Taming Redundant Data Bundling
Balancing fairness and latency for redundant bundling in TCP

Bendik Rønning Opstad
Master’s Thesis Spring 2015
Taming Redundant Data Bundling

Balancing fairness and latency for redundant bundling in TCP

Bendik Rønning Opstad

Spring 2015
Corrections

This version contains corrections on the final version submitted for review and publication. The main changes are as follows:

- Captions added to the key plots presented in chapter 5.
- Corrections to the description of the test parameters used in each experiment in chapter 5.
- Changes to figure placement
- Merging of sentences that incorrectly formed multiple paragraphs

The full errata list can be found in Appendix E.
Abstract

The Internet is used by a vast range of different services with different requirements and needs. Along with the continuous increase in both bandwidth capacity and users, we have seen a development in the later years of more and more latency dependent applications being used. The strict latency requirements of applications such as online gaming and IP-telephony (Voice over IP (VoIP)) differ greatly from traditional greedy applications like FTP, that seeks to achieve the best possible throughput. As all the network traffic share the same network resources, the underlying network has a difficult task in trying to balance the resources and the needs of different applications.

Being the most used protocol on the Internet, Transmission Control Protocol (TCP) is used for many different kinds of services. Due to its reliability guarantees of in-order data delivery and error detection, it is the first choice for many use cases where the investment in custom solutions, based on such as User Datagram Protocol (UDP), cannot be justified.

Applications transferring time-dependent data, often send thin-stream traffic, characterized by small packet payload and high inter-transmission times (ITTs). The major cause of high latencies in thin stream are lost packets, and how the mechanisms for retransmitting the lost data work (Griwodz and Halvorsen [2006, a1]).

Thin streams suffer from the fact that TCP has been tuned for greedy traffic, where the low ITT ensures that retransmissions are initiated faster, giving satisfactory results even for latency sensitive data such video streaming (Wang et al. [2008, a2]).

In this thesis we continue the work on a sender side TCP modification called Redundant Data Bundling (RDB) for the Linux kernel, that aims at improving the latencies for thin streams, without being unfair to competing network streams.
Acknowledgements

First I would like to thank my supervisors, Dr. Andreas Petlund, Prof. Dr. Pål Halvorsen, and Prof. Dr. Carsten Griwodz, for great guidance during the work on this thesis. They have all provided a wide range of useful feedback on the background work, as well as on the writing process of the thesis. Their understanding for how much time certain tasks may require is also greatly appreciated.

A special thanks goes to Jonas Sæther Markussen, for a long and fruitful collaboration that served us both well. Also the countless discussions on various topics during our time here will be remembered.

Finally, I would like to thank my family and friends for their support.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corrections</td>
<td>i</td>
</tr>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Contents</td>
<td>vi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xiii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xv</td>
</tr>
<tr>
<td>List of Equations</td>
<td>xvii</td>
</tr>
<tr>
<td>List of Source code listings</td>
<td>xix</td>
</tr>
<tr>
<td>List of command examples</td>
<td>xxi</td>
</tr>
<tr>
<td>Glossary</td>
<td>xxiii</td>
</tr>
<tr>
<td>Acronyms</td>
<td>xxxi</td>
</tr>
</tbody>
</table>

### 1 Introduction

1.1 Background and motivation .......................... 1
1.2 Problem statement ................................... 2
1.3 Research Method ..................................... 3
1.4 Main Contributions ................................... 4
1.5 Outline ............................................... 5

### 2 Thin streams

2.1 Transport layer protocols for interactive applications .......................... 7
2.2 Interactive applications ................................ 8
   2.2.1 VoIP ............................................. 8
   2.2.2 Video streaming ................................ 8
   2.2.3 Online games .................................. 10
2.3 Characterizing different network streams .................. 11
   2.3.1 What is a thin stream? ........................ 12
   2.3.2 Identifying thin streams ...................... 13
## Contents

2.4 Overview of TCP ................................. 14
  2.4.1 Data flow .................................. 15
  2.4.2 Congestion Control ......................... 16
    2.4.2.1 Basis for the modern congestion control in TCP 16
    2.4.2.2 Congestion Window ....................... 19
    2.4.2.3 Different types of congestion control mechanisms 19
    2.4.2.4 Equation based Congestion Control .......... 20
    2.4.2.5 Binomial Congestion Control Algorithms .... 21
  2.4.3 Nagle’s algorithm .......................... 22
  2.4.4 Delayed ACKs .............................. 23
  2.4.5 RTT measurements .......................... 23
    2.4.5.1 Retransmission ambiguity problem ......... 24
    2.4.5.2 TCP Timestamps ......................... 24
  2.4.6 Retransmission timeout ..................... 26
  2.4.7 Exponential Backoff ....................... 27

2.5 Fairness ........................................ 27
  2.5.1 Measuring fairness ......................... 28
  2.5.2 Fairness metrics ........................... 28
  2.5.3 TCP-Friendliness ........................... 29
  2.5.4 Fair allocation of what among what? ......... 29

2.6 Mechanisms for improving Latency ................. 30
  2.6.1 Linear Retransmission Timeouts .. .......... 31
  2.6.2 Modified fast retransmit .................... 32
  2.6.3 Early retransmit ........................... 35
  2.6.4 Tail Loss Probe ............................ 35
  2.6.5 RTO Restart ............................... 35
  2.6.6 Redundant Data Bundling .................... 36

2.7 RDB prototype v1 .............................. 38
  2.7.1 TCP-engine in Linux ....................... 39
  2.7.2 RDB prototype 1 (RDBv1) ................... 40
  2.7.3 Issues and critique of RