rate for every retransmission. In addition, if a packet is lost, the receiver will have to wait for it to be retransmitted (and arrive) before more data can be delivered to the application. Retransmissions are either triggered when a timer expires, which is known as a retransmission timeout (RTO), or after receiving multiple acknowledgments of the last packet which was received in order. The receiver in a TCP connection acknowledges (ACK)

[1]The stream will not consume more than its fair share of the bandwidth, discussed in section 2.2.2.

all data it has received in-order, and if a packet is lost, packets will arrive out of order and the receiver will instead send a copy of the ACK for the last packet that arrived in order (known as a duplicate ACK, dupACK). After the sender receives a given number of dupACKs, a fast retransmit is triggered. In a thin stream scenario, generating the required number of dupACKs might take a while, and the period while waiting for a retransmission will increase the latency even further. For example, in Linux, the minimum value for the RTO timer is 200 ms.

Optimizing TCP with respect to thin streams will therefore improve the experience for users with lossy or congested links to the sender. In addition, when applications use TCP instead of UDP, the bandwidth is shared fairly.

1.2 Problem definition

TCP and it's congestion control mechanisms are tuned for applications that want to transmit non time-critical data through a network as fast as possible, like those doing bulk data transfer. In this scenario, it is not important when a packet arrives, but that it arrives eventually. Thus, any lost packet can be retransmitted later. However, in a time-dependent thin stream scenario, the perceived user experience will suffer if the application has to wait for retransmissions. In other words, TCP's congestion control mechanisms are not suited for this type of traffic and will contribute to an increased latency. The minimum RTO timer value is often so high that it exceeds the latency requirements for some applications, and thin streams with a high IAT will not be able to generate the required number of dupACKs for a fast retransmission.

As such, our goal is to find a way to improve the latency for time-dependent thin streams, something we intend to accomplish by making modifications to TCP. To ensure compatibility with a range of operating systems, and to be compatible with existing applications, any modifications we make need to be transparent and compliant with existing standards.

1.3 Research method

Our latency reduction mechanisms for TCP are implemented and experimentally evaluated. With Redundant Data Bundling (RDB), which will be presented in detail in this thesis, we first spent a lot of time designing and discussing different aspects of it. Then we implemented RDB in the Linux 2.6.22.1-kernel, followed by several experiments to determine if it had any effects. We have also conducted a user survey to see if our modifications improve the user experience.

1.4 Main contributions

RDB, the TCP modification that will be presented in this thesis, aims to improve the latency for time-dependent thin streams. It was inspired by the piggybacking techniques presented in [8], and functions by bundling old data in new packets. As such, we increase the chance of having potentially lost data arrive at the receiver before TCP performs a retransmit, thus reducing the latency and in many cases improving the user experience.

The most important reason for implementing RDB as a sender side modification, is that it remains transparent to the receiver. This has many benefits, as no extra requirements (like patching the OS kernel) will be imposed on the user, and only the sender machine(s) has to be updated. In addition, by implementing RDB in the Linux-kernel, it is made transparent to the application. To enable RDB, the applications either has to set a socket option (which has to be done by the application and is not fully transparent) or enable a global system variable (completely transparent for the application).

RDB significantly reduces the latency for many time-dependent thin streams, both at the transport layer (when the data is delivered to the application) and the application layer (when the application can use the data). Also, it improves the user experience for several applications generating time-dependent thin streams. Unfortunately, RDB causes the bandwidth consumption to increase. The bundled packets will often be significantly larger than those sent by standard TCP.

1.5 Outline

The document is organized as follows. Chapter 2 goes into detail on interactive applications, the network traffic they generate and TCP, and chapter 3 describes RDB. Chapter 4 contains test results and an evaluation of the proposed modification. Finally we summarize our findings and make suggestions for future work in chapter 5.

Chapter 2

TCP performance for interactive thin streams

Interactive applications are applications where the quality of the user experience depends on real-time input/output. Examples are games, stock applications and video conferences. These applications' network functionality generates time-dependent traffic. If you play a game, delayed data might cause your avatar to be killed.

Normally, UDP is best suited for these applications. It has no congestion control and thereby no mechanisms that will affect the latency. Its lack of reliability allows the applications to handle lost or late packets. Video streaming software can drop packets to ensure a smooth playback (with some corruption), or games might do estimation to compensate for lost data regarding the movement of a character. However, using UDP is often not ideal or possible. For example, UDP is not considerate to other streams (it can potentially consume the entire bandwidth of a link), and it is often blocked by firewalls.

TCP, on the other hand, is fair to the network (what this means is presented in section 2.2.2) and more often let through firewalls. The development of TCP has focused on improving the performance of throughput-intensive streams, while a lot of interactive applications generate time-dependent thin streams. Many interactive applications consume very little bandwidth; they tend to send short packets with high IAT. We have decided to call this type of low bandwidth traffic thin streams, and we will focus on those thin streams that are time-dependent. Other thin streams have no extra requirements compared to the streams TCP is already tuned for.

In this chapter, we will look at the network characteristics of several applications that generate time-dependent thin streams. We will then present

			payload size			packet interarrival time (ms)						avg. bandwidth	
	application	prot-	(bytes)							percentiles		requirement	
		ocol	avg.	min	max	avg.	median	min	max	1%	99%	(pps)	(bps)
1	Anarchy Online	TCP	98	8	1333	632	449	7	17032	83	4195	1.582	2168
2	BZFlag	TCP	30	4	1448	24	0	0	540	0	151	41.667	31370
3	Halo 3 - 8 players	UDP	247	32	1264	36	33	0	1403	32	182	27.778	60223
4	Halo 3 - 6 players	UDP	270	32	280	67	66	32	716	64	69	14.925	35888
5	Test Drive Umlimited	UDP	80	34	104	40	33	0	298	0	158	25.000	22912
6	Tony Hawk's Project 8	UDP	90	32	576	308	163	0	4070	53	2332	3.247	5812
7	World of Warcraft	TCP	26	6	1228	314	133	0	14855	0	3785	3.185	2046
8	Casa	TCP	175	93	572	7287	307	305	29898	305	29898	0.137	269
9	Windows remote desktop	TCP	111	8	1417	318	159	1	12254	2	3892	3.145	4497
10	Skype (2 users)	UDP	111	11	316	30	24	0	20015	18	44	33.333	37906
11	Skype (2 users)	TCP	236	14	1267	34	40	0	1671	4	80	29.412	69296
12	SSH text session	TCP	48	16	752	323	159	0	76610	32	3616	3.096	2825
13	Bulk data transfer	TCP	1335	29	1460	10	0	0	1498	0	391	100.000	1247552

Table 2.1: Examples of thin stream packet statistics based on analysis of packet traces. Also included is one "regular" stream, bulk data transfer.

TCP, before we look at how altering different parameters (like the link's loss rate) will affect a TCP connection.

2.1 Interactive thin stream applications

In interactive applications, the quality of the user experience depends on real-time input/output. Several of these applications provide features that utilize computer networks, and we will now look at the traffic generated by different types of interactive applications.

2.1.1 Games

Games are one of the oldest forms of interactive applications, and feature more and more complex multiplayer modes and other network functionality. In the last couple of years, we have gone from small-scale battles, and to MMORPGS with thousands of players interacting with each other. The seven first applications in table 2.1 are all games, and most of them use UDP. We believe that their traffic pattern would be very similar if they used TCP instead, and have therefore chosen to include them.

We can see that the different types of games generate very different traffic. Most of the traffic from intensive games, like the first person shooters BZFlag and Halo 3, has low IATs (compared to the rest). On the other hand, World of Warcraft and Anarchy Online, two large MMORPGs, have a higher IAT. This is because they have a slower pace and there are fewer packets to send. For example, in a shooter, the player will move around and shoot almost constantly, thus generating several packets. When a player in one of the two MMORPGs attacks a monster, the player will only intervene when he or she is going to cast a spell, use a special attack, or similar. In other words, less packets will be sent because the time interval between events is longer.

Independent of the type of game, we see that on average, almost all the games send small packets (compared to bulk data transfer and most of the other interactive applications). This is most likely caused by the fact that not every message has to contain much information. For example, we believe that when issuing an attack command in an MMORPG, all the server needs to know is which attack to use on what enemy. The reason for Halo 3's large packets (compared to the rest of the games), is most likely that the Voice over IP (VoIP) feature was enabled. Speech consists of a lot of data, and will thus occupy more space in a packet than e.g. a command.

The latency requirement, however, varies depending on the type of game. In [3], it is shown that to ensure a good user experience, the latency can not exceed 100 ms for a first person shooter (like Halo 3), 500 ms for role-playing games (like World of Warcraft and Anarchy Online), and 1000 ms for real time strategy games.

2.1.2 Audio conferences

Audio conferencing (VoIP) is another example of applications that generate time-dependent thin streams, and they are becoming increasingly common. IP telephones are reaching the mass market, and many games allow the users to audio-chat with each other. Many of the VoIP telephone systems use one of the G.7xx formats recommended by ITU-T [1]. For example, G.711 and G.729 have bandwidth requirements of 64 and 8 Kbps respectively, and the packet size is determined by the packet transmission cycle. This is typically a few tens of ms, resulting in packets that are between 80 and 320 bytes for G.711 [9].

The traffic generated by Skype, a popular VoIP-software, has similar characteristics as those described in the last paragraph. In table 2.1 we see that the packets are small (on average 236 bytes using TCP) and the bandwidth requirement is low (69 Kbps). To avoid that the user experience suffers, ITU-T has specified that the latency (one way) should not exceed 150-200 ms, and it must not exceed 400 ms [4].

[1]http://www.itu.int/ITU-T/

2.1.3 Remote login

Remote login is when you log into and work on a machine remotely. For this study, we have measured an SSH text session and Windows remote desktop. As shown in table 2.1, they generate very similar traffic. The packets are small and the average IAT high, which is caused by the fact that there is no need to send data unless the user interacts with the machine. And it will often take at least tens or hundreds of ms before a user hits a new key. According to a study presented in [10], the average computer user types 33 words per minute on average (when copying from a transcript). If every word is five characters long, the user will type an average of 2.75 characters per second, thus 360 ms will pass between each letter.

2.1.4 Discussion

For an interactive application to send data, an event (in the application) must occur that is of interest to the other parties involved. For example when somebody says something into an IP telephone, the speech must be transferred to the receivers. Or if somebody plays a game and moves around, both the server and the players in the vicinity must be notified about his or her new position. Otherwise, their game clients will not give a correct view of the game world. On the other hand, if everybody is standing still and nobody talks, no data needs to be transferred (except maybe small updates to let others know that the connection is still alive).

This is quite different from the streams that TCP is tuned for, like the ones generated by bulk data transfer (e.g. downloading a file). They are only concerned with throughput and send as much data as possible through the network. TCP's reliability ensures that lost packets will be retransmitted, and that the data will be delivered to the application in order. Since these streams have no timeliness requirements (a file is for instance only useful once all data has been received), the delays imposed by waiting for retransmission will not be critical.

In table 2.1 we present the measured network traffic of several interactive applications, as well as one application generating more typical TCP traffic. The reliance on something to happen before data is sent leads to very distinct traffic patterns. Compared to bulk data transfer, thin streams have a high IAT and small packet sizes. The latter is because these applications mostly send very short messages to each other. For instance, position updates do not have to contain much more than a new (x,y,z) co-ordinate. In addition, the applications need to send data almost instantly after the events occur. E.g. an IP telephone can not store speech in a buffer and wait until it contains a certain number of bytes, or a timeout occurs before sending the data. If it does, the delayed delivery will reduce the quality of the conversation; it will take a while before the other parties receive what the sender said, thus delaying their reply.

We have decided to call these streams, identified by small packets and/or high IATs, thin streams. In addition to these requirements, thin streams often face strict latency requirements, like those imposed by many interactive applications. It is worth mentioning that not all interactive applications generate thin streams, e.g. video conferences and video-/high quality sound streaming generates packets that are as large as the network is able to transfer. For example, in Ethernet, the MSS is 1500 bytes [11]. Thus, if a connection uses TCP, the payload can be up to 1448 bytes large (the TCP and IP header occupy 52 bytes).

2.2 TCP

In the TCP/IP reference model [12], the transport layer is the second highest layer. It provides transparent data transfer between end nodes, i.e. the applications do not have to worry about how the data is sent or received. A transport layer protocol is a protocol that is located at this layer, and is responsible for turning the simple services provided by the network layer into something more complex. IP (the TCP/IP network layer) only deals with the actual moving of data through the network. For example, it has no way of knowing if a sent packet arrives, or if the payload has been tampered with.

TCP [5] is one of the most commonly used transport layer protocols. SMTP (e-mail), HTTP (web) and FTP (file transfers) are just some of the application layer protocols that builds on top of it, and it is frequently used by games and for streaming various media. The protocol has many desirable and advanced features:

- **Connection-oriented** - a connection has to be established before data can be transferred.

- **Stream-oriented** - the application can send a continuous stream of data for transmission, TCP is responsible for dividing it into suitable units for the network layer to process.

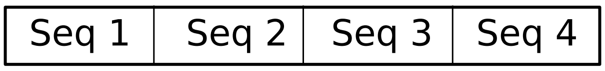
- **Reliable** - all sent data will arrive and be delivered in order to the application. In addition, TCP uses checksums to detect (and reject) corrupted packets.

- **Flow control** - throughout the connection, the receiver keep the sender updated on how many packets he or she is able to receive. The sender has to adjust the packet send rate to avoid exceeding this limit, otherwise the receiver would be overwhelmed with packets.

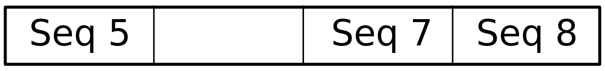
- **Congestion control** - to stop the sender from consuming so much bandwidth that it would affect the performance of other streams, TCP limits the packet send rate. In addition, TCP assumes that all packet loss is caused by congestion, and reduces the send rate when loss occurs.

One of TCP's goals is to be considerate to other streams sharing the same link. TCP assumes that all packet loss is due to congestion, and as long as no packets are lost the sending rate is increased. When congestion occurs, the protocol follows an "Additive Increase, Multiplicative Decrease" scheme (AIMD) to adjust the sending rate. The name of the scheme implies that the sending rate increases linearly and decreases quadratically.

To ensure reliability, each TCP packet is marked with a sequence number. This number is the byte offset for the packet's payload (the data contained in the packet) in the file/stream that is transferred. To let the sender know that the data is received, the receiver sends an ACK-packet containing the next expected sequence number. In other words, the receiver lets the sender know that it has received all bytes up to this sequence number. Should a packet arrive out of order (i.e., the sequence number is higher than the expected one), the receiver sends a dupACK. Exactly what these are used for, and how the sender reacts to them will be discussed in the next section.



(a) All packets arrive in order and is delivered to the application.



(b) The packet with sequence number 6 is lost. The two last packets cannot be delivered to application before the lost packet is retransmitted (and received).

Figure 2.1: Examples of a TCP receive buffer.

If a packet arrives out of order, it will be buffered (stored) at the receiver until the expected packet(s) arrive (figure 2.1). The size of the receiver's advertised window (`rwnd`) says how much outstanding data the receiver is able to store, and every ACK contains it's size. The sender has to adjust the send rate accordingly, and this is the flow control [13]. At the sender, the congestion window (`cwnd`) determines the amount of data that can be sent before receiving an ACK. The `cwnd` and `rwnd` change throughout the connection, and the lowest of the two decide the transfer rate. RFC2581 [13] states that TCP is never allowed to send data with a sequence number higher than the sum of the highest acknowledged sequence number, and the minimum of `cwnd` and `rwnd`. In other words, the size of the payload can never exceed the size of any of the two windows.

2.2.1 TCP congestion control

Congestion control is concerned with how much network resources senders are allowed to consume, and it's goal is to avoid what is known as a congestive collapse. This is a condition where the network is constantly overloaded, thus the delays will be long, the loss rate high and the throughput low. A large number of TCP protocol variations have been developed to cater to different scenarios (very high speed links, wireless, and so on), and what they alter is mostly related to the congestion control. For example, TCP Westwood [2] [14] [15] [16] reduces the sending rate in a way that is specifically tuned for WLANs. In these networks, most of the loss is caused by corruption, and not congestion, as TCP originally assumed. Unfortunately, so far none of the variations are optimized for thin streams. The different congestion control mechanisms are tuned for streams with very low IAT's and no timeliness requirements. TCP is mostly used together with throughput-intensive applications like bulk data transfer, where latency is not an issue.

In [1], C. Griwodz and P. Halvorsen investigated how the different TCP variations behave when faced with a typical thin stream. They measured the delay between the first retransmission and the delivery of the packet, and they found out that the different TCP protocol variations performed about the same. Still, TCP New Reno [17] was marginally better in most cases. For this reason, we have used TCP New Reno in our tests, and the rest of this section will focus on New Reno and the original Reno. New Reno builds upon TCP Reno [13] and improves some of the original's mechanisms. We will also present TCP SACK, which is a TCP option

[2] http://www.cs.ucla.edu/NRL/hpi/tcpw/

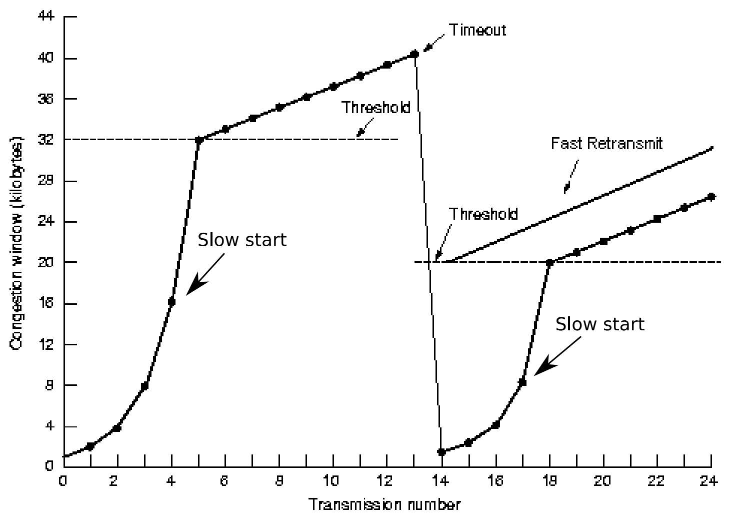


Figure 2.2: Slow start, retransmission timeout and fast retransmit

that aims to aid the performance of both Reno and New Reno when facing multiple packet losses.

In the rest of this thesis, unless something else is specified, when we say "TCP" we mean TCP New Reno and "RDB" means TCP New Reno with RDB enabled.

TCP Reno

When starting a transmission, TCP Reno uses a technique called slow start to avoid sending more data than the network can transfer. During the initial phases of a connection, TCP determines the MSS and initializes the `cwnd` to be less than or equal to `2*MSS` (depending on the congestion control). The size of the `cwnd` is increased by one MSS for every ACK that the sender receives, which means that the size of the congestion window doubles for every round-trip time (the time it takes for a packet to travel to and from a receiver, RTT [18]). Provided that there is enough data to transfer and no packets are lost, the connection will first be allowed to send one packet, then two, then four, and so on without receiving ACKs. Figure 2.2 shows how the `cwnd` grows, and also what happens when a packet is lost. The latter will be discussed later in this section.

This doubling of the `cwnd` continues until it reaches a pre determined threshold called the slow-start threshold (`ssthresh`). When this value is passed, the connection enters the congestion avoidance phase. The `cwnd` is increased by one MSS for each RTT, thus we have exponential growth before ssthresh is passed and linear growth after. To avoid relying on clocks (which are often to coarse), [13] recommends that the cwnd is updated for every received ACK using the following formula:

$$\texttt{cwnd} = \texttt{cwnd} + (MSS * MSS/\texttt{cwnd}). \quad (2.1)$$

Even though RFC2581 [13] specifies that the cwnd is calculated in bytes, several OS' (including Linux) use packets instead. In other words, provided that the sequence number of the packets does not exceed the limit enforced by the flow control [13], the connection is allowed to send one packet, then two, then four, and so on. When the connection passes ssthresh, the number of allowed, unacknowledged packets increases linearly (one for each RTT). The number of sent, but unacknowledged packets is known as the number of packets in flight.

TCP Reno uses two different techniques to discover and retransmit lost packets. If no acknowledgments are received before the retransmission timer expires, an RTO is triggered. Exactly how the timer is updated is OS-specific and discussed in more depth in section 3.3.1. When a timeout occurs, the ssthresh is set to cwnd/2 and the connection enters slow start again, i.e. it has to start with a cwnd of 2*MSS (or less). The reason for reducing ssthresh is that TCP Reno assumes that all loss is due to congestion. The estimated share of the bandwidth was apparently too high, and must therefore be reduced. Also, a lower ssthresh-value ensures a slower growth rate. This stops the connection from suddenly flooding the network with more data than it can handle.

Another important thing that happens when an RTO occurs, is that the retransmission timer is doubled. This is called the exponential backoff and will inflict severe delays if the same packet is lost several times. In Linux, the minimum RTO (minRTO) value is 200 ms, meaning that if the packet is lost three times, then the wait for the next retransmission is over a second long (200 * (2^0 + 2^1 + 2^2)). This is very bad for thin streams, as will be discussed in section 2.4.1.

As mentioned in the previous section, when a receiver receives a packet with a higher sequence number than the one expected (e.g. if the previous packet was lost), it sends a dupACK. This is done to let the sender know that it has not received all of the previous packets, meaning that it can not deliver any more data to the application. Because of the in-order requirement, data delivered to the application must form a continuous byte range.

Until the packet with the expected sequence number arrives, the receiver will continue to send dupACKs for every received packet. After

N dupACKs (three in the Reno/New Reno implementation we used) are received, the sender will retransmit the first lost packet (this is called a fast retransmit) and enter fast recovery. In this state, `ssthresh` is set to `cwnd`/2 (as when an RTO occurs), but the connection does not have to go through slow start again. DupACKs tells us that the packets are buffered at the receiver and no longer consume network resources.

Instead of `2*MSS` (or less), `cwnd` is set to `ssthresh + 3*MSS`. The latter part of the last equation is there to ensure that new data can be sent, as long as it is permitted by the `cwnd` and `rwnd`. The three packets that generated the dupACKs have not been "properly" acknowledged, and are therefore occupying space in the congestion window. Thus, the `cwnd` has to be artificially inflated to allow new data to be transferred. The `cwnd` is increased by one MSS for every received dupACK for the same reason. When the next ACK arrives, the `cwnd` is reduced to `ssthresh` and the connection leaves fast recovery.

Because an ACK indicates that the network is no longer congested, it is "safe" to start with a congestion window size of `ssthresh` (since this is the estimated share of bandwidth). If the sender continues to receive dupACKs, the fast retransmit/recovery will be repeated until the `cwnd` is so small that no new packets can be sent. If the retransmitted packet is then lost, a retransmission timeout will occur.

TCP New Reno

Unfortunately, the way TCP Reno deals with fast recovery is not ideal when you have multiple packet losses. If the ACK that makes the sender leave fast recovery is followed by N dupACKs, the sender will enter fast recovery again, halving the `cwnd` once more. New Reno [19] modifies the way fast recovery/retransmission works, it stays in fast recovery until all unacknowledged data has been confirmed received.

To be able to do this, New Reno uses a partial acknowledgment concept. When entering fast retransmit, the highest sequence number sent so far is stored. Every received ACK is then compared against this value, and if the acknowledged sequence number covers the stored sequence number, then it is safe to leave fast recovery. Otherwise, more packets have gone missing and the first unacknowledged packet is retransmitted (when an ACK arrives).

The partial acknowledgment concept also allows TCP New Reno to check for false dupACKs. The highest transmitted sequence number is also stored whenever an RTO occurs. When three dupACKs are received, the sender checks if they cover this sequence number. If they do, then the connection enters fast recovery. Otherwise, the dupACKs are for packets sent before the timeout, thus the lost packet is already retransmitted.

One scenario where false dupACKs may be a problem, is when you have long RTTs. If a packet is lost and the N consecutive packets arrive, the dupACKs might not get back before an RTO is triggered. When they are received they will acknowledge a packet with a lower sequence number than the highest transmitted. The sender will then detect these dupACKs as false and not enter fast recovery.

Unfortunately, TCP New Reno does not fix all the flaws in TCP Reno. It could still take many RTT's to recover from a loss (since you have to wait for the ACK/more dupACKs), and you will have to send enough data to get the receiver to respond with N dupACKs. Both of these are challenges for thin streams, especially the latter. As shown in chapter 2, the IAT is often high, and it could take several hundred milliseconds before three new packets are sent.

TCP SACK

Selective Acknowledgment (SACK) [20] [21] is a strategy that handles multiple packet losses better than plain TCP Reno and TCP New Reno. When a packet arrives out of order, TCP will send a dupACK acknowledging the last packet that arrived in order. Thus, the sender will be informed that one packet has arrived, but not told which one. This forces the sender to wait at least an entire RTT to discover further packet loss (since it must wait for a retransmission to be acknowledged), or it might retransmit packets that have already been received (if allowed to by the flow- and congestion control). In other words, multiple packet losses can cause a significant reduction in throughput.

With SACK, however, the dupACK will also contain the sequence number of those packets that have arrived out of order. SACK does this by using something called SACK Blocks (which are stored in the option part of the TCP header), and each block contains the start and end sequence number of each continuous byte range that has been received. This leads to an increased throughput, the sender no longer has to wait at least one RTT to discover further packet loss, and only the packets that have not arrived will be retransmitted.

2.2.2 TCP fairness

One of the most important aspects of TCP is the fairness principle. If N TCP streams share the same link, they will each get an equal share (1/N) of bandwidth. Congestion control, which was discussed in the previous section, is used to enforce fairness by limiting the packet send rate (the `cwnd`), and reducing it if needed. As previously mentioned, TCP assumes that all loss is due to congestion, i.e., the stream has exceeded its fair share of the bandwidth. For example, a TCP session sessions is never allowed to send more than the congestion window allows to avoid flooding the network, and exponential backoff tells the streams to hold back due to congestion.

Unfortunately, the fairness principle does not apply to all streams. If several streams with different RTT use the same link, then those with a short RTT will have an advantage. Their send rate will grow faster (e.g., in slow start the `cwnd` will double for each RTT), and they will spend shorter time recovering from a loss. Thus, they will consume a larger share of the bandwidth.

2.3 Thin stream experiments in a lab environment

As discussed in the previous section, interactive applications tend to generate traffic that TCP is not tuned for. In this section, we will look at what effects altering different parameters will have on a thin stream sent over TCP. We decided to perform three sets of tests and varied the loss rate, the RTT, and finally the IAT. The packet size was always 120 bytes (the average packet size generated by the interactive applications we had measured when these tests were performed), and the results are based on ten 30 minute long runs for each combination of the test parameters.

For every test, we measured the number of retransmissions, and our findings will later be compared to those with RDB (chapter 3) enabled. Retransmissions are one of the largest enemies of thin streams - due to TCP's reliability, the connection has to wait for a lost packet to be retransmitted before it can deliver any more data to the application. We disabled all our TCP modifications, so even though we have used a thin stream in the tests (because the performance will later be compared to RDB), our findings would apply to throughput-intensive streams as well.

In the tests, we used a constant IAT of 140 ms, RTT of 100 ms and 0.5 % loss, unless the parameter was the one we varied. 140 ms IAT was the average of the time-dependent thin streams we had measured when these tests were performed, while the RTT was the average of several measurements made between the University of Oslo (UiO) and different machines around the world. The loss rate was chosen because we wanted it to be as low as possible, while still forcing the connection to trigger retransmis-

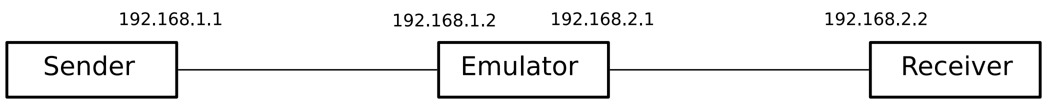


Figure 2.3: Our test-network.

sions. After experimenting with different rates, we found out that 0.5 % was ideal.

2.3.1 Test setup

To perform the tests presented in the previous section, we built a small network consisting of three machines (figure 2.3). We ran **streamzero** on the sender and receiver to create/receive traffic, this allowed us to have full control over the generated stream. To impose loss and delay on the links, we ran **tc** on the emulator. Unfortunately, **tc** uses a uniform loss pattern, so we rarely saw multiple packet loss. In other words, SACK was not able to improve the performance. All the software is described in appendix A. By sending data to IP-address 192.168.2.2, all data from the sender goes through the emulator (which also acts as a bridge).

All machines were Pentium 4 1.6Ghz with 512 MB RAM and Fast Ethernet network cards. This ensures a 100 Mbit/s link between the sender, emulator and receiver. The machines ran our modified 2.6.22.1 Linux-kernel (RDB and the modifications presented in section 2.4 were switched off).

2.3.2 Loss rate and TCP New Reno

In figure 2.4 we see that a higher loss rate resulted in a larger number of retransmissions. This is more or less given; when packets are lost and the connection uses TCP, they have to be retransmitted due to TCP's reliability. Hence, when we increase the chance of a packet being lost, the number of retransmissions will also rise.

A high loss rate will decrease the performance of any stream. For bulk data transfer and other "regular" TCP streams, it will lead to a lower transfer speed and probably an annoyed user. But unlike those generated by interactive applications, "regular" streams have no timeliness requirement and the extra wait is therefore not critical. The user experience in interactive applications generating time-dependent thin streams will, on the other hand, suffer as the loss rate increases.

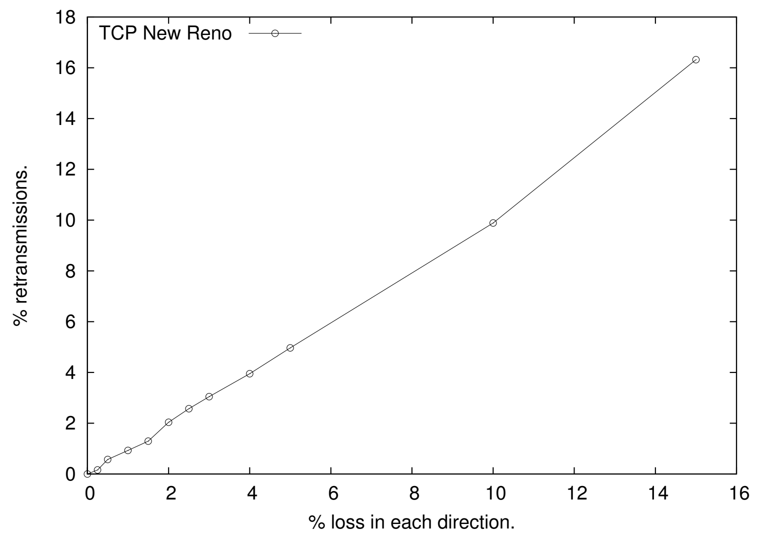


Figure 2.4: Retransmissions versus loss rate while using TCP. 100 ms RTT and 140 ms IAT.

Waiting for retransmissions will increase the latency. It will take longer before a lost packet arrives and the sender is able to deliver more data to the application. In addition, several thin streams are not able to trigger fast retransmits due to a high IAT, as mentioned in the thin stream definition in chapter 1. Instead, RTOs will be triggered and make the situation even worse.

2.3.3 RTT and TCP New Reno

RTT is a measurement of the delay in the network, and figure 2.5 shows that the number of retransmissions were independent of the RTT. This is because the TCP-calculation of the RTO timer (presented in detail in section 3.3.1) only cares about the RTT variance. As long as the RTT does not fluctuate, the RTO timer adapts to the increased RTT and does not trigger any more retransmissions than for lower RTTs.

The average transport layer (and thereby application layer) latency increases along with the RTT. This is also as expected, as a higher RTT forces the packets to spend more time traveling. Thus, it will take longer before a packet is received, and until it is acknowledged. Both the wait for an RTO (due to the way the RTO is calculated) and fast retransmit (due to the longer wait for acknowledgments) will be longer, increasing the wait for a retransmissions and potentially reducing the user experience.

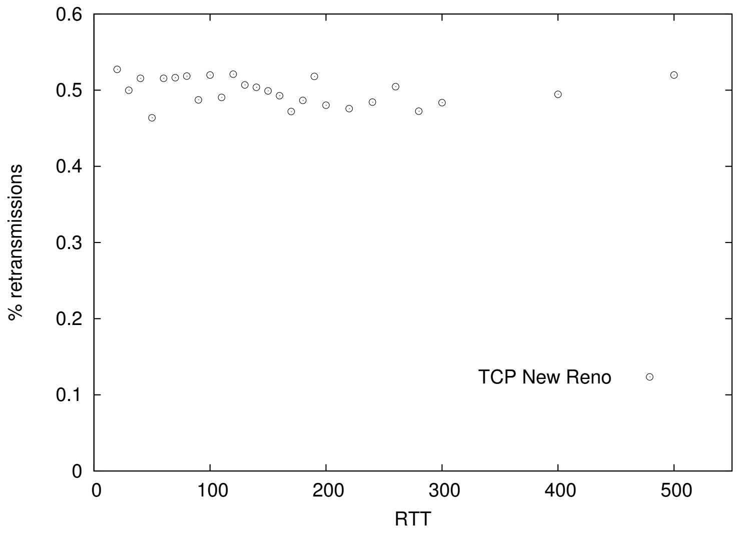


Figure 2.5: Retransmissions versus RTT while using TCP, loss = 1%, IAT = 140 ms.

2.3.4 IAT and TCP New Reno

The IAT is defined as the time interval between each time the application sends a packet. Compared to the TCP streams generated by e.g. bulk data transfer (where the IAT is as close to zero as possible), the IAT for thin streams are often high (as shown in table 2.1). In figure 2.6, we see how increasing the IAT affected the number of retransmissions.

As long as the IAT was less than 200 ms, the number of retransmissions remained more or less constant. However, when the IAT passed 200 ms, the share of retransmissions increased significantly (and then remained constant again). This was caused by the IAT (plus the RTT) crossing the RTO value - when we kept the RTT constant at 100 ms, the RTO was always close to 200 ms plus the RTT. Exactly how this value is calculated, and why it always was at least 200 ms, is presented in section 3.3.1.

Since the loss rate was constant, the same share of packets was lost for all IATs. However, when the IAT was less than 200 ms, two or more packets were sent between every RTO. If the acknowledgment for the first packet was lost, the second packet would implicitly acknowledge the previous one and avoid a retransmission by timeout. When the IAT grew past 200 ms, RTOs were the only option since less than one packet was sent between each timeout. This would increase the latency, the applications would have to wait longer for a retransmissions, and exponential backoff would increase the latency even more if the same packet was lost multiple

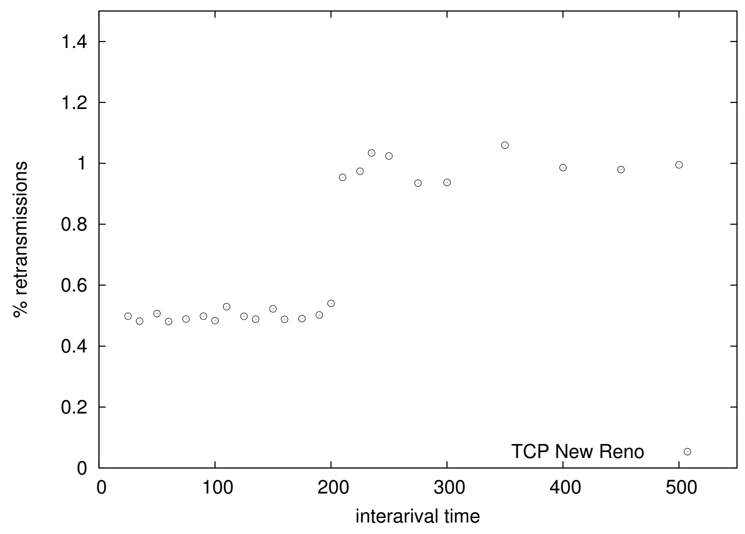


Figure 2.6: Retransmissions versus packet IAT while using TCP, loss = 0.5 % (in each direction), RTT = 100 ms.

times. In other words, a high IAT will severly reduce the performance of a stream with regard to latency.

2.4 Modifications

In order to improve TCP's performance for time dependant thin streams, we at IFI/Simula have developed two modifications to TCP New Reno, in addition to RDB. The latter will be introduced, discussed and evaluated in the next two chapters, in this section we will give a presentation of the two other modifications.

2.4.1 Removal of exponential backoff

As mentioned in section 2.2.1, exponential backoff will double the value of the RTO timer every time the same packet has to be retransmitted. In Linux, the `minRTO` is 200 ms, and it is at least as high as the RTT. The RTO value is calculated using the formula presented in section 3.3.1, and in our experiments it was always close to `minRTO` + RTT. Thus, if the RTT was 100 ms, the RTO would be around 300 ms. If the same packet was lost three times, the kernel would have to wait for more than one second before it attempted another retransmission.

This is very damaging for interactive applications, waiting over a second for an event will in many cases lead to unexpected behavior by the applications, and an irritated user. In [1], C. Griwodz and P. Halvorsen modified the behavior of the exponential backoff mechanism. Due to the low number of packets sent by the application that they based their experiments on (the MMORPG Anarchy Online), they saw that almost all the retransmissions were caused by RTOs. Thus, exponential backoff was the main cause of the occasional high latency, and they removed the effect of this mechanism for special cases.

Provided that three or fewer packets are in flight, the exponential backoff is disabled and the retransmission time is only increased by a smoothed RTT measurement (Smoothed Round-Trip Time, SRTT)for every RTO. However, if more than four packets are in flight, the connection is able to trigger a fast retransmit and is defined as "thick". The modification is then turned off, restoring regular TCP behavior.

Since the number of packets is so low (at least in their experiment), removal of the exponential backoff will not cause the network to be flooded with unnecessary retransmissions, at least not for the tested loss rates. However, the result would be different had the loss rate been extremely high. Several packets would be lost multiple times, and disabling the exponential backoff mechanism would ensure that they were retransmitted both faster and more often.

Figure 2.7 shows that the gain is significant when the same packet is lost multiple times. The first retransmission is triggered at the same time as without the modification, since exponential backoff has not yet updated the RTO. The second and third, on the other hand, are triggered much faster and ensure a speedier delivery of the data.

2.4.2 Reduce number of required dupAcks

As discussed in section 2.1, a lot of applications that generate thin streams send out packets quite rarely. The number of dupACKs is therefore limited, and the chance of triggering a fast retransmission small. Instead, an RTO will occur, which means that the exponential backoff will double the timer value and make the situation even worse (unless you use the modification in the previous section).

In [8], E. Paaby reduces the number of required dupACKs from three to one, provided that less than four packets are in flight. This will make fast retransmissions much more likely, and thus reduce the number of RTOs. As with the removal of exponential backoff, the limit of four packets in

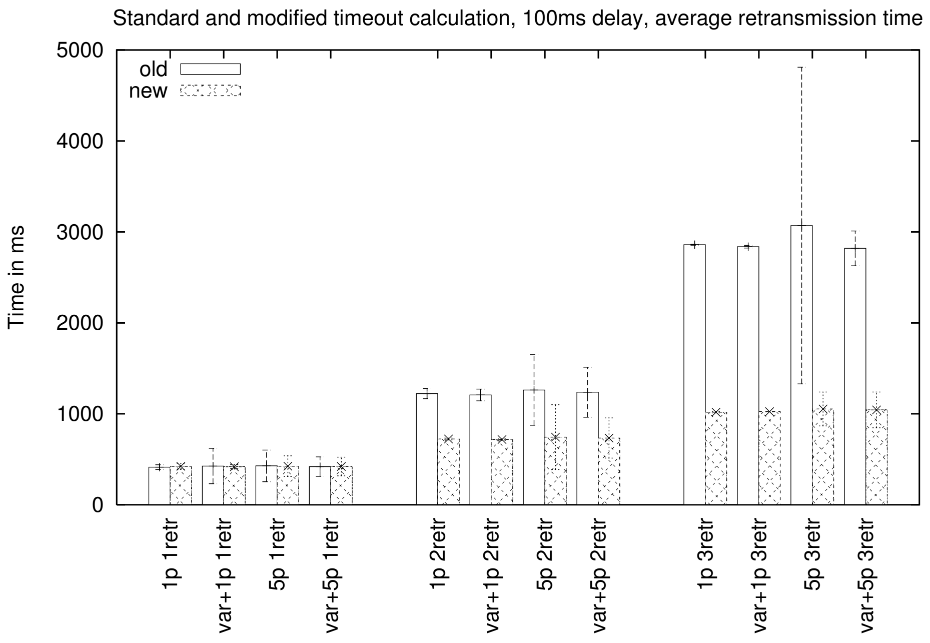


Figure 2.7: 100 ms path delay, successful 1., 2. and 3. retransmissions [1]

flight is chosen because this is the minimum number of packets needed to trigger a fast retransmission. A stream with four or more unacknowledged packets is most likely able to trigger these on it's own.

2.5 Summary

Interactive applications with network functionality generates time dependent traffic. UDP is, due to it's simplicity, often the best suited protocol for this kind of applications. However, UDP is not fair to other streams and is often blocked by firewalls. Thus, it is desirable to use TCP instead.

Most of the development of TCP, however, has been focused on improving the performance for throughput-intensive streams with no timeliness requirements, like the ones generated by applications doing bulk data transfer. These streams will try to send as large packets as possible, as fast as possible. Many interactive applications, however, have thin stream characteristics; the IAT is high or packet size small. In addition, they face strict latency requirements and will thus generate time-dependent, thin streams. If data is delayed, the application might not behave as intended and the user experience will suffer.

In other words, the network characteristics of many interactive applications are the complete opposite of high throughput streams (which TCP is tuned for). The high packet IAT is caused by the application's dependency on some event to happen before they have any data to transfer, while the small packet size is due to the exchange of short messages (like position updates in games). After measuring and looking at the network characteristics for many time-dependent thin stream applications, we saw that different types of applications generate different traffic. They all sent small packets, but fast paced games and the VoIP-application Skype had a much lower IAT than the other application types.

Unlike UDP, TCP provides the applications with a reliable, flow-controlled, stream- and connection-oriented way to send data over a computer network. There exist several different TCP variations that cater to different scenarios. We have chosen to focus on TCP New Reno because it has been shown in [1] that it is the best protocol variation for thin streams. We have presented how it deals with lost packets and retransmissions (congestion control), which constitute a problem for thin streams. The different mechanisms often increase latency, which is not good when you have interactivity. If the links were perfect, TCP and UDP would perform the same when faced with this kind of network traffic.

To find out how TCP performs when faced with a thin stream, we ran several tests inside our controlled network environment. We saw that both a higher loss rate, RTT and IAT increased the transport (and thereby the application) layer latency, something that might damage the user experience. In addition, many streams are so thin that they are only able to trigger retransmits by RTOs, which will make matters worse if the same packet is lost multiple times. Additionally, exponential backoff will double the RTO for every loss.

We have developed two modifications that aim at making the congestion control better suited to thin streams. Provided that the stream is not able to trigger a fast retransmit (the number of unacknowledged packets is less than four), we remove the exponential backoff and reduce the number of required dupACKs to trigger a fast retransmit to one. In the next chapter, we will present RDB, which aims at reducing both the transport- and application layer latency of streams with small packets and IATs less than RTO - RTT.

Chapter 3

Redundant data bundling

As presented in chapter 2, interactive applications have strict requirements when it comes to latency. Multiplayer games have a required latency between 100 ms and 1000 ms in order to be playable [3], while the (one-way) latency for audio conferences and IP-telephony cannot exceed 400 ms [4].

As such, having to wait several hundred milliseconds (or more) for a retransmission, which the different TCP mechanisms may do, will have a severe impact on the user experience. As these delays occur as a result of TCP's congestion control mechanics (see section 2.2.1), we have in this thesis look at a possibility for elevating this situation. With the intent of reducing the latency to the application when exhibiting thin stream characteristics using TCP.

In this chapter, we will go into details of our sender-side TCP modification that aims at improving the latency for thin streams. We have chosen to call it Redundant Data Bundling, as the main idea is to include old, which in this context means unacknowledged, data into new packets. By doing this, lost data might be delivered earlier than with an ordinary retransmission, because it is piggybacked in the subsequent packet. An advantage with RDB is that it is transparent to the sending application, it only requires a modification of the sender's TCP implementation. Additionally, the receiver needs to make no changes to reap the benefits.

3.1 Utilizing free space in a packet

One of the thin stream-characteristics, as we defined them in section 2.1, is that the packet sizes are small. When coupling this with the fact that network technologies like Ethernet often have a minimum frame size, e.g., Gigabit Ethernet (in half-duplex mode) sends a minimum of 512 bytes (4096

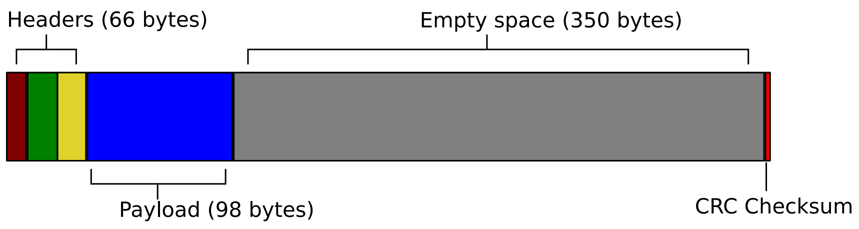


Figure 3.1: The minimum size of a gigabit Ethernet frame when transferring a 98 byte Anarchy Online-packet.

bit time slots), no matter the size of the payload [11]. Take, for example, an Anarchy Online packet with a 98 byte payload (table 2.1). The Ethernet frame carrying it will look like figure 3.1 and have a lot of unused space.

The TCP-implementation in the Linux kernel already does some limited bundling of data when sending and retransmitting. Nagle's algorithm [22] merges small user writes, and the stream control transmission protocol (SCTP) [23] can carry more than one user message in the same packet. In addition, SCTP and an extension to the real-time transport protocol (RTP) [24] sometimes carry unacknowledged (possibly redundant) data in new packets. Enet, a popular UDP-middleware, also does some limited bundling [7].

In [8], Espen Søgård Paaby suggests filling up the empty space in Ethernet frames (up to the current MSS) by merging packets upon retransmission, i.e., the kernel takes two or more "small" packets and merges them into a larger one. By doing this, he reduces the chance of additional retransmissions of the same packet(s), and suggests that somebody should try doing the same when sending packets. If you bundle upon send, you do not have to wait for a retransmission to merge packets and further increase the chance of the data arriving. We tried to implement merging on send, but experienced severe difficulties and decided to go for another approach.

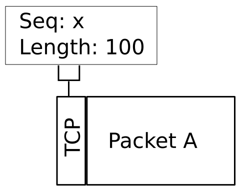


Figure 3.2: The 100 byte large packet A right after it is sent.

Inspired by the work of Paaby and the other techniques discussed here, we implemented Redundant Data Bundling, a technique which we have

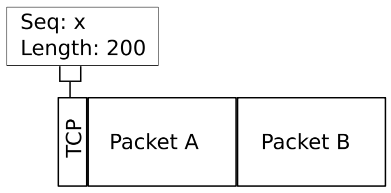


Figure 3.3: Packet B is sent before packet A is acknowledged, and a bundle is performed.

found no equivalent to. Instead of merging packets, we bundle (copy) potentially redundant data into unused space in the packets (up to the current MSS), which are either to be sent or retransmitted. Take the scenario where we have two packets of 100 bytes that are ready to be transmitted. The first one is sent using TCP and has the sequence number X and a payload length of 100 (figure 3.2). Then, when the second packet is processed, and if the first has not yet been acknowledged, the first packet is bundled with the second, which results in a packet that has the same sequence number X, but a payload length of 200 (figure 3.3). Thus, we consume more bandwidth than the piggybacking mechanism in [8], because in contrast to merging we potentially send every byte several times.

If packets are merged (as [8] suggested), you risk being left with a few large packets. Packets sent (to the kernel) before an ACK arrives will be merged, and if enough data for four packets is acknowledged, the kernel might only have one new packet to send. If RDB is used, the kernel will instead have packets of increasing size. This increases the chance of the data arriving faster at the receiver, as the unacknowledged data is continuous resent as long as there is sufficient space available.

3.2 Transparency

RDB is designed to be implemented in the OS kernel and thereby remain transparent to the application. Because we wanted it to be as easy to deploy as possible, we made RDB a sender side modification. TCP can receive and process bundled packets without modifications, thus the receivers do not have to patch their systems to acquire the benefits of RDB. However, all parties in nearly every time-dependent thin stream application will be a sender, so it will be beneficial to use RDB on all the machines. Consider a conversation held using IP telephony, in which both parties will act as a sender and receiver.

```
if (!after(TCP_SKB_CB(skb)->end_seq, tp->rcv_nxt)) {
  //If this test is true, the packet is a retransmission.
  ...
}

if (before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt)) {
  //If both this and the previous test is passed, the packet is
     what the kernel calls a partial packet (seq < rcv_next <
     end_seq), and it contains new data.
  ...
}
```

Figure 3.4: The two checks that determine if a packet contains new data (even though the sequence number does not equal the expected one). From the 2.6.23 Linux-kernel.

Requirement

The only requirement imposed by RDB is that the receiver OS checks both the packet sequence number and payload size. This is something most OSes do, including Linux. After having checked if the packet belongs inside its `rwnd`, the Linux-kernel performs the two tests presented in figure 3.4 to check if the packet contains any new data. `tp->rcv_nxt` contains the next expected sequence number, while `seq` and `end_seq` are respectively the first and last sequence number of this socket buffer (SKB). Each SKB represents one packet and contains meta-information like the sequence number, length, and checksum, as well as pointers to the payload.

However, if the OS does not check both the sequence number and payload size, the connection will experience a severely reduced send rate. If the IAT is lower than the RTT and the packets are small enough, bundles will be performed and data resent before acknowledgments are received. Bundled packets will have the same sequence number as a previously transmitted packet, but with different payload size because of the bundled data (as seen in figure 3.3). Thus, if a packet with the same sequence number has already been received and the payload size is ignored, the bundled packet will be discarded. The receiver will also stop delivering data to the application while waiting for a packet with the expected sequence number. When an ACK is received, the sender (when using RDB) removes the acknowledged data from all bundled packets containing it. However, this is too late to avoid a reduction of the send rate. Provided that a bundle is possible, the earliest packet that may have the expected sequence number,

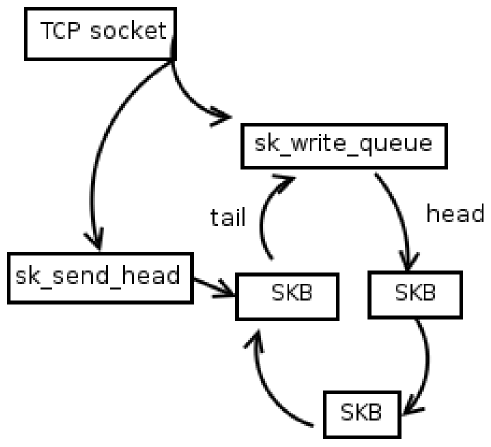


Figure 3.5: TCP output queue [2].

is the next one that is to be sent. If the IAT is low enough to fill up the `cwnd`, the send rate reduction will be even bigger. When the `cwnd` is full, no more packets can be sent and the connection must stop and wait for a retransmission before it can proceed. Also, because of the low IAT, this process will be repeated for every packet.

3.3 Implementation

In this section, we will examine in detail the implementation of RDB, which was done in the Linux 2.6.22.1-kernel (and later ported to 2.6.23.8). As we started out using the 2.6.22.1-kernel, this is the one we are most experienced with, and all code examples originate from here. There are, however, no differences between the two RDB-implementations, and patches for both kernels can be found on the included CD-ROM.

3.3.1 Linux networking internals

Output queue

One of the most important parts of the Linux TCP-implementation is the output queue, as shown in figure 3.5. Every socket has one queue, which is represented as a linked list consisting of SKBs. `sk_write_queue` points to the head of the output queue, while `sk_send_head` keeps track of which SKB is the next to be sent. If all SKBs have been sent, `sk_send_head` is NULL.

When an application sends data (e.g. with the `send()`-call), an SKB is created and placed at the tail of the queue. If the `cwnd` is open, or ACKs arrive and open it up, `sk_write_queue` is traversed from `sk_send_head` and sends as many SKBs as the flow control allows. Since TCP is reliable and has to hold onto data until the correct ACK arrives, the packets are kept in the queue after they are sent [5]. When an ACK arrives, the kernel starts at the head of the output queue and removes the SKBs that have been acknowledged.

Sending TCP-packets

We summarize this send-call sequence in figure 3.6 and describe the different functions in table 3.1.

When an application calls `send()` to transmit data through the network, `tcp_sendmsg()` is invoked, copies the data into an SKB and adds it to the end of the output queue. Depending on the number of packets that remain to be sent, `tcp_push_one()` or `tcp_push_pending_forward()` is called. The difference is that `tcp_push_one()` sends one packet, while `tcp_push_pending_frames()` keeps sending as long as `sk_send_head` is not NULL or the flow control or congestion control allows it. This is done by a call to `tcp_write_xmit()`, which loops through the output queue from the SKB provided as an argument.

`tcp_transmit_skb()` is called for every SKB that is going to be sent. It completes the TCP segment by building the header, checksums the packet, and then passes it along to `icsk->icsk_af_ops->queue_xmit()` (icsk is a struct containing information about an IP-socket), which is a pointer to the underlying protocol's service access point (SAP) (in this case IP).

If the `cwnd` is full, no more packets will be sent until it opens up. Packets are removed from the `cwnd` when ACKs arrive, these are first handled by `tcp_rcv_established()` (provided that the connection is established) and then sent to `tcp_ack()`, which calls `tcp_data_snd-_check()`. The latter uses `tcp_push_pending_forward()` to send data from the send queue.

Retransmissions also use the output code. Different functions are called depending on the type of retransmission, but they all end up in `tcp-_retransmit_skb()`. This function retransmits one SKB by calling `tcp-_transmit_skb()` with the SKB as a parameter.

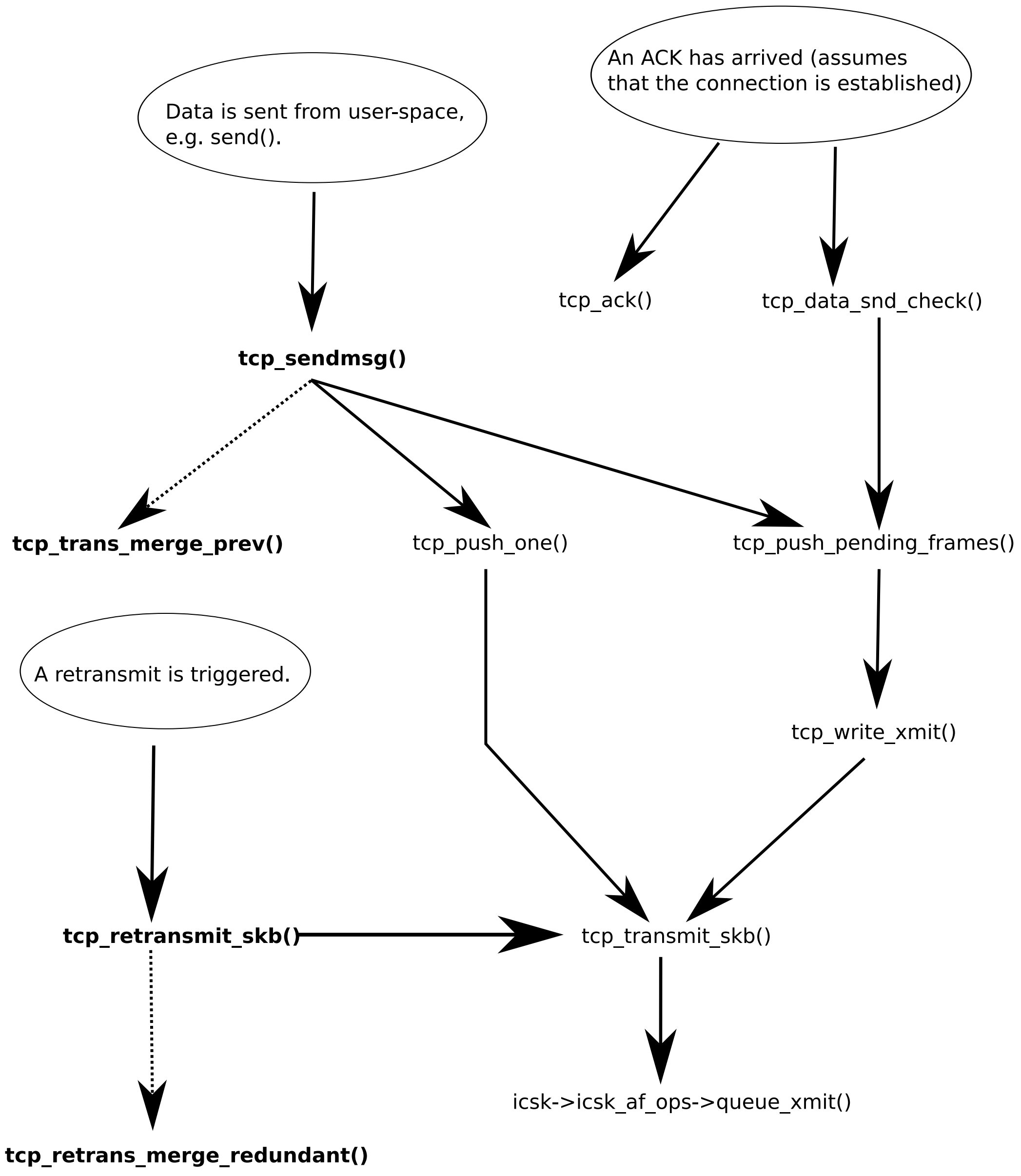

Data is sent from user-space, e.g. send().
An ACK has arrived (assumes that the connection is established)
tcp_ack()
tcp_data_snd_check()
tcp_sendmsg()
tcp_trans_merge_prev()
tcp_push_one()
tcp_push_pending_frames()
A retransmit is triggered.
tcp_write_xmit()
tcp_retransmit_skb()
tcp_transmit_skb()
icsk->icsk_af_ops->queue_xmit()
tcp_retrans_merge_redundant()

Figure 3.6: The call-sequence for outgoing TCP-packets, the functions we have implemented or modified are marked in bold.

Function name	Description
`tcp_ack()`	Deals with incoming ACKs. Updates values related to the send window.
`tcp_data_snd_check()`	Tries to send data from the send queue when the send window opens up. It is automaticually called when an ACK arrives.
tcp_sndmsg()	The entry point for data from user-space that is to be sent using TCP. Copies the data into an SKB.
tcp_trans_merge_prev()	Attempts to bundle one SKB with the previous one in the TCP output queue, implemented by us.
`tcp_push_one()`	Called if there is only one packet on the output queue that has yet to be sent. Checks if the send window has enough room for this SKB, and passes it along to `tcp_transmit_skb()` if that is the case.
`tcp_push_pending_frames()`	The same as `tcp_push_one()` when more than one SKB is yet to be sent. Calls `tcp_write_xmit()`, which traverses the output queue and send SKBs.
`tcp_write_xmit()`	Writes packets to the network and advances `sk_send_head` as long as the send window is open. Loops through the output queue from `sk_send_head` and till it reaches the end or the flow- or congestion control says stop.
tcp_retransmit_skb()	All retranssmisions end up in this function, which checks if the send window is open, transmits the SKB provided as an argument, and updates different counters related to the number of retransmittions.
tcp_retrans_merge_redundant()	Attempts to bundle the provided SKB with as many of the following SKBs on the output-queue as possible. Copies only the data that the SKB does not already contain, implemented by us.
`tcp_transmit_skb()`	Completes the TCP segment by building the header, checksums the packet and passes it to the IP-layer.
`icsk->icsk_af_ops->queue_xmit()`	The service access point in the IP-layer, i.e. this is were the SKBs enter the IP-layer.

Table 3.1: TCP output functions, the functions we have implemented or modified are marked in bold.

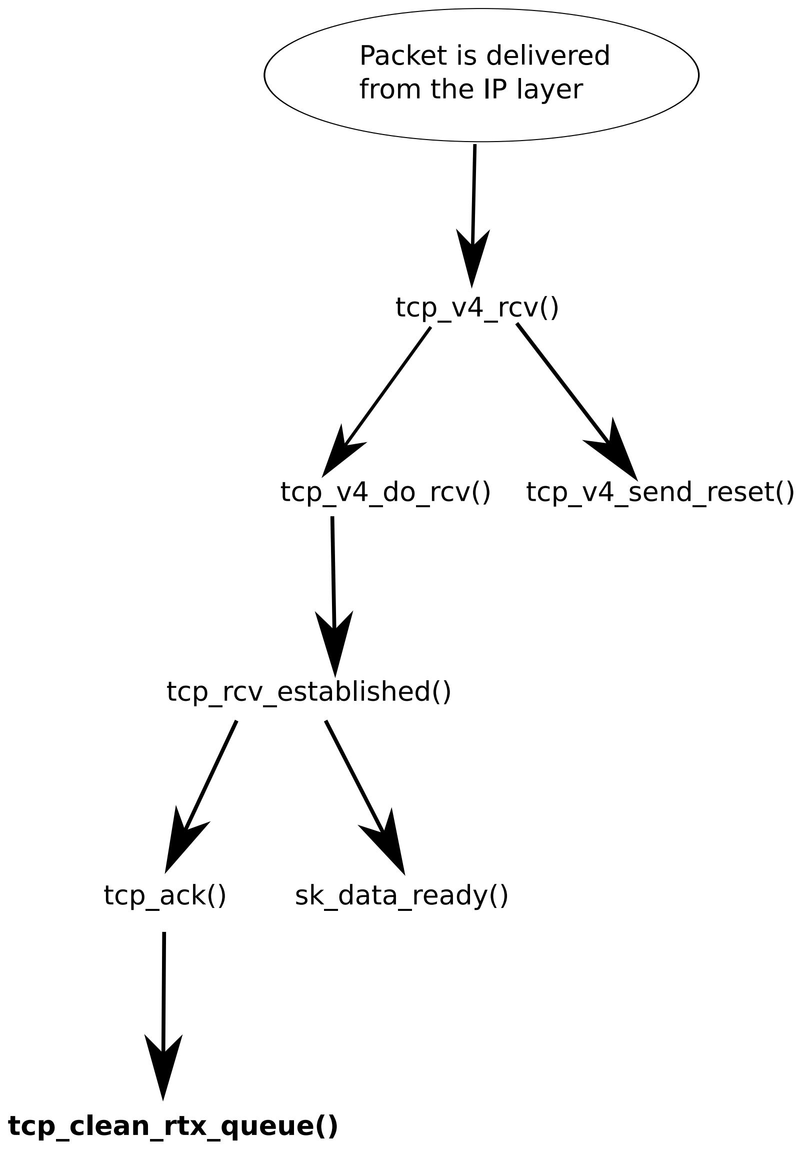


Figure 3.7: The call-sequence for incoming TCP-packets, the functions we have implemented or modified are marked in bold.

Function name	Description
`tcp_v4_rcv()`	The SAP at the TCP layer that deals with the packets IP passes upwards. Fills in the different values in the SKB and sends it to `tcp_v4_do_rcv()`, or `tcp_v4_send_reset()` if the packet doesn't belong to an active connection.
`tcp_v4_do_rcv()`	Checks if the packet has been received correctly and passes it on.
`tcp_v4_send_reset()`	Sends a RESET-packet to instruct the sender to close the connection.
`tcp_rcv_established()`	Copies the packets' payload to the receive memory of the process and notifies it if the packet has arrived in-order. If this is not the case, the packet is placed in the out of order-queue.
`tcp_ack()`	See table 3.1
tcp_clean_rtx_queue()	Removes acknowledged packets from the output queue.
`sk_data_ready()`	Notifies the application that it has received new data from the network.

Table 3.2: TCP input functions, the functions we have implemented or modified are marked in bold.

Receiving TCP-packets

We summarize the receive call sequence in figure 3.7 and describe the different functions in table 3.2.

`tcp_v4_rcv()` is the SAP at the TCP layer that deals with all packets that IP passes upwards. The function first determines if the packet belongs to a connection, and calls either `tcp_v4_do_rcv()` or `tcp_v4-_send_reset()`. The latter is called if the connection is not established and sends a RESET-packet, otherwise `tcp_v4_do_rcv()` is called.

We assume that the connection is established and `tcp_v4_do_rcv()` sends the packet to `tcp_rcv_established()`. Data packets are then copied to the receive memory of the process that owns the connection. If the packet has arrived in order, it is placed on the receive queue and the application is notified by a call to `sk_data_ready`. Otherwise, it has to wait in the out of order-queue for the packets that have gone missing or are delayed.

ACKs are treated by `tcp_ack()`, as briefly mentioned in the last section. `tcp_ack()` updates different variables related to the `cwnd`, and then calls `tcp_clean_rtx_queue()` to remove acknowledged SKBs from the output queue. When it is done, `tcp_data_snd_check()` is called. It checks if the output queue contains unsent packets, and sends as many as allowed to by the flow- and congestion control.

The RTO-timer

One of the most important parts of TCP when it comes to RDB, is the RTO timer value. This value decides how long the sender should wait for a reply before triggering an RTO, and for RDB to have an effect it must "beat" the RTO timer. The packet containing the bundled data must be sent and acknowledged before the timeout occurs. RFC2988 [25] specifies that the RTO must be reset whenever 1) a packet is sent and there are no other unacknowledged packets, 2) when an ACK/dupACK is received (and there are still unacknowledged packets), or 3) when an RTO is triggered. The value is updated for every ACK received.

The RTO-calculation is based on the SRTT and Round-Trip Time Variation (RTTVAR). [25] specifies that the default RTO value should be set to three seconds and updated using the following calculation (where K=4 and G is system granularity):

$$RTO = SRTT + max(G, K * RTTVAR) \tag{3.1}$$

When the first RTT measurement R is made (done after the first ACK is received), SRTT is set to R while RTTVAR is R/2. For each subsequent RTT measurement R′, [25] specifies that the SRTT and RTTVAR should be updated according to the following formulas:

$$SRTT = 7/8 * SRTT + 1/8 * R' \tag{3.2}$$

$$RTTVAR = 3/4 * RTTVAR + 1/4 * |SRTT - R'| \tag{3.3}$$

After these calculations, the RTO-value is recalculated using formula 3.1. If a retransmission occurs, RTO is set to RTO * 2 (the exponential backoff, presented in section 2.2.1).

Unfortunately, formula 3.1 is not perfect. After a long period of a constant RTT, a small additional delay can trigger an RTO. In addition, the calculation misbehaves with variable round-trip times. If the RTT suddenly drops, the last part of the RTTVAR-calculation will give an increased RTO-value. This is, however, intended by the developers of the formula. The idea is that if we have lots of jitter (the RTT is fluctuating rapidly), we should trust the RTO less and rely more on fast retransmissions. This will protect us against spurious retransmissions and having to enter slow start again. However, in most cases, this has turned out to be a bad idea and the increased RTO decreases performance.

Because of these two issues, Linux uses a modified version of formula 3.1. To avoid the first problem, the minimum RTO is 200 ms, ensuring that the timer will never get too close to the RTT. A large increase in the delay is needed to trigger a retransmission. The issue of the RTO-calculation misbehaving is resolved by changing the RTO-calculation (*icsk_rto*) to the following formula:

$$icsk\_rto = min((srtt >> 3) + rttvar, TCP\_RTO\_MAX) \qquad (3.4)$$

The weight of the RTTVAR is reduced, and the effect of a large fluctuation in the RTT is reduced. TCP_RTO_MAX is set to two minutes, and SRTT and RTTVAR are calculated the same way as in [25]. Both the initial RTO and SRTT start out as the values specified in [25], while RTTVAR is initially 200 ms. This is also the RTTVAR's lower bound, thus the `minRTO` is 200 ms. A retransmission will, as long as RTTVAR does not exceed TCP_RTO_MAX, cause the RTO to double (the exponential backoff).

The calculation is still not perfect though. Large amounts of jitter increase RTTVAR and in turn `icsk_rto`, leading to unnecessary long waits for retransmissions. Also, `icsk_rto` is initialized as three seconds (as specified in [25]. In other words, losing a packet at the start of a connection leads to a large retransmission delay.

3.3.2 Implementation details

The Linux TCP-implementation is spread out over several large files, but we are only interested in the parts that deal with the sending and receiving of packets. After studying the code, we found out that we only had to modify three functions. None of these affect the congestion control, so RDB should work with all TCP variations. We edited the following files:

- **tcp.c** is where packets start or end their journey through the kernel, and contains `tcp_sendmsg()`. We have modified this function by adding a call to our own `tcp_trans_merge_prev()`, which performs the (potential) bundle.
- **tcp_output.c** is involved in the actual sending of packets, as well as retransmissions. We have modified `tcp_retrans_skb()` by adding a call to our own `tcp_retrans_merge_redundant()`, which performs the (potential) bundle.
- **tcp_input.c** processes received TCP packets. All ACK packets are treated here, and to remove data from packets containing both acknowledged and unacknowledged data (i.e. a bundled packet), we

```
if(skb_headlen(skb) > 0){
  memmove(skb->data + uad_head, skb->data, old_headlen);
}

skb_copy_to_linear_data(skb, prev_skb->data +
    (skb_headlen(prev_skb) - uad_head), uad_head);
```

Figure 3.8: RDB - copying of linear data. `uad_head` contains the number of unacked bytes in the linear memory area of the previous skb payload.

had to modify `tcp_clean_rtx_queue()`. We want to avoid waisting bandwidth by transmitting already received data.

Bundling on send

When data is sent from user-space and RDB is used, `tcp_sendmsg()` first checks if it is possible to attempt a bundle. The criteria are that the SKB cannot be (or contain) a SYN or FIN, it has to be smaller than the current MSS, and it cannot be alone in the output-queue. If every test is passed, `tcp_trans_merge_prev()` is called.

Since the current SKB is the most recent, the kernel will attempt to copy the payload from the previous SKB in the output-queue. After calculating the amount of unacknowledged data, checks are performed to see if the size of the bundled packet will exceed the MSS, that the packet has enough room to store the potentially redundant data, and so on. The current SKB's payload is moved backwards in memory to make room for the "old" data at the front of the packet (as shown in figure 3.3), and the bundle is performed. `tcp_trans_merge_prev()` also calculates a new checksum for the SKB if the network card does not support checksum offloading (the card will do the calculation itself). In addition, the sequence number and payload-length are updated.

When both the current and the previous SKB contains only linear data, meaning that the payload is stored in one continuous memory area, the memory-operations are trivial (as shown in figure 3.8). The kernel has to perform one `memmove()`-call to move the current SKB's payload backward, and one `memcpy()`-call to copy the unacknowledged data.

If the SKBs are non-linear, the situation becomes more complicated. Non-linear means that the data is spread out over several pages in memory and is used in conjunction with zero-copy. Instead of copying the entire payload when sending SKBs to the IP layer, the kernel sends page references. The most important reason for doing this is to lighten the load

```
if(skb_is_nonlinear(skb)){
  memmove(skb_shinfo(skb)->frags + ua_nr_frags,
      skb_shinfo(skb)->frags,
      skb_shinfo(skb)->nr_frags*sizeof(skb_frag_t));
}

/*Copy info and update pages*/
memcpy(skb_shinfo(skb)->frags, skb_shinfo(prev_skb)->frags +
    (skb_shinfo(prev_skb)->nr_frags - ua_nr_frags),
    ua_nr_frags*sizeof(skb_frag_t));

for(i=0; i<ua_nr_frags;i++){
  get_page(skb_shinfo(skb)->frags[i].page);
}
```

Figure 3.9: RDB - copying of non-linear data. `ua_nr_frags` is the number of unacknowledged frags, and the frags has a pointer to the page that contains the data (amongst others).

on the kernel, memory-operations take time and occupy resources [26].

A non-linear SKB is designed to start with `skb_headlen(skb)` bytes in the linear area, and then continues into the non-linear area for skb -> data _ len bytes. This puts one additional constraint to when a bundle can be performed - all non-linear data must be stored after the linear data. Thus, you cannot bundle if the current SKB has linear data while the previous has non-linear data.

Information about the pages (called fragments) is stored in an array called `frags` in each SKB, and RDB moves the existing fragments backwards in the array first. Then the fragments containing unacknowledged data are copied, and the number of references to these particular pages are increased by one, as shown in figure 3.9. The latter is done to make sure that the Virtual Memory Manager (VMM) does not remove a page too early. If we did not update the number of references, the pages (and thereby the data) would be removed together with the original SKBs containing them. This removal would occur when the data contained in the pages is acknowledged, and cause the kernel to crash when it tries to send a bundled packet.

Bundling on retransmission

When a retransmission is triggered, the kernel will always end up in `tcp-_retransmit_skb()`. If the connection uses RDB, this function checks

```
if(skb_is_nonlinear(skb) && remove_frags > 0){
  no_frags = 0;
  data_frags = 0;

  /*Remove unnecessary pages*/
  for(i=0; i<skb_shinfo(skb)->nr_frags; i++){
    if(data_frags + skb_shinfo(skb)->frags[i].size ==
        remove_frags){
      put_page(skb_shinfo(skb)->frags[i].page);
      no_frags += 1;
      break;
    }
    put_page(skb_shinfo(skb)->frags[i].page);
    no_frags += 1;
    data_frags += skb_shinfo(skb)->frags[i].size;
  }
}
```

Figure 3.10: The process of removing pages from a partly acknowledged packet. `put_page()` decreases the number of users on a page by one, the VMM takes care of the actual removal.

if the SKB contains a SYN or FIN, and that it is not the only SKB that has been sent (but not acknowledged). If the tests are passed, the kernel calls `tcp_retrans_merge_redundant()`, which is very similar to `tcp_trans_merge_prev()`.

RDB wants to transmit as much unacknowledged data as possible, so the kernel attempts to bundle with the following packets on the output queue. Because the data is added after and before the SKBs original payload, the actual bundling and the non-linear constraint are the inverse of what was presented in the previous section. The kernel does not have to do any `memmove()` calls to move data around, and will simply copy the data or fragments into this SKB (provided that the size of the bundled packet will not exceed the MSS, that there is room in this SKB, and so on). However, if this SKB contains non-linear data and the following linear data, a bundle is not possible due to the way SKBs are designed.

ACK with RDB

The `tcp_ack()` function deals with incoming ACKs, and calls `tcp_clean_rtx_queue()` to remove acknowledged data from the output queue. As long as the entire payload is acknowledged, this function behaves exactly like with TCP and removes the entire SKB.

However, to avoid wasting bandwidth, we have modified `tcp_clean_rtx_queue()` so that it removes data from partially acknowledged SKBs (a packet containing both acknowledged and unacknowledged data). For example, if packet A is acknowledged (figure 3.2) after packet B is sent (figure 3.3), the acknowledged data should be removed from B as well. If a partially acknowledged packet is detected, our version of `tcp_clean_rtx_queue()` calculates the number of acknowledged bytes, and how many of them are in the linear and non-linear areas. It removes these bytes and updates the SKB. If a non-linear packet is acknowledged, we reduce the number of references to each of the acknowledged pages (figure 3.10), and move the unacknowledged pages forward in the array. If the packet is linear, we move the unacknowledged data to the front of the packet. In both cases, a new checksum is calculated, and the sequence and payload length are updated.

I/O Control and Proc-variables

To make RDB as dynamic as possible, we have implemented two different ways to turn it on and off. If somebody wants to use RDB in conjunction with a proprietary application or do not want to change the source code, it can be enabled by setting a proc-variable.

Proc-variables are also referred to as system controllers, and are kept in the `/proc`-folder at the root of a Linux file system. To set a variable, you have to pipe a value into it, e.g. `echo 1 > /proc/sys/net/ipv4/tcp_force_thin_rdb`. The kernel treats a proc-variable like any other variable, e.g. the check to see if `tcp_force_thin_rdb` is true looks like this:

`if(sysctl_tcp_force_thin_rdb){...}`

The proc-variables are stored in the `sysctl.h`-file of the kernel, and we have added the constant `NET_IPV4_TCP_FORCE_THIN_RDB`. To give the TCP code access to the variable, we had to declare an external integer in `tcp.h` (which we chose to call `sysctl_tcp_force_thin_rdb`), and create a mapping between the two variables. This was done by adding an entry the `ctl_table ipv4_table[]` in `sysctl_net_ipv4.c`, which maps proc-variables to external integers from `tcp.h`.

The problem with proc-variables is that they are global and will affect the entire OS. If no applications generate thin streams, this will not be an issue, because RDB will not be able to bundle because the packets are to large. However, if the machine is connected to a network with very limited and/or expensive bandwidth, this might be an issue. Provided that an application generating a thin stream is running, the increased bandwidth usage (due to the bundling) might cause the performance to drop or lead to increased costs.

We have solved this by implementing an I/O-control that allows people to use RDB on a per-socket basis. By setting a predefined constant when a socket is created, RDB is enabled only for this socket. For somebody to use the I/O-control, they must have access to the source code. End-users will rarely have this, so the I/O-control is mostly for developers.

The implementation of the I/O-control is simpler than of the proc-variable. First, we added a constant `TCP_THIN_RDB` to the file `tcp.h`, and the variable `thin_rdb` to the `tcp_sock`-structure (which contains various TCP-related information about a socket). `thin_rdb` is set by a user-space call to `setsockopt()`, which calls upon `do_tcp_setsockopt()` to enable RDB. Just like the proc-variable, the kernel treats an I/O-control like an ordinary variable. The check to see if the RDB-I/O-control is enabled looks like this (where `tp` is the `tcp_sock`-structure):

```
if(tp->thin_rdb){...}
```

3.4 Discussion

In this section, we will discuss two important aspects related to RDB. The increase in packet size, which might be bad for some users or network types. In addition, in [26] it is shown that memory operations are expensive. RDB performs several such operations for each bundle, and we will discuss if this will effect the performance of the machine and the network-part of the application.

3.4.1 Increased bandwidth usage

RDB bundles data if possible, which means that every packet might be expanded by a certain number of bytes if there is unacknowledged data. However, because of the increased packet size, RDB will use more bandwidth than regular TCP if bundles are performed.

As table 2.1 shows, several interactive applications generating thin streams have an average packet size of less than 100 bytes. Even if a packet is bundled with a number of other packets, the size will still be small. In addition, the payload cannot exceed the current MSS (e.g 1448 bytes in Ethernet, excluding the IP and TCP header), thus a packet will never be larger than what the network is able to handle.

We have figured out two situations where the increased bandwidth usage is a problem. Cellphone users that connect to the Internet (or another network) with their phones, usually pay for the amount of bandwidth they have consumed. If the current link either has a huge RTT or the IAT is lower than the RTT, bundles will be performed because the application sends data faster than ACKs arrive. If the link is reliable, most of the bundles will be unnecessary. Thus, cellphone users will (depending on the subscription type) pay more and get nothing in return.

One possible solution to this scenario is to check if the stream is thin and apply RDB more dynamically. One could e.g. implement something similar to how TCP's exponential backoff works - every retransmission allows the kernel to perform one more bundle (up to a pre-determined limit), and each subsequent ACK reduces the number of allowed bundles by one (until it reaches zero).

In another situation, the increased packet size is a problem. This is when the physical transport medium is of poor quality. Larger packets are more likely to be corrupted during transmission. We have yet to come up with a solution, but have decided that this is not a critical issue. Since the bundled packets are never larger than what the link is able to transfer, every network application (independent of RDB) will be affected by this. In addition, the MAC-layer sorts out any retransmissions due to corrupted data. However, this will not always solve the problem. If there is congestion at the MAC-layer, the delay before a packet is retransmitted might be so large that TCP will trigger a retransmission. This causes a reduction of the send rate, and the performance of the stream will be decreased even further.

3.4.2 Copying data

When bundling packets, RDB performs several memory-operations. It has to copy the payload from another SKB, and often has to move data inside an SKB as well. As shown in [26], memory-operations are expensive and a lot of research has been done to optimize them. In addition, the kernel-versions of `memcpy()` and `memmove()` do not do any optimizations and copy or move data byte by byte.

To reduce the number of the memory operations, RDB always works with the largest possible chunk of data. This way we do not occupy the different resources (CPU, memory bus, and so on) more than needed. Also, memory operations are so fast that they will not affect the network-performance of an application. Memory access is (at least) two orders of magnitude faster than both the IAT and RTT (micro- vs. milliseconds). For example, DDR2-RAM spend between 10ns and 3.75ns on one memory operation (and can transfer between 3200 MB/s and 8533 MB/s) [27], the IAT for a thin stream application is rarely below 100ms (table 2.1), and the RTT is most of the time larger than 100 ms (RTT is independent of the type of stream).

3.5 Summary

RDB is a TCP-modification that aims at reducing the number of retransmissions by bundling potentially redundant data. It is designed to be a sender-side modification and transparent to both the sender and receiver application. It also requires small (if we want to enable RDB on a per socket basis) or no (if we want to enable RDB on a system-wide basis) changes to work with existing applications.

In this chapter, we have presented the concept behind RDB and how we decided to implemented it in the Linux-kernel. In the next chapter, we will experimentally evaluate the performance and usability of RDB.

Chapter 4

Evaluation of RDB

In an ideal situation, when faced with an application generating a thin stream, TCP and UDP will perform similarly (except for TCP overhead). No packets would be lost due to poor link quality, and congestion would not be an issue since the network equipment would be able to deal with the traffic. Unfortunately, this is rarely the case.

Most networks have some loss and might struggle with congestion. If an application uses UDP, it can do various tricks to compensate for lost or late packets. For example, depending on the application type and application-layer protocol, it may drop old (i.e. delayed) packets. By doing this, a multimedia application will ensure a smooth playback with some image or audio corruption, or do recovery to compensate for lost data, e.g., predicting for the movement of characters in a game.

TCP, however, does not allow the applications to behave in this manner. For data to be delivered immediately to the receiving application, it must arrive in the correct order. If a packet arrives out of order (e.g. if the previous one was lost), it will be put in a queue and has to wait until the expected packet is retransmitted and received. Should more packets go missing, only those that follow directly after one another will be passed on to the application.

Waiting for lost packets will increase the application layer latency, i.e., the time it takes before an application can use the data. RDB is a TCP-modification that aims at lowering this latency when used with applications that generate thin streams. These applications have often strict latency requirements, thus having to wait several milliseconds (or seconds) for a retransmission could be disastrous. In other words, our main goal is to reduce the number of retransmissions.

In this chapter, we are going to present the findings from our tests of RDB's performance. First, we will show how the loss rate, RTT and IAT affect the number of retransmissions when using RDB. Then, we will investigate how different applications behave with and without RDB. In addition, for each application, we will also present the result of a survey conducted to determine if RDB and the mechanisms presented in section 2.4 have an effect on the user experience. Since there was a limit to how long we could occupy the test subjects, we did not have time to perform the user tests with only RDB. For each test, we discuss instead how much effect RDB had (compared to the other modifications). Finally, we discuss and summarize the results.

4.1 Thin stream experiments in a lab environment

In section 2.3, we looked at how TCP behaved under variable network conditions when faced with a thin stream. We measured the number of retransmissions, as these are one of time-dependent thin streams largest enemies. Having to wait for a retransmitted packet will hurt the user experience, especially if the connection experiences repeated loss of the same packet. If this occurs, exponential backoff will double the current RTO for every loss. Thus, if TCP or RDB experience repeated loss of the same packet (or data), the delivery latency can potentially increase from milliseconds to seconds.

In this section, we look at how RDB performs under the same conditions and with the same streams as in section 2.3. We used the same setup and a fixed payload of 120 bytes. And, unless the parameter was the one we varied, the IAT was 140 ms, RTT 100 ms, and loss rate 0.5 % in each direction. The reason for choosing these values is explained in chapter 2. The values in both the graphs and tables (in this section) are the average of ten tests run under the same settings for thirty minutes.

We started off by observing how RDB performed when the loss rate was increased. After wards, we investigated how RDB affected the performance of the thin stream when we varied the RTT and IAT. Our findings from all three experiments are presented in their separate subsections, along with tables that contain the ten worst-case average packet size differences in the given test.

One of the trade-offs discussed in section 3.4 was that RDB generates larger packets than TCP, because of the bundling, and we wanted to find out how much larger they were compared to TCP. Because we used a fixed payload size of 120 bytes, this was the lowest possible packet size. Thus, the desired average packet size was as close to this size as possible.

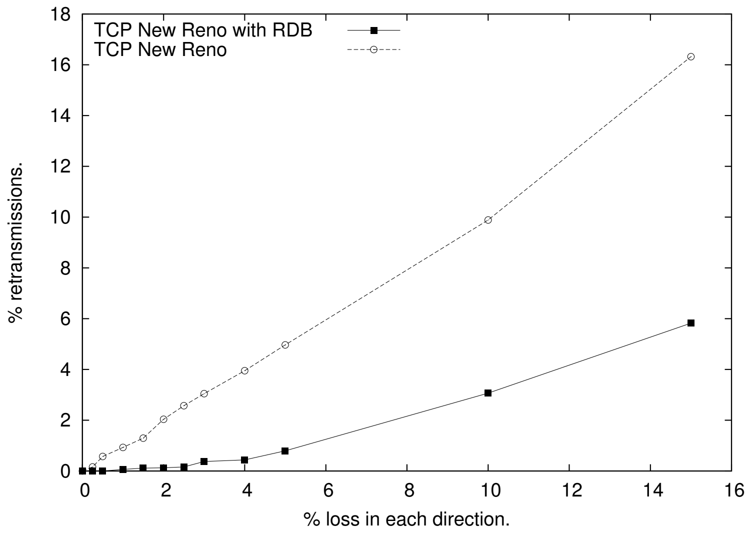


Figure 4.1: Retransmissions versus loss rate while using RDB. 100 ms RTT and 140 ms IAT.

4.1.1 Loss rate and RDB

In section 2.3.2, we showed that the number of retransmissions grew when we increased the loss rate. We expected RDB to behave similarly, but with a slower growth due to the bundling. With a constant RTT of 100 ms, the RTO was always close to 300 ms, allowing at least two packets to be sent and acknowledged before a retransmission timeout occurred. In other words, the effect of RDB should at least have been noticeable for the low loss rates.

Figure 4.1 shows that the results were as expected. The growth rate was much slower and we hardly experienced any retransmissions when using RDB, except for loss rates above 10 %. Unfortunately, when the loss rate is very high, RDB has some drawbacks.

The number of packets increases because the combined size of the packets bundled into one TCP segment reaches the MSS eventually, and there is no room for more bundling. Each retransmission would then be sent by the standard TCP retransmission scheme as a separate packet. When a retransmission occurs, standard TCP merges the unacknowledged data in one packet and thereby reduces the number of packets sent. Nevertheless, in our bundling scheme, the latency is less than in TCP New Reno even for the high loss rates.

As shown in table 4.1, the packet size increased along with the loss rate. Until a sent packet is acknowledged, it will occupy space in the `cwnd`.

Variable loss rate				
Loss rate	TCP (bytes)	RDB (bytes)	Difference, packet size (RDB - TCP)	Difference, packets sent (RDB - TCP)
10%	132.0	153.0	21.0 (15.91%)	283.0 (2.26%)
5%	121.0	134.0	13.0 (10.74%)	-450.0 (-3.45%)
4%	121.0	130.0	9.0 (7.44%)	-388.0 (-3.00%)
3%	120.0	128.0	8.0 (6.67%)	-320.0 (-2.49%)
2.5%	120.0	126.0	6.0 (5.00%)	-285.0 (-2.23%)
2%	120.0	125.0	5.0 (4.17%)	-236.0 (-1.85%)
1.5%	120.0	123.0	3.0 (2.50%)	-145.0 (-1.15%)
1%	120.0	122.0	2.0 (1.67%)	-105.0 (-0.83%)
0.5%	120.0	121.0	1.0 (0.83%)	-70.0 (-0.56%)
0%	120.0	120.0	0.0 (0.00%)	-2.0 (-0.02%)

Table 4.1: The worst-case packet size difference loss rate.

When the loss rate increases, this window is filled up more quickly. In addition, more and more packets are stored on the output queue, waiting for the `cwnd` to open up so they can be sent. Thus, when a new packet is sent from the application, there is a large probability that there is unacknowledged data. If RDB is enabled, bundles can be made (provided that the combined packet size is less than the MSS).

The reason for TCP's increased packet size when the loss rate exceeded 4 %, was that the existing bundling mechanisms in the Linux-kernel kicked in. As long as there is room in the packet, Linux will try to merge packets on retransmission, and copy data into the last packet that has yet to be sent (if any) upon the first send of the data. When the loss rate is high, the `cwnd` fills up quickly, and it takes a while before packets are removed and new ones let in. Thus packets have to wait outside the `cwnd`, and bundles can be performed.

We were a bit surprised by the packet size when we used RDB, we expected it to generate much larger packets than it did. When for example 10% of the packets were lost, we expected the send queue to be so full that more or less every packet would be bundled. Thus, the average packet size would be much larger than it actually was and it would get close to the MSS. This would eventually happen though, but not with the parameters used in this test. You can fit 12 120 byte packets into one MSS, which requires 12 bundles of the same data. Even with 15 % loss we did not reach this number, and if the loss rate reaches 15 %, it can be argued that the link is unsuitable for any application.

We expected RDB to generate even larger packets than TCP when the loss further, but this was not the case. As seen in table 4.1, the average packet size for 15 % loss is not there. This was most likely caused by the high loss rate, which caused unacknowledged data to always be present, as well as data waiting to be let into the `cwnd`. Thus, the built-in mechanisms were able to bundle on both retransmission and send, leading to larger packets. RDB bundled more data at 15 % than for the lower loss rates. The average packet size was larger than for 10 % loss, but the redundancy increased the chance of data arriving and being acknowledged. On average, it took more than 700 ms before a packet was acknowledged without RDB, compared to 169 ms with RDB. The amount of unacknowledged data and thereby potential bundles was reduced.

Even though the packet size was often larger when RDB was enabled, the number of packets was smaller for most loss rates (even for 0 %, which is most likely completely random). This was caused by the increased number of retransmissions - TCP had to retransmit more packets, thereby increasing the number of sent packets. RDB, on the other hand, experienced fewer retransmissions and did not generate as many additional packets as TCP. However, with 10 % packet loss, RDB sent more packets than TCP. Unlike the lower loss rates, most retransmissions experienced by TCP were RTOs because of the increased number of lost ACKs. After an RTO, the connection must enter slow start, which will reduce the send rate significantly. In addition, the connection have to stop and wait for the RTO timer to expire, reducing the send rate even more. RDB, again, experienced fewer retransmissions and was able to maintain a higher send rate. Hence, it sent more packets than TCP.

4.1.2 RTT and RDB

The RTT is the time it takes for a packet to travel from the sender to the receiver and back, and different TCP-variations use the RTT-value for different purposes and in different ways. However, it is always used in the calculation of the RTO-value. In the Linux-versions we used, the RTO was always approximately 200 ms plus the RTT.

With regular TCP, the RTT did not affect the number of retransmissions (section 2.3.3). This is because the RTO-calculation (described in section 3.3.1) is independent of the actual RTT, as it only cares about the variance. Consequently, it adjusts the RTO-value to the increased RTT, and waits longer before triggering a retransmission.

As shown in figure 4.2, this also applied to when we used RDB; the RTT did not influence the number of retransmissions. The lower number of retransmissions was caused by the IAT always being less than the RTO minus the RTT, which enabled RDB to deliver potentially lost data and get it acknowledged before a timeout. In our tests, the RTO was (as already

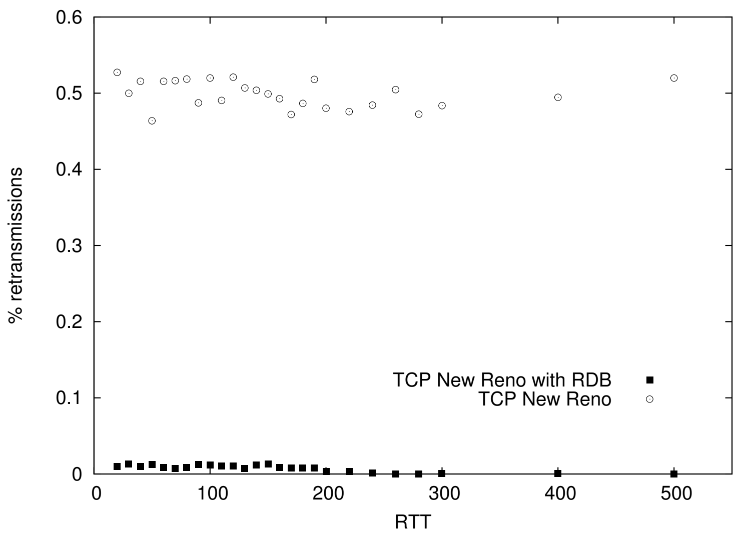


Figure 4.2: Retransmissions versus RTT when using RDB, loss = 1%, IAT = (150 +- 10) ms.

mentioned) always equal to `minRTO` (200 ms) plus the RTT. For example, when the RTT was 10 ms, the RTO was 210 ms. For RDB to be able to bundle and deliver the data, and get the ACK before a timeout is triggered, the packets had to be sent with less than 200 ms IAT (RTO minus the RTT). This means that IATs of 140-160 ms, which were used in this experiment, were small enough for the stream to benefit greatly from RDB.

RDB will also have an effect if the IAT passes the RTO, but it is much smaller. As long as the packets are small, bundles are made and lost data delivered in an earlier packet than it would with TCP. This is discussed in more detail in the next section.

Still, a very small number of retransmissions occurred when the RTT was low. This was caused by the loss of multiple packets in a row. The RTO timer is only updated when 1) no packets are in flight and a new one is to be sent, 2) a timeout occurs, or 3) an ACK is received (as discussed in section 3.3.1). So, if a packet is lost, the retransmission timer is not updated until it expires. In our experiment, if two packets were lost, a retransmission timeout would occur before the third packet was sent when the RTT was low. As the RTT increased, so did the number of packets that were sent between each RTO (and the RTO itself), thus the chance of the data arriving and being acknowledged before a timeout was higher. This, combined with the low loss rate, ensured that no retransmissions were triggered for the higher RTT values. The actual number of retransmissions

Variable RTT				
RTT (ms)	TCP (bytes)	RDB (bytes)	Difference, packet size (RDB - TCP)	Difference, packets sent (RDB - TCP)
500	121.7	481.0	359.3 (295.23%)	152.8 (1.23%)
400	120.8	395.4	274.6 (227.32%)	50.4 (0.40%)
300	120.1	334.1	214.0 (178.18%)	28.0 (0.22%)
280	120.9	277.1	156.2 (129.20%)	69.0 (0.55%)
260	120.4	269.9	149.5 (124.17%)	37.0 (0.29%)
240	120.1	254.6	134.5 (111.99%)	27.6 (0.22%)
220	120.0	243.2	123.2 (102.67%)	-37.7 (-0.30%)
200	120.0	241.0	121.0 (100.83%)	-37.0 (-0.29%)
190	120.0	240.8	120.8 (100.67%)	-70.8 (-0.56%)
180	120.0	240.4	120.4 (100.33%)	-57.2 (-0.45%)

Table 4.2: The worst-case packet size difference, RTT.

was rarely larger than three, which RDB managed to compensate for when the RTT was high.

Table 4.2 shows that a high RTT caused the packet size to increase significantly when RDB was used. This was mostly due to the increased time it took before an ACK was received. When the RTT increases and IAT is constant, more packets can be sent before an ACK arrives and the `cwnd` opens up. As a result, more packets are waiting outside the `cwnd`, the number of (possible) bundles increases, and consequently the packet size.

Even though the increase in average packet size was quite large, the packets were still small. The maximum size for a TCP packet that is to be transferred over an Ethernet is 1448 bytes (excluding headers, [11]), and the largest average packet was 481 byte. Another thing worth noticing in table 4.2, is that the difference between the number of packets sent by RDB and TCP increased along with the RTT. This was caused by the higher RTT, which forced the connection to wait longer before triggering a retransmission. More time passes before dupACKs are received, and the RTO timer increases along with the RTT. In addition, the high RTT causes the `cwnd` to have a slow growth rate. In other words, when a retransmission occurs, it takes a while before the send rate is back to normal. RDB, on the other hand, experienced close to zero retransmissions and was able to maintain a constant send rate. Hence, RDB sent more packets.

4.1.3 IAT and RDB

In the last section, we described how the number of retransmissions with RDB was independent of the RTT, and that RDB reduced the number of retransmissions for IATs between 140 ms and 160 ms. We also explained that the IAT must be less than the RTO minus the RTT for RDB to have

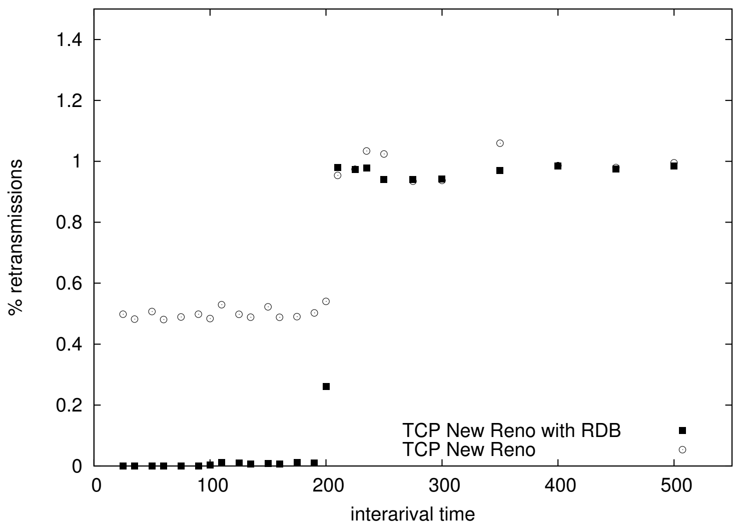


Figure 4.3: Retransmissions versus packet IAT when using RDB, loss = 1%, RTT = 100 ms.

maximum effect.

After checking how the RTT affected the number of retransmissions, we wanted to find out what effect variations in the IAT would have with RDB. For this test, we kept the RTT constant at 100 ms, which gave us an RTO value of around 300 ms. Thus, we expected RDB to be better than TCP until the IAT passed 200 ms, and then the number of retransmissions would be equal to TCP.

This is confirmed by figure 4.3. When there were (relatively) many packets that could be bundled, i.e., the IAT was low, no retransmissions were triggered because a copy of the data was sent (and acknowledged) before the potentially lost packet had to be retransmitted. However, the gain of bundling was reduced as the IAT increased. This was due to of the reduced amount of packets sent between each retransmission timeout.

When the IAT passed 200 ms, RDB started to retransmit as expected. This was caused by the IAT (plus the RTT) crossing the RTO value, and RDB was no longer able to "beat" the RTO timer and prevent retransmissions. There was at least one RTO between every sent packet, thus no bundles upon send were possible. Still, the connection was able to bundle on retransmission (provided that there was unacknowledged data), so RDB was able to deliver the data faster than TCP. However, 200 ms is longer than the average IAT for most of the applications in table 2.1. Therefore, RDB should be able to improve the application layer latency, and possi-

Variable IAT				
IAT (in ms)	TCP (bytes)	RDB (bytes)	Difference, packet size (RDB - TCP)	Difference, packets sent (RDB - TCP)
25	121.2	477.5	356.3 (293.98%)	574.0 (1.00%)
35	121.0	361.0	240.0 (198.35%)	193.1 (0.44%)
50	121.0	251.4	130.4 (107.77%)	162.2 (0.51%)
90	120.0	241.0	121.0 (100.83%)	-93.5 (-0.50%)
75	120.0	241.0	121.0 (100.83%)	-104.4 (-0.47%)
60	120.0	241.0	121.0 (100.83%)	16.9 (0.06%)
100	120.0	166.7	46.7 (38.92%)	-81.1 (-0.48%)
250	120.8	121.9	1.1 (0.91%)	-6.0 (-0.08%)
200	120.1	121.2	1.1 (0.92%)	-22.7 (-0.26%)
275	120.9	122.0	1.1 (0.91%)	0.3 (0.00%)

Table 4.3: The worst-case packet size difference, IAT.

bly also the user experience, in several thin stream applications (at least when faced with similar link characteristics). In the next sections, we try to determine if that is actually the case.

In table 4.3, we see that a low IAT causes RDB to generate much larger packets than TCP. This was, as with the RTT, caused by the increased number of packets sent between every ACK. The lower IATs generated the highest average packet size differences, which was expected, as this was when most bundles could be performed. The average packet size was more or less constant for TCP, which indicates that the built-in mechanisms are independent of the IAT (at least when the loss rate is low) and not very aggressive. Still, some bundling occurred for the lowest IATs. We believe this was because the large number of packets sent increased the chance of there being unacknowledged data at the sender. As a result, both of the built in mechanisms were able to bundle.

As seen in table 4.3, the connections with and without RDB transferred about the same number of packets (at least with our settings). Still, there were some difference between the IATs. When the IAT was short, RDB sent more packets than TCP. This was due to the larger number of packets sent from the application. Even though the loss rate was constant for all IATs, the interval between each retransmission was shorter with a short IAT. Thus, the connection using TCP spent more time waiting for retransmissions and the send rate was reduced more frequently. RDB experienced close to no losses and was able to maintain a nearly constant send rate, leading to a larger number of sent packets.

4.2 Perceived user latency

When faced with "typical" thin stream traffic and realistic network conditions, RDB produces significantly fewer retransmissions than TCP in our thin stream laboratory experiments. Unfortunately, the number of retransmissions does not say anything concrete about the user experience, it only provides an indication. To determine if RDB actually affects the end-user experience, we have to measure the latency at the application level.

The application layer latency says how long it takes before the received data can be used by the application. Since TCP is reliable, it has to deliver all the data to the application in order. Thus if a packet is lost, the application will have to wait for one or more retransmission before it receives more data. This wait might make the application behave unintentionally, and reduce the quality of the user experience. In a game, for example, if a player is under attack, delayed information from the server might lead to him or her being killed before being able to put up a defense.

The graphs presented in this section are called Cumulative density function-plots (CDF), and show the share of data that is delivered to either the transport layer (the data is received by the machine) or application layer (the data can be used by the application) within a certain time from being sent. Delays with a value above zero, represents packets that have been queued up at either the sender or receiver, waiting for lost data to be retransmitted or acknowledged. Our goal is to reduce the latency, i.e., we want to increase the share of received data at low delays.

4.2.1 Test setup

The different applications that we used when testing RDB are all non-deterministic. In other words, two sessions never generate the same traffic. To get directly comparable statistical results, we captured the traffic when different tasks were executed over a perfect link. The captured traffic was then fed to **tracepump** (see appendix A), and the links exposed to loss and delay using **netem**. We used the same setup as in section 2.3.1 to be able to fully control the links.

After running the "World of Warcraft"-test (section 4.2.2), we discovered that the links between the UiO and University of Massachusetts, Amhurst (UMASS) and a data center in Hong-Kong were close to perfect. This was not the case when we measured the performance of the same links from home, so we decided to run the rest of the tests through our emulated network. After performing several measurements between machines inside Norwegian and American access networks, we decided to use an RTT of

130 ms (30 ms more than in the lab environment tests) and loss rates of 2 % and 5 %. The RTT and first loss rate were the average, while 5 % was the worst case loss rate we experienced. We used a different RTT because 100 ms (as in the theoretical tests) was never achieved from our home connections, and we wanted settings as realistic as possible. Unfortunately, we did not have time to rerun the tests presented in section 4.1 and 2.3 with the new RTT. However, only the IAT test would be affected by the new RTT - the number of retransmissions would have increased when the IAT crossed 230 ms (instead of 200 ms).

In addition, we did some preliminary measurements against our machine in Hong Kong. This connection had the same loss rate, but a higher RTT. We chose not to use both RTTs in our tests because the number of retransmissions are, as discussed in 4.1.2, independent of the RTT (provided it is close to constant). The only major difference would have been lager packets due to the increased number of potential bundles, and all data would have been delayed (compared to a scenario with a lower RTT). However, since the CDFs show the relative (to the optimal delivery time) delay for the two streams, the plots would have looked the same.

Except for "World of Warcraft", all the replays were run in a loop for eight hours, and we performed four replays for each application. One for each of the two loss rates, and one with and without RDB. Since RDB is designed to be a sender side modification, it was only enabled on the sender (expect with SSH, as explained in section 4.2.4).

In addition to measuring different statistics, we wanted to find out how RDB (and the modifications presented in section 2.4) affects the user experience. As mentioned in the introduction to this chapter, since there was a limit for how long we could occupy the test subjects, we were not able to single out RDB in our tests. How these tests were conducted differed between the applications, and will be described in their respective sections.

4.2.2 World of Warcraft

The test

"World of Warcraft" (WoW) is currently the most popular MMORPG in the world [28]. It uses TCP and generates thin streams. We therefore decided to find out if RDB affected the end user experience (at least statistically). We captured a one hour playing session where we focused on traveling and fighting in the most populated areas in order to generate traffic. Our captured traffic has an average packet size of 270 bytes and IAT of 322 ms.

We replayed the captured stream in a loop over a 24 hour period using

tracepump. We used the client as the sender to make sure that we had the original traffic. If we had used the server, a packet could, for example, originally have been a retransmission and we would not have been able to replay the original stream. All tests were performed at the same time, but from different (but similarly equipped) machines. When these tests were performed, we had not yet implemented the I/O control. Because proc-variables are global and therefore affect the entire system, we needed one machine for the TCP tests and one for the RDB tests.

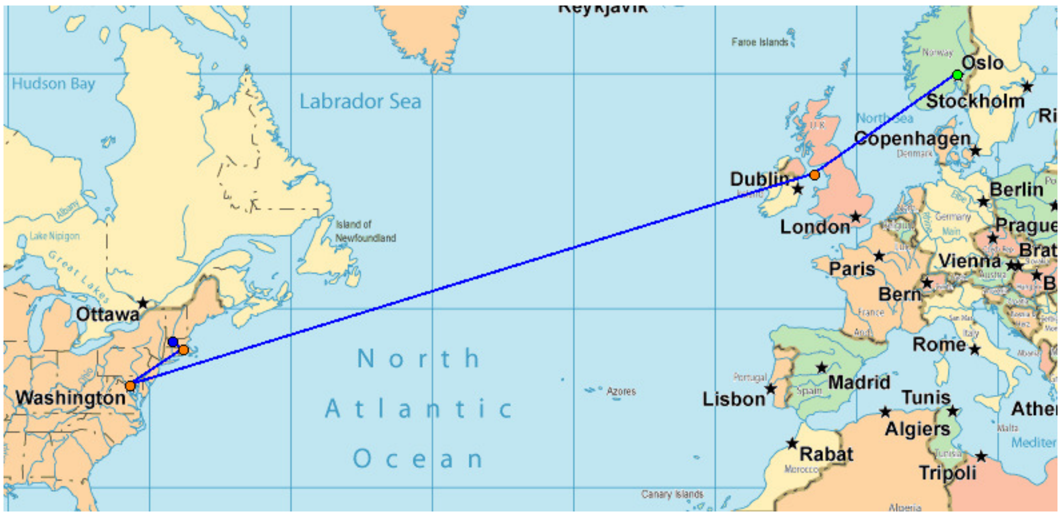
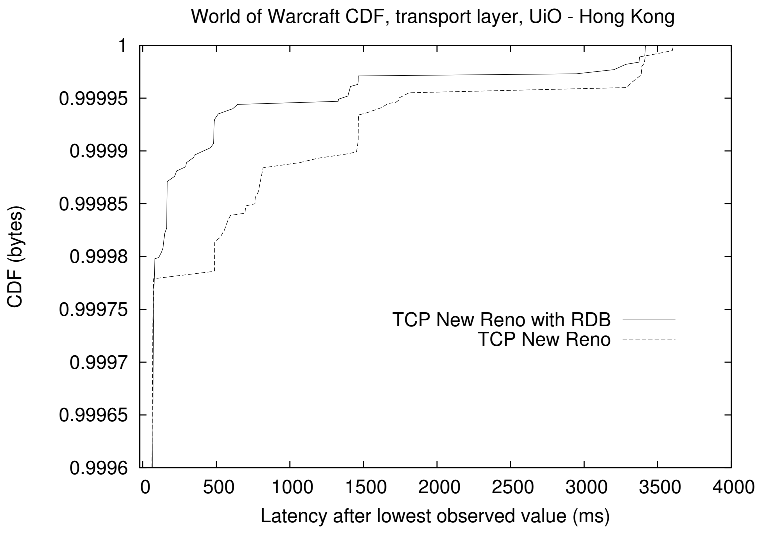


Figure 4.4: The route our packets followed to UMASS.

To get the most accurate values for the effect of RDB on the user experience, we decided to run the tests over the Internet. The sender machine was located at the UiO, while the receivers were at UMASS (path shown in figure 4.4) and a data center in Hong Kong.

Results

Figure 4.6 show the transport- and application layer delay to UMASS. Both layer's latency was lower with RDB than TCP. The receiver's transport layer latency received 99.9 % of the data within 337 ms with TCP. For RDB, the value was 158 ms. The difference at the application layer was more significant. With RDB, the application layer delay (for the same share of data) was equal to the transport layer delay. TCP, however, could not deliver 99.9 % of the data to the application before 342 ms had passed. These results were confirmed by our Hong Kong test, shown in figure 4.5. 99.98 % of the data was received by the machine after 127 ms with RDB,



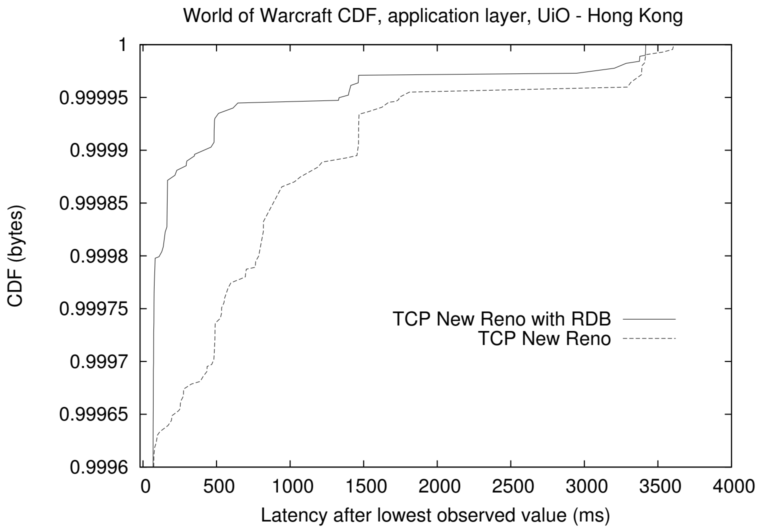
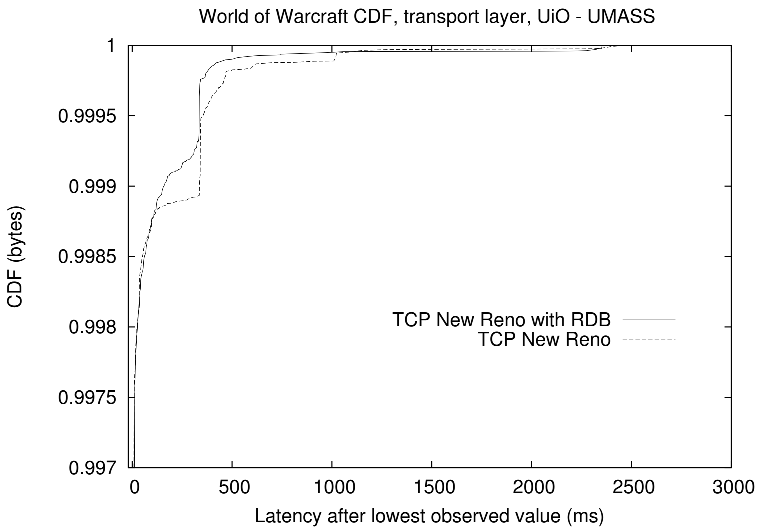


Figure 4.5: Transport and application layer latency differences running WoW 24 hours to Hong Kong (19/11-2007).

while TCP did not reach the same share until 489 ms. The same share of data was consumed by the application after 128 ms with RDB and 818 ms with TCP. Because the application layer latency is the most important, we only include graphs showing and discuss this latency in the rest of the thesis. What directly affects the user experience is when the data can be used by the application, and not when it is received by the transport layer.

However, neither of the differences were large compared to the total number of bytes transmitted. This was due to the high quality of the links,



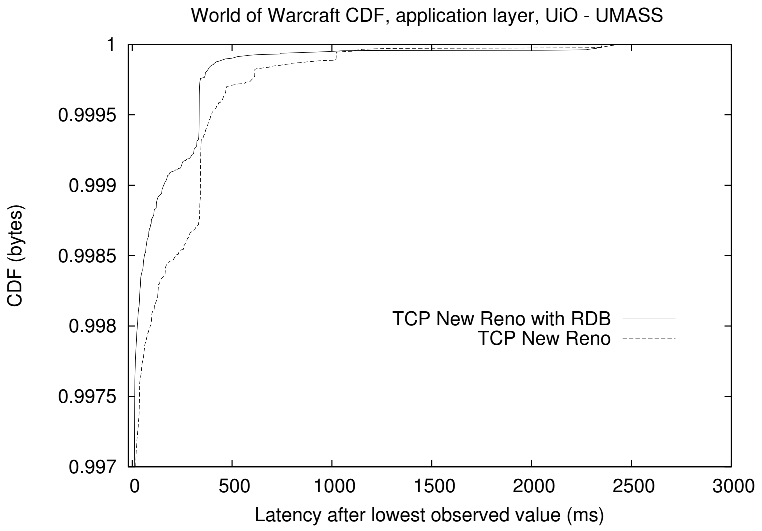


Figure 4.6: Transport and application layer latency differences running WoW 24 hours to UMASS (19/11-2007).

both to the US and to Hong-Kong. We had an average loss rate of 0.06% and RTT of 122 ms to UMASS, and 0.03 % loss and 261 ms RTT to Hong-Kong. Thus, few retransmissions were triggered and RDB did not have that much of an effect. Even though the links were close to perfect, we decided to present the result. If a player is killed because the final “Attack” command was lost, he does not care if most packets before and after arrived quickly. In other words, we wanted to see if we could improve the perceived user latency on high quality links as well.

Because RDB delivered data faster than regular TCP, it decreased the latency. Since we do not control the server, we were not able to see if the application behavior changed. Had we run the tests from locations with poorer links (e.g. at home), we would probably have seen a more significant difference between when RDB was used and not. When pinging the two machines several thousand times at various times of the day, we always experienced a couple percent loss when using our home connections.

4.2.3 Skype

The test

Skype[1] is a popular and free VOIP-application. By using it you can call both other Skype users and regular phones, and Skype is able to use both UDP and TCP to transfer data. The former is default, but Skype falls back to TCP if UDP is blocked (e.g. by a firewall). Since a lot of corporate firewalls block UDP, TCP often is the only alternative. We were curious as to wheter RDB would enhance the user experience.

In World Of Warcraft we had no control over the server, other receivers or links. With Skype, however, we can fully control every aspect of a connection and experience if our different TCP modifications enhance the user experience. To do this, we made recordings of both a podcast and two songs when played over Skype through our emulated network environment (for 2 % or 5 % loss and the modifications on and off). The songs were chosen because it is much easier to hear delays and other effects when you have a rhythm. When the recordings were completed, we uploaded them to a web page [2] and asked people to vote for the clips they thought were the best. We decided to use one of the clips as a reference clip, and uploaded the same version twice but under different names. By doing this, we could detect if people tended to prefer the first alternative.

One important thing that we discovered is that Skype performs unexpected when encoding sound for transfer over TCP. Gaps and sound corruption occur even when there are no loss or delay. Since all the recordings would be affected by the same sound corruption, we decided that this would not affect the result and went ahead with the test. Unfortunately, since Skype is a proprietary and thereby closed source application, we were not able to investigate the cause of this further.

[1]Can be downloaded from http://www.skype.com

[2]A copy of the webpage is included on the attached CD-ROM.

Connection information, Skype	Share of retrans.	Average packet size (in bytes)	Latency (in ms)			Unique number of bytes	Total number of bytes	Average IAT (in ms)	Number of packets sent
			Min.	Avg.	Max.				
2% loss, TCP	1.97%	322	130	149.8	11018	195279922	199334594	37	777882
2% loss, RDB	0.03%	1154	130	132	22018	195279922	901149955	35	828250
5% loss, TCP	4.88%	381	130	182.8	22772	195279922	206132337	45	653392
5% loss, RDB	0.18%	1124	130	136.6	11015	195279922	872352828	35	824109

Table 4.4: Information gathered from our Skype-experiments, 130 ms RTT

Before we present the statistical results and the results from the rest of our tests, we will repeat what a time-dependent thin stream actually is. In chapter 2, we have defined thin streams as streams with high IAT or small packet size (compared to "regular" TCP streams), thus they will consume small amounts of bandwidth. Thin streams are often generated by applications with strict latency requirements, and TCP's different congestion control mechanisms will lead to an increased latency, thereby hurting the performance of time-dependent thin streams.

Statistical results

Skype is a typical thin stream application, it generates small packets and sends them out with short IATs. In addition, Skype does not use a popular technique called "noise suppression" when UDP is blocked, which means that the application transfers data even when nobody is talking. In the original dump (the one that was fed to **tracepump**), we had an IAT of 19 ms, which is very low and ideal for RDB (together with the small packets).

Table 4.4 summarizes our findings from the Skype-experiments when using our emulated network. As we can see, RDB reduced the number of retransmissions significantly, but at the cost of bandwidth. The packets were close to four times larger than with TCP, both for 2 % and 5 % loss, and around 700 MB more data was transferred for each loss rate (an increase of 361 % and 323 % respectively). This quite significant increase was due to the low IAT and small packet size, which combined with the RTT allowed several bundles to be performed before an ACK was received or retransmission occurred.

The number of packets sent did not increase as significantly as the amount of bytes when we used RDB. For 2 % and 5 % loss, the increase was 6 % and 26 %. This increase was caused by the low IAT, as discussed in section 4.1.3. TCP has to stop and wait for retransmissions more frequently than RDB, which together with the effects of a retransmission led to a lower send rate.

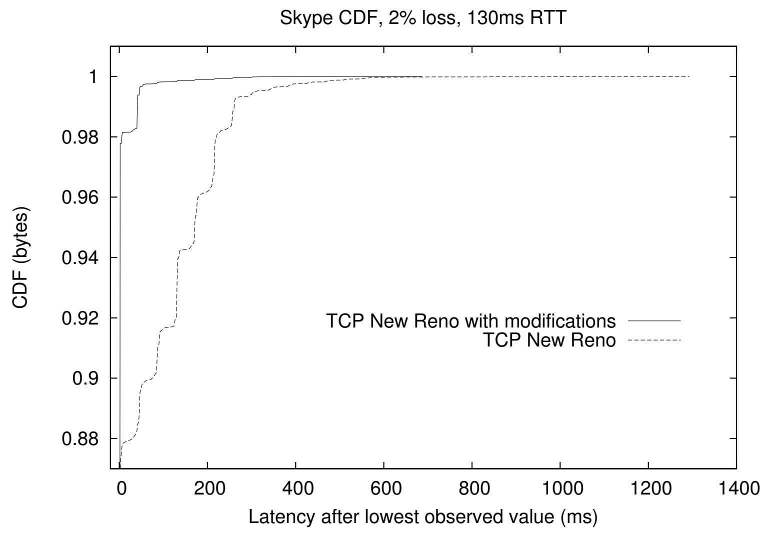


Figure 4.7: Skype application layer CDF, 2 % loss and 130 ms RTT.

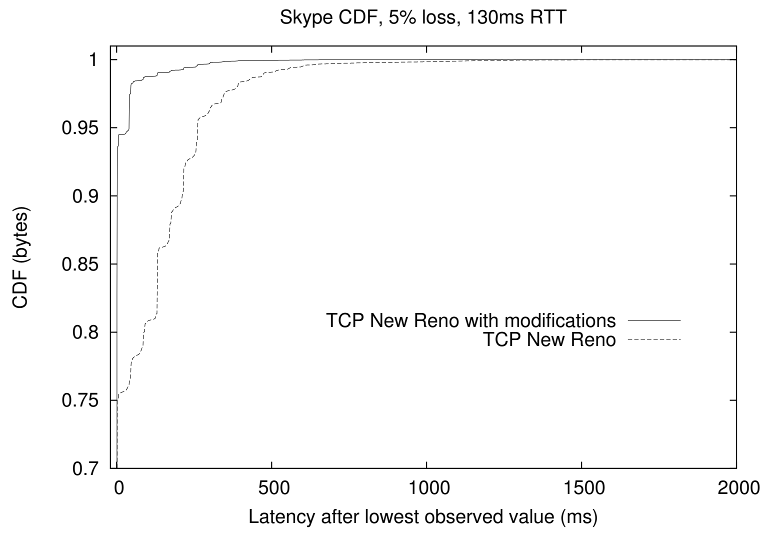


Figure 4.8: Skype application layer CDF, 5 % loss and 130 ms RTT.

RDB also reduced the transport layer latency. For example, with TCP we saw an average latency of 149.8 ms when we had 2 % loss, compared to 132 ms when RDB was used. The application layer latency was also reduced, as shown in figure 4.7 and 4.8. With 2 % loss, 98 % of the data was delivered to the application almost instantly (i.e., the only delay was the RTT). With TCP, it took 221 ms before the same share of data was delivered. The difference was even larger in the tests with 5 % loss. With RDB, 98 % was delivered within 46 ms, while TCP did not reach the same level before 388 ms had passed.

Since the data was delivered faster, we expected the quality of a conversation performed over a link with the same characteristics to increase. Due to the reliability of TCP, a connection have to stop and wait for retransmissions if packets are lost. When talking to someone in real-time, we believe this leads to delayed speech and decrease the quality of the conversation. For instance, the person at the other end of the line might have started talking about something else, or be annoyed because he or she has to wait so long for a reply. When RDB is used, our statistical results indicate that the number and length of the delays are going to be reduced. Hence, the speech arrives faster, leading to a better user experience.

The high maximum latency values were caused by multiple losses of the same packets. The captured data from the experiments tells us that most of the retransmissions have been timeout retransmissions, thus exponential backoff have kicked in and doubled the RTO for every loss. This is something that RDB is not able to compensate for. If data does not arrive or ACKs are lost, then it is impossible to do anything. In other words, the maximum latency will be the same as TCP.

User experience

Figure 4.9 shows the results of our Skype survey. We got 88 people to vote, and they preferred the clips with the modifications. At least for the third clip, the majority voted for the version with the modifications turned on. According to the comments we got, the differences in the first clip were much more difficult to hear. Still, over half of the people that voted chose the version with the modifications enabled.

The votes for the reference clip indicate that users tend to prefer the first version they hear, and this might have affected the results for clip one. If people went through the test linearly, then the version with the modifications was the second they heard. With clip three, the order was the opposite of clip one, but the tendency to vote for the first version would most likely not matter. As shown in figure 4.9, over 90 % said the version

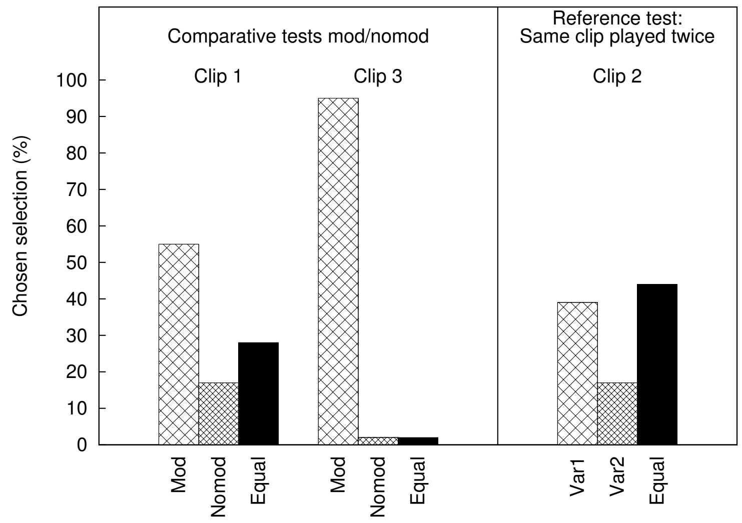


Figure 4.9: Results from the Skype user survey.

with the modifications switched on was the best.

The effects of running Skype over a connection with both loss and delay were not as severe as we thought (see the second to last paragraph in the previous section). Most of the people that took the test commented that the sound quality tended to be worse without the modifications (more background noise) and the sound often sped up. The latter we suspect was caused by Skype trying to regain some of the time spent waiting for a retransmission. With the modifications enabled, people reported that even though the sound was not perfect, the overall quality was much better.

Unfortunately, 88 persons are not enough to draw a reliable conclusion to whether the modifications have an effect or not. But, due to the vast majority voting for "our" clips, we believe that the trend would continue if we asked more people. Thus, we got a strong indication that the modifications actually improve the user experience.

When we looked at the captured network traffic from when we made the recordings, it had the same characteristics as those presented in table 4.4 and discussed in the previous section. As our statistical results show, RDB reduces the latency when faced with traffic generated by Skype (at least with our network settings), thus it is most likely contributed a lot to improving the user experience. The other modifications, as discussed in section 2.4, only kick in when less than four packets are unacknowledged at a time. If one packet was lost, the low IAT indicates that more than four packets were sent before any of the other modifications would have been activated. Also, as shown in section 4.2.3, when we enabled RDB we hardly had any retransmissions, making the other modifications more or less redundant. In other words, they did not contribute very much to the result.

4.2.4 SSH

The test

SSH, or Secure Shell, is a popular application level protocol that allows data to be exchanged over a secure channel between two computers. It is built upon TCP, and the applications that use it, SSH clients, are most commonly used for logging into Unix/Linux-machines remotely. When logging into a machine, you have access to a console and, providing the remote machine supports it, the X window system. Thus, if the application has a GUI, you can opt to use that instead of the text only version. Captured traffic from different SSH sessions shows that when we use only the command line, we have a typical thin stream. When X comes into play (the user starts a program that has a GUI), the stream is still thin but the packets are so large that RDB is not able to bundle. We have therefore only focused on the command line and applications that can be run directly in the console.

As with Skype, we have access to both the sender and receiver, and can experience any possible enhancements our self. We decided that both ourselves and 25 users should perform some specific tasks to determine if RDB (and the other modifications) had an effect. For the theoretical tests, we logged into a machine, changed directory, used “ls” to show the content of a large directory, and finally typed a small text into the a text editor. We captured the network traffic when we performed these tasks, and replayed it as described in section 4.2.1.

The user’s tasks differed from ours in that they focused more on typing. As typing is interactive it was much easier to notice differences in quality. Due to time constraints we had only time to test for 2 % loss, 130 ms RTT and with/without all the modifications (no specific RDB test). After the test was done, the user was asked if the first or second session was the best. He or she was not aware of which session was with the modifications, and we changed the order before every test.

Since both machines acted as senders, we activated the modifications on both. “Local echo” was switched off by default, so one machine sent the input, while the other transmitted what was to be shown/updated on the screen. All key presses were transmitted to and returned from the other

Connection information, Skype	Share of retrans.	Average packet size (in bytes)	Latency (in ms)			Unique number of bytes	Total number of bytes	Average IAT (in ms)	Number of packets sent
			Min.	Avg.	Max.				
2% loss, TCP	2.25%	167	130	148.7	22930	17547400	18018736	166	178360
2% loss, RDB	0.48%	223	130	136.6	10800	17547400	27886416	166	177218
5% loss, TCP	5.53%	171	130	181.9	21739	17547400	18743912	166	178030
5% loss, RDB	1.38%	228	130	146.5	10954	17547400	28992248	166	177921

Table 4.5: Information gathered from our SSH-experiments, 130 ms RTT

machine before being displayed.

Statistical results

As mentioned briefly in the introduction, SSH is another typical thin stream application. In our "perfect" packet dump, the average IAT was 83 ms and packet size 116 bytes. With a constant RTT of 130 ms, we had an RTO of around 330 ms. Thus close to four packets would be sent between every RTO. In other words, RDB should be able to reduce both the latency and number of retransmissions.

Unfortunately for RDB, table 4.5 shows that the average IAT increased by over 30 ms when we introduced some loss and delay. With an IAT of 166 ms on average, only one packet would be sent before an RTO occurred. Still, RDB should be able to improve both the average application layer latency and reduce the number of retransmissions. If the packet sent between the first packet and the RTO contained the lost data, it would be acknowledged before a timeout occurs (provided that neither it nor the ACK is lost).

As shown in table, 4.5, RDB improved the average latency with 8% (12 ms) for 2% loss and 33% (35.4 ms) for 5% loss, which should make the application behave more pleasantly. In addition, the share of retransmissions dropped with 1.77 % and 4.15 % respectively.

The maximum latency was, as with Skype, extremely high and had the same reasons (multiple, consecutive packet losses). As shown before, the cost of reducing the latency is increased bandwidth. The average packet size when using RDB was around 50 bytes larger, and in total the RDB connections transferred close to 10MB more than TCP for both loss rates (an increase of 49 % and 56 % for 2 % and 5 % loss). If somebody uses SSH over a connection where they have to pay for the amount of data transferred, this might be an issue. But unless somebody has a long and very active session, the increased cost will be slim.

Unlike Skype, the connections using RDB transfered fewer packets than the ones without (a decrease of 0.6 % and 0.1 % for 2 % and 5 % respectively). The IAT was close to constant in all the tests, so the only thing that differed was the loss rate. As discussed in section 4.1.1, RDB transfers fewer packets than TCP for most lower loss rates (like 2 % and 5 %). The high number of retransmissions forces TCP to send additional packets containing the lost data, something that RDB most of the time manages to avoid. For example, the share of retransmission with 2 % loss was 1.97 % with TCP and 0.03 % with RDB.

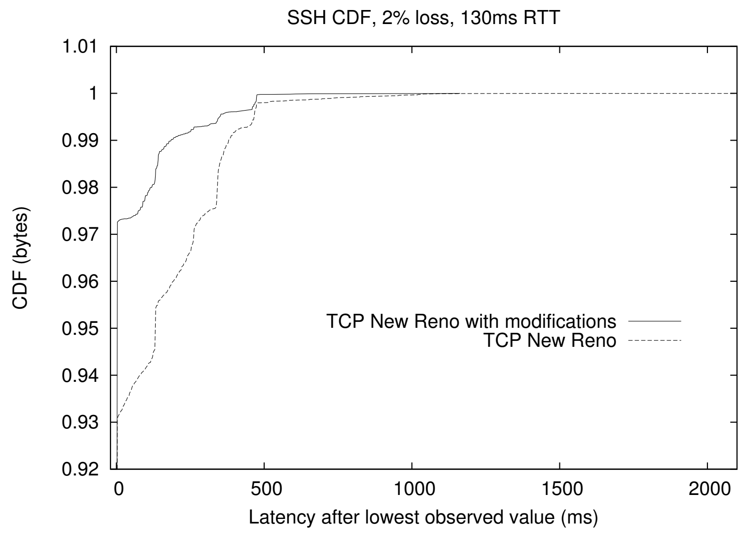


Figure 4.10: SSH application layer CDF, 2 % loss and 130 ms RTT.

Figure 4.10 and 4.11 show that the application layer latency gains from using RDB. The connection received 97 % of the data instantly with 2 % loss, and 94 % with 5 % loss. TCP did not reach the same levels before 400 ms had passed for both loss rates. This reduced application layer latency should lead to a better user experience. If the latency is high, it takes a while before the keyboard strokes arrive and the remote machine reacts. Also, if a packet is lost, a connection using TCP has to wait for a retransmission before it can process more commands. Hence, we expect characters to arrive more bursty when RDB (and the other modifications) are switched off.

User experience

In our user tests, the test subjects were, as mentioned in the introduction, asked to perform different tasks focusing on interactivity. The modifications were run on both the local and remote machine, and the connec-

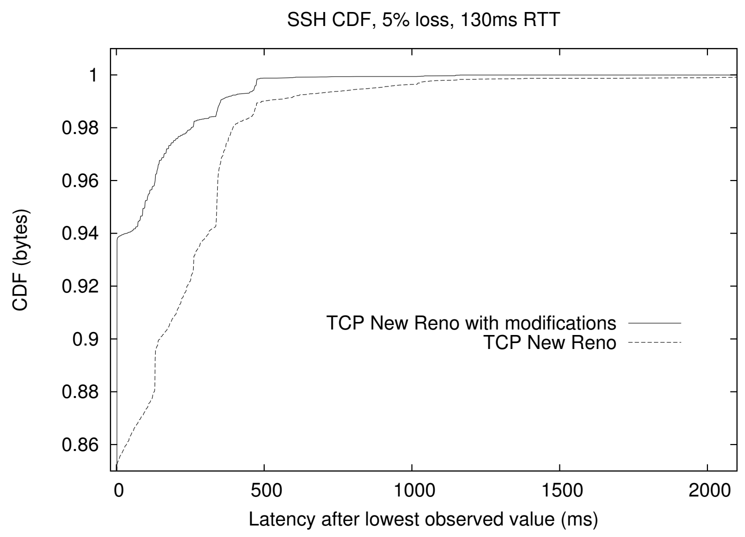


Figure 4.11: SSH application layer CDF, 5 % loss and 130 ms RTT.

tion faced 130 ms RTT and 2 % loss in each direction (emulated through **netem**). After we had set up an emulated network with these settings, we had 25 people perform the tasks.

The results in figure 4.12 show that the test subjects clearly preferred to have the modifications switched on. They commented that the input was smoother. When the modifications were switched off they noticed the effects described in the last paragraph of section 4.2.4. The input was burstier, for example, the remote machine did not immediately notice when a key was released.

As with Skype, we captured the network traffic from the different test sessions. This traffic had the same characteristics as in table 4.5, and we believe RDB contributed a lot to improving the user experience. In our statistical experiment, RDB reduced both the share of retransmissions and the average latency. However, unlike in the Skype test, the other modifications were able to affect the connection positively as well. This was due to the low number of packets in flight (because of the high IAT).

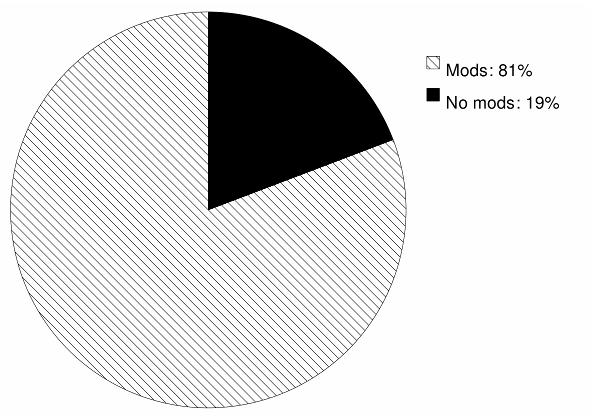


Figure 4.12: Results from the SSH user survey.

4.2.5 BZFlag

The test

BZFlag[3] is short for BattleZone capture the Flag, and is a popular open-source multiplayer shooter game. The goal is to capture the other team(s) flag and bring it back to your base as many times as possible. The game can use both TCP and UDP, and as most first person shooter games it is fast and intense. In other words, you want the packets to be sent and arrive as fast as possible to get the most accurate positions of the other players, and make sure that your actions are quickly pleased.

All clients are connected to a central server, which is responsible for broadcasting messages sent from the different players. If you fire a shot or move, the packet is first sent to the server, and then to all the other players. If the server has a lossy connection to the network, it affects all the players in the game. Shots are delayed, the difference between the perceived and actual position of another player might be quite large, and so on.

To generate the dump used to get our statistical results, we decided to pit one player against 20 computer controlled opponents (the maximum our machines could handle). The opponents were located on another machine to force packets through the server, and the play sessions lasted for 30 minutes. We captured the data and then replayed the stream between the server and the human client through our emulated network (as described in section 4.2.1).

Unlike Skype and SSH, we did not have time to perform any formal surveys to find out if RDB (and our other TCP modifications) enhanced

[3]http://www.bzflag.org

Connection information, Skype	Share of retrans.	Average packet size (in bytes)	Latency (in ms)			Unique number of bytes	Total number of bytes	Average IAT (in ms)	Number of packets sent
			Min.	Avg.	Max.				
2% loss, TCP	1.98%	112	130	150	21930	35482101	36223120	38	771646
2% loss, RDB	0%	267	130	131.5.6	10989	35482101	233969911	25	1159202
5% loss, TCP	4.91%	129	130	182.9	22050	35482101	37497492	50	593143
5% loss, RDB	0.02%	269	130	133.5	22061	35482101	235372004	25	1154273

Table 4.6: Information gathered from our BZFlag-experiments, 130 ms RTT

the user experience. Instead, the user experience section is based on how we felt that the game behaved with the modifications turned on. Since we always knew if the modifications were enabled or not, our opinions were colored. Still, we have been as objective as possible. The impressions are gathered from several sessions with different loss rates (either 2 % or 5 %) and 130 ms RTT. Since the modifications are intended to be sender (i.e., server) side to avoid imposing any requirements on the clients, they were only enabled on the server.

Statistical results

As mentioned above, BZFlag (and other shooter games) are very sensitive to latency [3]. If packets are lost/delayed, the built-in movement prediction has to cover a longer time period. When the lost packets finally arrive, the tanks in the game have to jump into the correct position. This difference between the perceived and actual position of the other players leads to a less than optimal user experience. Even though the bullet flies right through the opponents tank and he appears to be hit, he might in fact be somewhere else on the map and the player misses.

In the sample dump that we fed to **tracepump**, the average IAT was 25 ms. As in the other tests, an RTT of 130 ms gave us an RTO of 330 ms, hence, several packets were sent between each RTO and bundles could be made. Due to the low IAT, we expected RDB to reduce the latency quite significantly, something the results in table 4.6 show. With 2% loss, the average latency was close to 14% lower, while with 5% loss it was reduced by 37%. Unfortunately, as with Skype, the bandwidth use increased quite significantly. The packets were on average close to 2.5 times larger than when RDB was switched off, and close to 200 MB more data was sent for both loss rates (an increase of 528 % and 546 % respectively).

The connections using RDB also sent more packets than the others, but the increase was not as significant as for sent bytes. With 2 % and 5 % loss, RDB caused, 50 % and 94 % more packets to be sent respectively. This was, as in the Skype test, because of the low IAT (discussed in section

4.1.3). Packets are lost more frequently, forcing TCP to stop and wait for a retransmission. In addition, a retransmission reduces the send rate significantly (because of the congestion control mechanisms). RDB, on the other hand, experienced close to zero retransmissions and was able to maintain a much higher send rate, thus sending more packets.

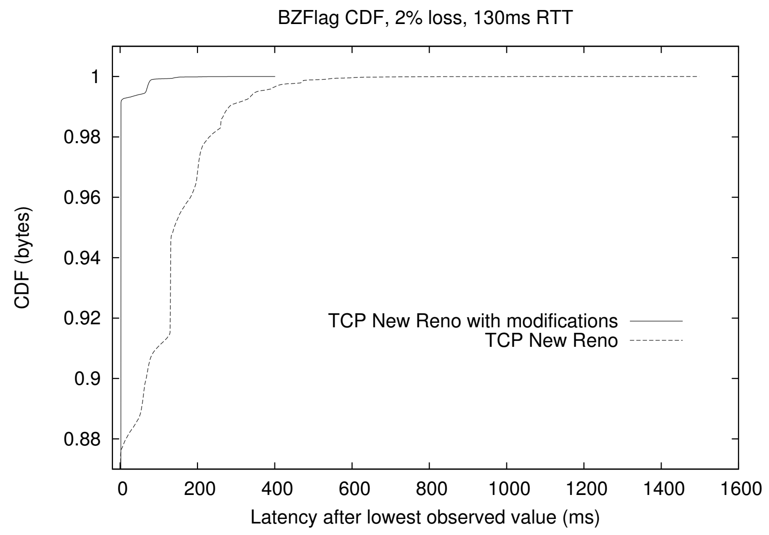


Figure 4.13: BZFlag application layer CDF, 2 % loss and 130 ms RTT.

Figure 4.13 and 4.14 show that the application layer latency also improved, i.e. data was delivered to the application faster when using RDB. With 2% loss, over 99% of the data was delivered almost instantly, something TCP did not achieve until 283 ms had passed. For 5% loss, TCP did not manage to deliver 98% until 393 ms, while RDB reached the same level almost at once. Because the data was delivered faster, we expected the difference between the actual and perceived position to decrease (under the same network conditions). This should in turn lead to less erratic tank movement and an improved hit ratio. It is more likely that the other tanks actually are where they appear on the screen. This will, in turn, lead to a more pleasant user experience.

Most games do all the logic on the server, and RDB would be most efficient if used on this machine. However, in BZFlag, the clients themselves are responsible for deciding the outcome of all actions and then reporting back to the server. If one of the clients has to deal with a link of poor quality, it hurts the experience for all the other players because of the delayed reporting. Hence, it would be useful if the clients enabled RDB as well, but due to time constraints, we did not have time to look at how large the

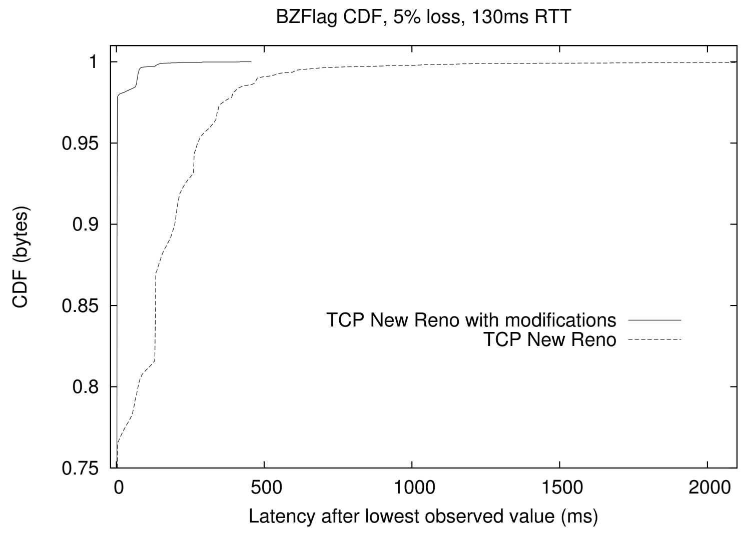


Figure 4.14: BZFlag application layer CDF, 5 % loss and 130 ms RTT.

(possible) improvement might be.

User experience

Unlike Skype and SSH, we did not have time to perform a proper user survey for BZFlag. Instead, we gathered several people, and played a number of matches against each other with different network settings and the modifications on and off. We tested both 2 % and 5 % loss, and the RTT was kept constant at 130 ms. The modifications were only enabled on the server.

Our impression was that the modifications made it much easier to hit the opponents, as expected. The difference between actual and perceived position was apparently much smaller with the modifications. In addition, we saw less erratic tank movement because BZFlag did not have to estimate so much.

The traffic showed similar characteristics as the one in table 4.6, and we believe RDB played a major part in improving the experience. The low IAT and small packet size were ideal conditions for RDB. With an RTT of 130 ms, several packets were sent for every RTT and before an RTO. Hence, bundles were possible, and the chance that the data arrived sooner than with TCP increased.

Since we knew all the settings and when the modifications were switched on, we were not fully objective. However, we felt that the modifications improved the user experience. It was easier to hit the opponents, and the tanks moved more smoothly. The modifications, in our perception, made playing BZFlag over a lossy link more pleasant.

4.3 Discussion

In this chapter, we have presented the results of several tests performed to find out if RDB improves the latency for thin streams. Since most of the applications generating thin streams are interactive and therefore non deterministic, we captured the network traffic (with **tcpdump**) from a sample session, and then replayed it through our controlled network environment (see section 2.3.1) using **tracepump**. Finally, we compared the results for each loss rate when RDB was and was not enabled. To impose loss and delay on the links, we used the **netem** module of the **tc** application, using a uniform loss pattern. In other words, loss did not occur at the same places in our experiments, but the results shows the average scenarios.

As the statistical results show, RDB improves the latency of certain applications that generate thin streams. If the IAT is less than the RTT, bundles can be performed because the next packet is sent before the previous packet is acknowledged (provided that the combined packet size is less than half the MSS). If the previous packet (or ACK) is lost and a bundle is made, the data is delivered faster to the receiver than with a retransmission. If the IAT is less than the RTO minus RTT (see section 4.1.2), using RDB also reduces the number of retransmissions (as long as the packet size criteria is met). The following packet is both received and acknowledged before an RTO is triggered. The smaller the IAT, the more bundles can be made, further reducing the number of retransmissions.

In addition to improving the latency, RDB (together with two other TCP modifications) also improves the user experience for some typical thin stream applications. The results from our survey indicates that there is a noticeable difference between when the modifications are switched on and off. Unfortunately, RDB is not able to aid all applications. For instance, we did tests with different video- and audio streaming servers/-clients, but they sent so large packets that bundles were impossible. Another problem was that some applications, most typically slower games like strategy games, had such a high IAT that the RTO expired before the next packet was sent.

Unfortunately, the benefits of RDB do not come for free. The packet size increases due to the bundling, and RDB consumes more bandwidth than TCP. As shown in both the Skype and some of the theoretical tests, the average packet size is almost four times as large. Thus, if somebody uses for example Skype over UMTS or GPRS and pay for the consumed bandwidth, they can only talk for about one fourth of the time. For people that are connected through "regular" networks, the increased bandwidth usage is not that much of an issue. The packets never grow beyond the MSS, hence, the network should be able to handle them without any performance or cost penalty (unless the person is connected to a very slow link or network). If many concurrent connections made across the internet enable RDB, the ISPs might start to complain though. Their equipment has to transfer more data, and in some cases also more packets. This might lead to higher costs for them.

Another problem that RDB is not able to fix, is if the connection experiences several losses in a row (bursty loss). If no packets arrive at the receiver, then no packets are acknowledged, and RTOs will be triggered. RDB does still help though, due to the bundling on retransmissions, the first retransmitted packet contains as many of the following packets as possible. Hence, the data is delivered faster than with TCP, which only merges the lost packet with the next packet on the output queue. In other words, RDB improves the latency and user experience when a retransmission occurs. Unfortunately, due to time constraints and technical difficulties with **netem** we were not able to simulate bursty loss. The lack of bursty loss also significantly reduced any effect of SACK, due to the lack of multiple packets losses it was not able to improve the performanc. In addition, when using RDB, lost data would most of the time arrive with the next packet, resulting in almost no gaps in the received byte range.

As discussed in section 2.2.2, one important aspect of TCP is the fairness principle. If N TCP streams share the same link, then they should each get an equal share (1/N) of the link capacity. Unfortunately, RDB does not uphold this principle. When competing with other TCP streams, it potentially uses a larger share of the link. This is because it will not be as severly affected by packet losses. A lost packet forces TCP to retransmit, hence, forcing it to stop sending new packets (while waiting for the retransmission to be triggered) and the packet send rate is reduced. RDB does not experience this if the next (or one of the following) packet contains bundled data, and is received and acknowledged before a retransmit is triggered. Thus, the packet rate may be higher than with regular New Reno, and a larger share of the link may potentially be used. Therefore, it is not fair to other streams. This lack of fairness is visible in the results from both the BZFlag- and Skype-experiments (table 4.6 and 4.4 respectively). In the BZFlag-experiment, the average IAT was almost half that of the IAT when RDB was switched off with both 2 % and 5 % loss, while

	TCP	RDB
Latency	-	+
Bandwidth consumption	+	-
Fairness	+	-
User experience	-	+
Friendliness	+	?

Table 4.7: Pros and cons of RDB compared to TCP when it comes to time dependent thin streams.

with Skype, it was reduced by almost 10 ms for both loss rates. Also, the connections with RDB enabled tended to send more packets than the ones who used TCP.

Fortunately, the violation is not that great. We do not alter the different congestion control mechanisms, so when a packet loss is detected, RDB is struck just as hard as any other stream by, e.g., exponential backoff. In addition, even though RDB generates larger packets and therefore transfers larger amounts of data, it is still limited by the congestion window and has to go through slow start and congestion avoidance.

Another interesting scenario that we did not have time to test properly, was if RDB had any effect on other TCP streams (known as TCP Friendliness). For example, the increased number of packets might consume too much resources in the routers and decrease the performance of other connections, or reduce the performance of other streams in the sender machine, and so forth. Preliminary tests showed that even though RDB was not fair, other streams were not affected by it. We ran two instances of a replayed dump containing several hundred connections between two machines at the same time (using the same setup as in 2.3.1), and they performed similarly as to when they had the machines and the network to themselves. Thus, we at least have an indication on that TCP New Reno with RDB is TCP friendly.

We have summarized this discussion in table 4.7. Here, we see that if somebody wants a reduced latency and provide a better user experience, he or she should use RDB (at least if the application has similar network characteristics to the ones we used). On the other hand, if bandwidth is an issue or the person is worried about fairness/friendliness, TCP would be the preferred choice.

4.4 Summary

In this chapter, we have presented our findings from several experiments to measure the performance of RDB. We have shown that the modification were able to decrease the latency for several thin stream applications, both statistically and when it comes to the actual user experience. Unfortunately, this comes at a cost. When using RDB a TCP connection consumes more bandwidth and often sends more packets, and is not fair to other TCP streams. By using RDB, you trade-off bandwidth for latency, which, depending on the scenario, may or may not be acceptable.

In the next chapter, we conclude our work, as well as provide some ideas for further work and interesting experiments.

Chapter 5

Conclusion

5.1 Summary

In this thesis, we have addressed the challenges present in TCP for supporting interactive, time-dependent thin stream applications. In this context, a thin stream consists of small packets or high IATs, something TCP is not tuned for [1]. This comes as a result of TCP development focusing on improving performance for throughput intensive streams, with little regard for timeliness requirements. The protocol's congestion control will, as such, inadvertently contribute to an increase in latency when a packet loss occurs, which is particularly harmful for time-dependent traffic.

As a remedy we have implemented RDB in the Linux 2.6.22.1-kernel and it has been evaluated experimentally and compared with TCP New Reno, which was shown in [1] to be the TCP protocol variation that performs best in a thin stream scenario. We looked at how varying the loss rate, IAT and RTT affected the number of retransmissions (and thereby the latency), and how RDB (and TCP) performed when faced with traffic from several applications generating time-dependent thin streams. In addition, we conducted a user survey to determine if our TCP modifications (RDB and those presented in section 2.4) improved the actual user experience.

5.2 Main contributions

Time-dependent thin streams generate packets that are several times smaller than the MSS. RDB takes advantage of this fact, as it is a modification to TCP that utilizes free space in the packets and reduces the latency. By bundling as much unacknowledged data as possible, without leading to fragmentation, into each packet that is to be sent, we hope to deliver lost data quicker than if the connection had to wait for a retransmission. Since this data may have been received already (the ACK might be lost or delayed), it is potentially redundant.

As shown in chapter 4, RDB is able to reduce the latency, provided that the combined packet size never exceeds the MSS. Also, if the IAT plus the RTT is less than the RTO, it also reduces the number of normal retransmissions. Reducing the number of normal retransmissions is desirable because TCP's congestion control mechanisms often contribute to an increased latency. For example, exponential backoff will double the time it takes before a packet is retransmitted for the second time. Likewise, often having a high IAT, many thin streams will never be able to trigger a fast retransmit, at least not until the RTO timer exceeds the time it takes to receive the required number of dupACKs.

By reducing both the transport- and application-layer latency, when the data is received and when it is ready for use by the application, RDB (along with the two modifications presented in chapter 2.4) also improves the user experience for a number of applications generating time-dependent thin streams. We conducted a survey in which we asked people how they experienced the behavior of two interactive applications that generated time-dependent thin streams with and without the modifications. The majority answered that the experience was best when the modifications were enabled. An analysis of the traffic showed that RDB contributed significantly to this result.

Unfortunately, the reduced latency comes at a cost to bandwidth. Because of the bundling, the packets would be larger than those sent with TCP. However, in our experiments, the increase in number of packets was never as large as the increase in average packet size. Thus, even though we send more bytes, the network would not have to deal with that many more packets. In addition, RDB would often generate fewer packets, as shown in table 4.1, 4.2 and 4.3.

5.3 Future work

Even though we have performed extensive tests with RDB, there are some areas that we have not investigated properly. The most important are related to TCP fairness. We have determined that RDB is not fair since it is able to maintain a higher send rate than TCP when loss occurs (as discussed in section 4.3), but we have not looked at how bad the violation of the fairness principle is and how it is affected by the different settings (like the IAT). In addition, we need to perform more tests to determine if

RDB is TCP friendly or not. Also, currently we have only looked at how the modification aid the performance of TCP New Reno. We believe that it would be interesting to see how the other protocol variations perform with RDB enabled.

Another thing that we would like to do, is to implement a more dynamic bundling scheme. Currently, RDB bundles as long as the combined packet size is less than the MSS, which might lead to unnecessarily large packets (e.g., if loss rate is low, but RTT high). Instead, it might, for example, be better to do something more similar to exponential backoff. If one packet is lost, the kernel would be allowed to bundle with one packet, and then double the number for every loss. For every ACK that is received, the number of permitted bundles would be reduced by one.

Finally, it would be interesting to see if RDB (and the other modifications) can improve the user experience for other applications than the ones that we have tested.

Appendix A

Software

In this appendix we will present the various pieces of software used in our different experiments.

A.1 Netem

To introduce loss and delay to the network, we needed some way to control the link's performance. TC is an application that enables us, through its netem module, to do this. The program is used to configure the Traffic Control in the Linux kernel, and allows us to change different aspects of the link(s) connected to the machine. An example of the use of tc and netem is:

```
tc qdisc add dev eth0 root netem delay 50ms loss 10%
```

This command tells the kernel that all packets sent on network interface eth0 should be exposed to a 50 ms delay, and that 10% of them should be dropped. Netem uses a uniform loss pattern by default, meaning that every tenth packet will be dropped in this example .

A.2 Streamzero

Streamzero is developed by us at Simula and consists of a sender and receiver. The receiver only receives data and optionally stores it in a file, while the sender generates a stream based on the command line arguments supplied. It can generate any type of stream, but we have used it to create the thin streams needed for our theoretical experiments. The sender is started like this:

streamzero_client -s 192.168.101.220 -p 9998 -d 120 -i 200 -n 100 -t rto-200-rdb.txt -eg

-s and **-p** specify the receiver's IP address and port, while **-d** is the duration of the connection. **-i** is the IAT and **-n** the size of the packet's payload (the amount of data in a packet). To get more information about the connection, the **-t** <**file**> tells streamzero to write the contents of the TCP_INFO-struct to file before each packet is sent. This struct contains different variables that describe the current state of the connection (such as the RTT and RTTVAR) and is updated for every ACK.

-e and **-g** are probably the most interesting command line options. They tell Streamzero to use a negative exponential distribution of IATs and packet sizes, respectively. Thus, the two variables tells Streamzero to generate more random and therefore realistic traffic.

A.3 Tcpdump

Tcpdump is a program that listens on a network interface and captures either all packets or the ones that match a specific filter. You can also tell it to only store a certain number of bytes of each packet. An example for using tcpdump is:

tcpdump -i eth0 -w rdb_1.pcap tcp

In this example, tcpdump listens on interface eth0 for any TCP-packets (the last parameter is the filter). By providing the **-w** <**file**> option, tcpdump writes the packet headers to the specified file. If the user wanted to store the entire packet, he or she would have to provide the **-s0** option as well.

A.4 AnalyzeRdb

AnalyzeRdb is also created by us at Simula, and is a program that analyzes the files created by **tcpdump** and prints out certain statistics. An example of the verbose output is shown in figure A.1, which was generated by the following command:

```
Src_port: 55850 Dst_port: 12000
Total packets sent: 828250
Total bytes sent (payload): 901149955
Average packet size: 1088
Number of retransmissions: 253
Number of packets with bundled segments: 816401
Estimated loss rate: 0.0305463%
Number of unique bytes: 195279922
Redundancy: 78.3299%
------------------------------------------------
Bytewise latency - Conn: 55850
Maximum latency  : 22018ms
Minimum latency  : 130ms
Average latency  : 132.068ms
------------------------------------------------
```

Figure A.1: Example output from analyzeRdb.

./analyzeRdb -s 192.168.1.2 -r 192.168.2.2 -p 12000 -f logger-kort/noalgo-1%-4%-140-0-16 -v

The **-s** and **-r** options are respectively the sender and receiver IP, and **-p** is the destination (receiver) port. We believe that the output presented in figure A.1 should be quite self explanatory, but some of the lines still deserve a mention.

In line 3 we have the total number of bytes sent, which is used to see how much more bandwidth our modifications consume compared to regular TCP. The number of bytes is used in the computation of the redundancy share (line 9), which says how much of the data was sent more than one time. The number of packets with bundled segments (line 6) is the number of packets modified by RDB, while the three last lines contain the maximum, minimum and average latencies experienced by the packets, i.e. the time it took from data was sent and until it was acknowledged.

The CDF-plots of the application layer latency (used in section 4.2) were generated using AnalyzeRdb. By providing **-g <receiver dump>** to the application, AnalyzeRdb calculates the latency and the output can be fed directly into e.g. gnuplot[1].

[1]http://www.gnuplot.info/

A.5 Tracepump

Tracepump replays a connection captured by **tcpdump**. It removes all acknowledgments and retransmissions, and then sends the "original" packets in the dump file with the same IAT. Just like **streamzero** it consists of a sender and receiver, and you start the receiver with the following command:

```
tracepump –recv
```

This tells **tracepump** to run in receiver mode and listen to connections on the default port (12000). To run the program in sender mode, you will have to write:

```
tracepump –send [Options] <receiver IP> <file>
```

The sender reads <**file**> and then tries to replay the TCP stream in real-time. The first detected connection connects to <**receiver IP**> on port 12000, and the port number is increased by one for every subsequent detected connection. All connections are established before the transmission of data starts.

tracepump allows us to get an impression of how applications would behave under different network conditions and with different TCP modifications. Since we feed the program with the same captured traffic in all tests, we can directly compare the results because we control all variables that affect the result (like the loss rate). All applications that we have tested are non-deterministic, so two sessions will never be the same. For instance, in an MMORPG, it is impossible to have the exact same gameplay experience twice. The area you are in might suddenly be flooded by other players, you might be attacked, and so on.

Appendix B

Contents of CD-ROM

Included on the provided CD-ROM are our two kernel patches (for the Linux 2.6.22.1 and 2.6.23.8 kernel), and a copy of the web page referred to in section 4.2.3. The patches and web page are stored in the **patches/** and **usertest/** folder respectively (located at the root of the CD-ROM).

To apply the patch, you first have to copy the desired patch to the folder that contains the matching kernel's source code. Then you execute the following command:

```
patch -p0 < <patch filename>
```

Appendix C

Published papers

This appendix contains the papers we have published as a result of our work with RDB. The papers are presented in the same order as they appear in this appendix, and they are:

- **Redundant Bundling in TCP to Reduce Perceived Latency for Time-Dependent Thin Streams**. Published in IEEE Communications Letters 12(4):334 – 336, 2008.
- **TCP Enhancements For Interactive Thin-Stream Applications**. To appear in NOSSDAV 2008, Braunschweig, Germany, 28-30 May 2008, 2008.

Redundant Bundling in TCP to Reduce Perceived Latency for Time-Dependent Thin Streams

Kristian Evensen, Andreas Petlund, Carsten Griwodz, Pål Halvorsen

Abstract—TCP and UDP are the dominant transport protocols today, with TCP being preferred because of the lack of fairness mechanisms in UDP. Some time-dependent applications with small bandwidth requirements, however, occationally suffer from unnecessarily high latency due to TCP retransmission mechanisms that are optimized for high-throughput streams. Examples of such thin-stream applications are Internet telephony and multiplayer games. For such interactive applications, the high delays can be devastating to the experience of the service. To address the latency issues, we explored application-transparent, sender-side modifications. We investigated whether it is possible to bundle unacknowledged data to preempt the experience of packet loss and improve the perceived latency in time-dependent systems. We implemented and tested this idea in Linux. Our results show that we can reduce the application latency by trading it against bandwidth.

Index Terms—retransmission latency, thin streams

I. Introduction

TCP is designed to carry variable traffic from sources that demand a lot of bandwidth (greedy sources), and a lot of work has been done to increase throughput without losing control of congestion. It is assumed that a sender will always try to send data as quickly as the flow- and congestion control mechanisms permit. A challenge then is to support the many applications that consume very little bandwidth (thin data streams). These thin-stream applications may also have stringent latency requirements. Many time-dependent applications use TCP, due to the need for reliability and because of firewall policy issues. Important examples of interactive, thin-stream applications are multiplayer online games, audio conferences, sensor networks, virtual reality systems, augmented reality systems and stock exchange systems. As representative examples that have millions of users world wide, we examine online games and audio conferencing. In a packet trace from Funcom's massively multiplayer online role playing game (MMORPG) Anarchy Online, the average packet payload size was 93 bytes and the average interarrival time (IAT) 580 ms. For the online first-person shooter (FPS) game Counter Strike, the average payload size was 142 bytes, and the packet IAT was about 50 ms. Standard-compliant voice-over-IP (VoIP) telephony systems using the G.7xx audio compression formats have an average payload size of 160 bytes and a packet IAT of 20 ms [1]. In a sample packet trace of Skype, we found average payloads of 110 bytes and packet IATs of 24 ms.

The authors are affiliated with both Simula Research Laboratory, Norway and Department of Informatics, University of Oslo, Norway, e-mail: {kristrev, apetlund, griff, paalh}@ifi.uio.no

These applications are also highly interactive and thus depend on the timely delivery of data. Different applications have different latency requirements. For example, the required latency is approximately 100 ms for FPS games, 500 ms for role playing games (RPGs) and 1000 ms for real-time strategy (RTS) games [2]. Latency in audio conferences must stay below 150-200 ms to achieve user satisfaction and below 400 ms to remain usable [3]. TCP variations that assume greedy sources do not accommodate thin-stream needs. It is unlikely that a separate transport protocol that addresses these needs will succeed any time soon, and it is also undesirable to make this distinction because applications may, at different times, be both greedy and interactive. It is, therefore, important to find solutions that react dynamically and use appropriate mechanisms according to the stream characteristics [4].

To address the low latency requirements of thin streams and the problems posed by the TCP retransmission mechanisms, we aim for application-transparent, sender-side modifications so that neither existing applications nor various multi-platform (OS) clients will need modifications. In particular, we propose a dynamically applicable bundling technique that includes unacknowledged data in the next packet if there is room. It is similar to several existing mechanisms. For example, TCP merges small user writes using Nagle's algorithm [5], and the stream control transmission protocol (SCTP) [6] has a multiplexing operation whereby more than one user message may be carried in the same packet. Thus, the number of sent packets is reduced.

Sending redundant data has been proposed both for latency reduction and error control. TCP and SCTP already bundle unacknowledged packets in some retransmission situations. For conferencing systems, the real-time transport protocol (RTP) packets may carry redundant audio data [7]. This allows a lost packet to be recovered earlier at the receiving side, because the data is also included in the next. However, to the best of our knowledge, no system performs such dynamic bundling for thin streams by packing unacknowledged data in a fully TCP-compatible manner before a loss is detected. We implemented a bundling mechanism and tested it in the Linux 2.6.19 kernel. The test results show that we can reduce the application latency experienced due to retransmissions by trading bandwidth for lower latency.

II. Redundant Bundling

Supporting time-dependent thin streams over TCP is a challenge, because current loss recovery and congestion control mechanisms are aimed at greedy streams that try to utilize the full bandwidth as efficiently as possible. To achieve this with TCP's set of mechanisms, it is appropriate to use buffers that are as large as possible and to delay packet retransmission for as long as possible. Large buffers counters the throttling of throughput by flow control, allowing huge application layer delays. Delayed transmission strives to avoid retransmissions of out of order packets, promoting these delays actively as a desired feature. Both methods come with an associated cost of increased delay, which means that interactive thin-stream applications experience greater delays in the application.

In our redundant data bundling (RDB) technique for TCP, we trade off space in a packet for reduced latency by reducing the number of required retransmissions. For important protocols such as gigabit Ethernet, this may be performed without exceeding the minimum slot size in our target scenarios, which means that RDB does not waste link bandwidth. It is implemented as a sender-side modification that copies (bundles) data from the previous unacknowledged packets in the send/retransmission-queue into the next packet. For example, assume that two packets of 100 bytes each should be sent. The first is sent using TCP sequence number X and a payload length of 100. Then, when the second packet is processed and the first is not yet acknowledged, the two packets are bundled, which results in a packet that has the same sequence number X but a payload length of 200. This scheme creates packets that could also appear with multiple lost acknowledgments (ACKs) and is fully TCP-compatible. An unmodified receiver should be able to handle streams with redundant data correctly.

The results of our experiments show that by sacrificing bandwidth (increasing packet size) without increasing the number of packets, time dependency can be supported better because a lost packet can be recovered when the next packet arrives. The approach has no effect at all on greedy sources that fill packets to their maximum transmission unit (MTU) size.

III. Experiments

To test our bundling scheme, we modified the 2.6.19 (and, later, 2.6.22) Linux kernel and transmitted thin data streams over 1) an emulated network, using the netem emulator in Linux to create loss and delay, and 2) over the Internet from Oslo to the Universities in Amherst, UMASS, (USA) and Hong Kong. Each test in the emulated network setting had a fixed round-trip time (RTT) (between 50 and 250 ms) and a fixed packet-loss rate (between 0 and 10 %). The packet size was 120 bytes for all experiments and we used packet IATs between 25 and 500 ms. These parameters match our analysis of packet traces from the thin-stream applications that were described briefly in section I. For the real Internet test where the RTTs and loss rates varied, we used real game traffic from World of Warcraft (WoW). For comparison, we used TCP New Reno with selective acknowledgement (SACK) because it performed the best out of the existing TCP variants in the thin-stream scenario [4]. We also turned off Nagle's algorithm [5].

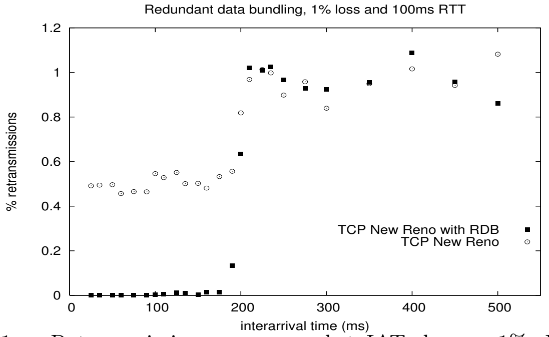


Fig. 1. Retransmissions versus packet IAT, loss = 1%, RTT = 100 ms

Fig. 1 shows the percentage of retransmissions experienced using the emulator when we vary the packet IAT using an RTT of 100 ms and a packet loss rate of 1% each way. The retransmission percentage for traditional TCP adheres to the packet loss rate and the retransmission scheme. At packet IATs below the minimum retransmission timeout (minRTO) value, the number of retransmissions depends on the number of lost packets. A lost ACK will not trigger a retransmission because the next one implicitly ACKs the previous packet before the timeout. However, when the packet IAT increases, packets are also retransmitted because of lost ACKs. This explains the jump from 0.5% to 1% for regular TCP in Fig. 1. On the other hand, Fig. 1 also shows that our bundling enhancement requires very few retransmissions, which again reduces the perceived latency. When there are (relatively) many packets that can be bundled, i.e., when IATs are low, no packets are retransmitted because a copy of the data arrives in one of the subsequent packets. However, the bundling gain lessens as the IATs increase. The reason for this is the reduced amount of data sent between the timeouts. Since less than one data segment is unacknowledged, there is no opportunity to bundle. Fig. 1 shows that the stream starts to retransmit at an IAT of roughly 200 ms (which is higher than in most of the applications we have analyzed). At this point, the performance of TCP with RDB quickly converges to that of the unmodified TCP because the minRTO is 200 ms. The results using other RTTs and loss rates show the same pattern.

Fig. 2 shows the percentage of retransmissions when the RTT increases. For these tests, we have varied the IAT randomly between 140 and 160 ms to reflect the traffic that we want to emulate. We can see that a very small number of retransmissions occurs, and as the RTT increases towards 500 ms, the tests stabilize at no retransmissions. The change from occasional retransmissions to none at all may be caused by changes in the retransmission timeout (RTO). In summary, the reduction in the number of

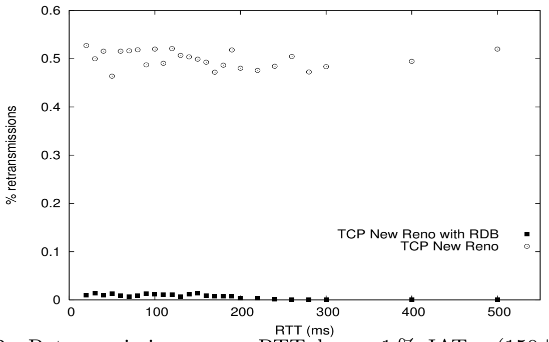


Fig. 2. Retransmissions versus RTT, loss = 1 %, IAT = (150±10) ms

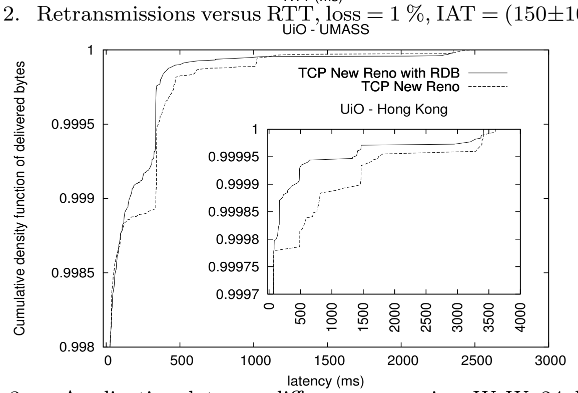


Fig. 3. Application latency differences running WoW 24 hours (19/11-2007).

retransmissions by timeout in TCP with RDB increases with the number of packets that are in transit and, thus, the ability to ACK data before a timeout occurs.

Of course, the performance gain of the bundling technique depends on the packet loss rate. This claim is confirmed in our experiments. The number of timeout retransmissions in TCP increases as the loss rate increases. In our tested scenarios, TCP with RDB hardly experiences retransmissions. However, for (unrealistically) high loss rates (such as 10%), the number of packets sent using our technique increases because the packets do not have room for more bundling. Each retransmission will then be sent by the standard TCP retransmission scheme as a separate packet. Here, standard TCP bundles the unacknowledged data in one retransmission and thereby reduces the number of packets sent. Nevertheless, in our bundling scheme, the latency is still less than in TCP New Reno.

Finally, the latency at the application level, which influences the end-user satisfaction directly, is the most important measure. Fig. 3 shows the measured application latencies using real WoW game traffic over the Internet when RDB and New Reno are used back to back to have equal conditions. In general, the data recovery latency is reduced when using TCP with RDB compared to plain New Reno (both to Amherst and Hong Kong), but we also see some few large latencies for both mechanisms which are due to larger burst losses. However, there are vital differences in the latency where we see that RDB usually delivers lost data much faster, i.e., normally in the next packet, compared to Reno's retransmitted data.

Using TCP with RDB, we make a tradeoff between bandwidth and latency. In the tests for which the results are presented in Fig. 1 and 2, we used the maximum bundling limit, such that we try to bundle as long as there is available space in a packet (up to 1448 bytes for Ethernet). To determine whether we could achieve the same gain by reducing the bandwidth in terms of transmitted bytes, we have also reduced the bundling limit to approximately half the gigabit Ethernet slot size (240 bytes bundling maximum one data element). For a scenario with uncorrelated losses, we did not see any differences. The bundling limit can in many cases be lowered while retaining a reduction in the perceived latency compared to New Reno.

IV. Dynamic behaviour

The experimental results show that we can reduce application latency for thin streams by introducing redundant bundling in TCP. RDB is efficient when packet sizes are small and IATs are less than the retransmission timeout value. In these cases, the number of packets remains the same; the overhead is in the number of transmitted bytes. Two issues that need to be addressed are when the technique should be applied and how the bundling limit should be chosen. In the current prototype, the bundling scheme is dynamically applied successfully (in terms of the results) when the data stream is thin, i.e. when the data stream has a low rate, small packets, and a low number of packets in transit. However, other factors should also be considered. For example, in a low loss rate scenario, a single loss can often be corrected by the next packet (as shown in Fig. 3). As the loss rate increases, the probability of a burst of losses increases. To compensate for this, and to increase the probability that the data will arrive before a retransmission occurs, it may be necessary to increase the bundling limit.

RDB performs best when it can preempt the retransmission timer with an ACK from a bundled packet. However, there is an intermediate stage in which spurious retransmissions due to timeouts may be experienced, but in which the latency of the application layer can still be improved. This happens when the IAT exceeds the minRTO value. When loss occurs, a timeout will be triggered before the ACK for the bundled packet is received. When the IAT exceeds minRTO + RTT, there is no gain, and the mechanism should be turned off completely. Thus, the RDB mechanism is turned on and off and the bundling limit should be set dynamically on a per connection basis according to the packet size, packet IAT, measured RTT, and estimated loss rate.

V. Conclusions and further work

Many current applications are highly interactive and depend on timely delivery of data. For applications characterized by thin streams, TCP can not guarantee delivery times, and newer variations worsen the problem because they rely on a greedy source. This being so, we implemented a redundant data bundling scheme to reduce the retransmission latency for such scenarios. Our experimental results, both in a lab setting and in real Internet tests, show that we can trade off some bandwidth to greatly lower the perceived latency at the application level in these scenarios.

References

[1] M. Hassan and D. F. Alekseevich, "Variable packet size of ip packets for voip transmission," in Proceedings of the IASTED International Conference conference on Internet and Multimedia Systems and Applications (IMSA). ACTA Press, 2006, pp. 136–141.

[2] M. Claypool and K. Claypool, "Latency and player actions in online games," Communications of the ACM, vol. 49, no. 11, pp. 40–45, Nov. 2005.

[3] International Telecommunication Union (ITU-T), "One-way Transmission Time, ITU-T Recommendation G.114," 2003.

[4] C. Griwodz and P. Halvorsen, "The fun of using TCP for an MMORPG," in Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV). ACM Press, May 2006.

[5] J. Nagle, "Congestion control in IP/TCP internetworks," RFC 896, Jan. 1984. [Online]. Available: http://www.ietf.org/rfc/rfc896.txt

[6] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson, "Stream Control Transmission Protocol," RFC 2960 (Proposed Standard), Oct. 2000, updated by RFC 3309. [Online]. Available: http://www.ietf.org/rfc/rfc2960.txt

[7] C. Perkins, I. Kouvelas, O. Hodson, V. Hardman, M. Handley, J. Bolot, A. Vega-Garcia, and S. Fosse-Parisis, "RTP Payload for Redundant Audio Data," RFC 2198 (Proposed Standard), Sept. 1997. [Online]. Available: http://www.ietf.org/rfc/rfc2198.txt

TCP Enhancements for Interactive Thin-Stream Applications

Andreas Petlund, Kristian Evensen, Carsten Griwodz, Pål Halvorsen
Simula Research Laboratory, Norway
Department of Informatics, University of Oslo, Norway
{apetlund, kristrev, griff, paalh}@simula.no

1. INTRODUCTION

TCP is frequently used for interactive multimedia applications like online games and voice-over-IP (VoIP) because it avoids firewall issues. However, traffic analysis shows that these streams usually have small packets and a low packet rate, and that in case of loss, severe latency penalties occur for all existing TCP variations in Linux [5]. In this demonstration, we show how small TCP enhancements greatly improve the perceived quality of such low latency, interactive applications.

2. INTERACTIVE THIN STREAMS

Many interactive applications have *thin stream* characteristics. In this context, a stream is considered *thin* if the application generates data in such a way that: a) The packet interarrival times are so high that the transport protocol's fast retransmission mechanisms are ineffective, and b) the size of most packets is well below the Maximum Segment Size (MSS).

Audio conferencing with real-time delivery of voice data across the network is an example of a class of applications that uses thin data streams and has a strict timeliness requirement due to its interactive nature. Nowadays, audio chat is typically included in virtual environments, and IP telephony is increasingly common. Many VoIP telephone systems use the G.7xx audio compression formats recommended by ITU-T. G.711 and G.729 have a bandwidth requirement of 64 and 8 Kbps, respectively. The packet size is determined by the packet transmission cycle (typically in the area of a few tens of ms, resulting in packet sizes of around 80 to 320 bytes for G.711). A similar example is Skype which boasts millions of registered users. We have, as an example, analyzed Skype sessions and seen that this application shares the characteristics of IP telephony. The packets are small (payload of approximately 110 bytes in average) and the bandwidth low (about 40 Kbps). To enable satisfactory interaction in such applications, ITU-T defines guidelines for the one-way transmission time. These guidelines state that users begin to get dissatisfied when the delay exceeds 150-200 ms and that the maximum delay should not exceed 400 ms [6].

Distributed online games are examples of thin-stream applications as well. We have analyzed game traces of several titles including Anarchy Online, World of Warcraft, Counter Strike, Halo 3 and Gears of War with respect to their network traffic characteristics. These games have packets sizes far below 250 bytes and rates below 20 packets per second. Occasionally, gamers experienced extreme worst-case delays. When considering user satisfaction, this class of applications requires tight timeliness, with latency thresholds at approximately 100 ms for first-person shooter games, 500 ms for role-playing games and 1000 ms for real-time strategy games [3].

With these characteristics and strict latency requirements in mind, supporting interactive thin-stream applications is challenging. The data streams in the described scenarios are poorly supported by the existing TCP variations in Linux. Their shortcomings are 1) that they rarely trigger fast retransmissions, thus making timeouts the main cause of retransmissions, and 2) that TCP-style congestion control does not apply because the stream cannot back off. Since improvements for TCP have mainly focused on traditional thick stream applications like web and ftp download, new mechanisms are needed for the interactive thin stream scenario.

3. TCP ENHANCEMENTS

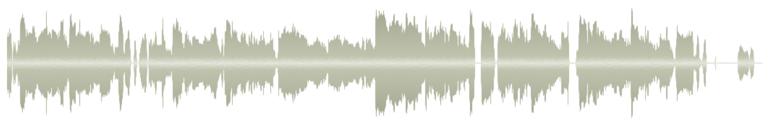
The results of our earlier investigations of TCP [4, 5, 7] show that it is important to distinguish between thick and thin streams with respect to latency. They also show that there is potential for a large performance gain by introducing new mechanisms in the thin-stream cases. In short, if the kernel detects a thin stream, we trade a small amount of bandwidth for latency reduction and apply:

Removal of exponential backoff: To prevent an exponential increase in retransmission delay for a repeatedly lost packet, we remove the exponential factor [5].

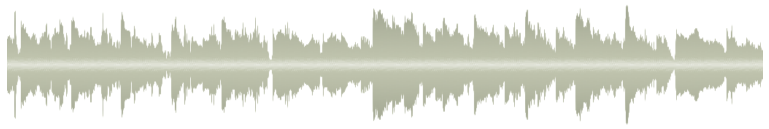
Faster Fast Retransmit: Instead of waiting for 3 duplicate acknowledgments before sending a fast retransmission, we retransmit after receiving only one [7].

Redundant Data Bundling: We copy (bundle) data from the unacknowledged packets in the send buffer into the next packet if space is available [4].

As mentioned above, these enhancements are applied only if the stream is detected as thin. This is accomplished by defin-

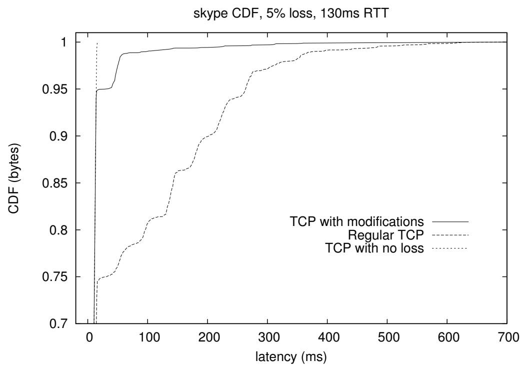


1) Unmodified TCP



2) Modified TCP

(a) A 20 second "audio spectrum" of a representative clip



(b) CDF of data arrival latency

Figure 1: Results from a Skype session.

ing thresholds for packet size and packets in flight. Also, we consider the redundancy introduced by our mechanisms acceptable because the streams are so thin that normal congestion mechanisms do not come into effect. Tests run so far indicate that fairness is indeed preserved.

4. RESULTS

We have performed several experiments with thin-stream applications like games and audio conferencing systems. All tests show improvements in user-perceived quality due to the reduced application layer latency when using our TCP enhancements. In this demo, we demonstrate the effects using the *Skype* audio conferencing system [2] and the *BZFlag* distributed game [1] as examples of interactive thin-stream applications.

We first demonstrate the performance gain of our enhancements in an audio conference. We have used Skype [2] which provides VoIP functionality, and which falls back to TCP when UDP is blocked (e.g., by ISP firewalls). The human ear is very sensitive to audio delays, and lost or delayed packets will quickly reduce the user-perceived quality. Figure 1(a) shows the received audio waves, and figure 1(b) shows the arrival latency of the received audio stream in a 2 % loss and 130 ms RTT scenario (UiO - UMASS) with and without our enhancements[1]. The figures show that the faster recovery of lost packets using our enhancements reduces the size and the number of gaps in the audio stream.

In the game demonstration, we show that, using our enhancements, we can achieve a better perceived gameplay. BZFlag [1] is a first person shooter, multi-user tank game. The game predicts movement, which has to cover a longer time period when packet loss delays delivery. In case of an erroneous prediction, tanks will jump to the correct position when the (retransmitted) position update arrives. Figure 2 shows the results from an example where the game is played for 5 minutes in a 5 percent loss and 100 ms RTT scenario with and without our enhancements. As we can see, the payload of lost packets is recovered faster when the server applies our enhancements.

Both examples will be set up for the demo session where we dynamically turn on and off our enhancements. Thus, the participants can interact with the applications and see and hear the increased quality of the service when our thin stream modifications for TCP are enabled.

[1]To be able to reproduce and compare the results, one of the audio conference participants played back different audio clips. The respective (received) audio clips can be downloaded from http://home.ifi.uio.no/apetlund/nossdavdemo

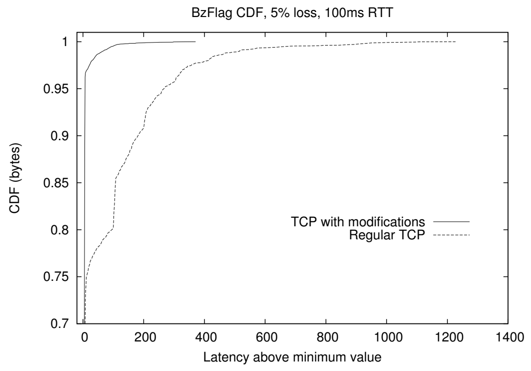


Figure 2: CDF of data arrival latency in BZFlag.

5. REFERENCES

[1] BZFlag, March 2008. http://bzflag.org.

[2] Skype, March 2008. http://www.skype.com.

[3] Claypool, M., and Claypool, K. Latency and player actions in online games. *Communications of the ACM 49*, 11 (Nov. 2005), 40–45.

[4] Evensen, K. R., Petlund, A., Griwodz, C., and Halvorsen, P. Redundant bundling in tcp to reduce perceived latency for time-dependent thin streams. *to appear in IEEE Communication Letters* (2008).

[5] Griwodz, C., and Halvorsen, P. The fun of using TCP for an MMORPG. In *International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)* (May 2006), ACM Press, pp. 1–7.

[6] International Telecommunication Union (ITU-T). One-way Transmission Time, ITU-T Recommendation G.114, 2003.

[7] Pedersen, J., Griwodz, C., and Halvorsen, P. Considerations of SCTP retransmission delays for thin streams. In *IEEE Conference on Local Computer Networks (LCN)* (Nov. 2006), pp. 1–12.

Bibliography

[1] Carsten Griwodz and Pål Halvorsen. The fun of using TCP for an MMORPG. In *International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 1–7. ACM Press, May 2006.

[2] David S. Miller. How the Linux TCP output engine works, February 2008. `http://vger.kernel.org/~davem/tcp_output.html`.

[3] Mark Claypool and Kajal Claypool. Latency and player actions in online games. *Communications of the ACM*, 49(11):40–45, November 2005.

[4] International Telecommunication Union (ITU-T). One-way Transmission Time, ITU-T Recommendation G.114, 2003.

[5] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFC 3168.

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

[7] Szabolcs Harcsik, Andreas Petlund, Carsten Griwodz, and Pål Halvorsen. Latency evaluation of networking mechanisms for game traffic. In *Workshop on Network and System Support for Games (NETGAMES)*, pages 129–134, September 2007.

[8] Espen Søgård Paaby. Evaluation of TCP retransmission delays. Master's thesis, Department of Informatics, University of Oslo, Oslo, Norway, May 2006.

[9] Mahbub Hassan and Danilkin Fiodor Alekseevich. Variable packet size of ip packets for voip transmission. In *IASTED International Conference conference on Internet and Multimedia Systems and Applications (IMSA)*, pages 136–141. ACTA Press, 2006.

[10] C.M. Karat, C. Halverson, D. Horn, and J Karat. Patterns of entry and correction in large vocabulary continuous speech recognition systems. In *CHI 99 Conference Proceedings*, pages 568–575, 1999.

[11] IEEE. *IEEE 802.3-2005 - Section One*, 2005.

[12] Andrew S. Tanenbaum. *Computer Netowrks*. Prentice Hall, 2003. ISBN 0-13-038488-7.

[13] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control . RFC 2581 (Proposed Standard), April 1999. Updated by RFC 3390.

[14] Luigi A. Grieco and Saverio Mascolo. Performance evaluation and comparison of Westwood+, New Reno, and Vegas TCP congestion control. *ACM Computer Communication Review*, 34(2):25–38, 2004.

[15] Claudio Casetti, Mario Gerla, Saverio Mascolo, M. Y. Sanadidi, and Ren Wang. TCP Westwood: end-to-end congestion control for wired/wireless networks. *Wireless Network*, 8(5):467–479, 2002.

[16] A. Dell'Aera, L.A. Grieco, and S. Mascolo. Linux 2.4 implementation of westwood+ tcp with rate-halving: a performance evaluation over the internet. In *Communications, 2004 IEEE International Conference on*, pages 2092–2096, Washington, DC, USA, 2004. IEEE Computer Society.

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

[18] G. Malkin. Internet Users' Glossary. RFC 1983 (Informational), August 1996.

[19] S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 2582 (Experimental), April 1999. Obsoleted by RFC 3782.

[20] V. Jacobson and R.T. Braden. TCP extensions for long-delay paths. RFC 1072, October 1988. Obsoleted by RFCs 1323, 2018.

[21] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgement Options. RFC 2018 (Proposed Standard), October 1996.

[22] J. Nagle. Congestion control in IP/TCP internetworks. RFC 896, January 1984.

[23] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960 (Proposed Standard), October 2000. Updated by RFC 3309.

[24] C. Perkins, I. Kouvelas, O. Hodson, V. Hardman, M. Handley, J.C. Bolot, A. Vega-Garcia, and S. Fosse-Parisis. RTP Payload for Redundant Audio Data. RFC 2198 (Proposed Standard), September 1997.

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

[26] Tom Anders Dalseng. Evaluering av datastiimplementasjoner for nedlasting og streamingapplikasjoner (in Norwegian). Master's thesis, Department of Informatics, University of Oslo, Oslo, Norway, November 2005.

[27] JEDEC Solid State Technology Association. *JESD79-2C: DDR2 SDRAM specification*, May 2006.

[28] MMOGCHART.COM. Mmog active subscriptions, 200,000+, April 2008. `http://www.mmogchart.com/Chart1.html`.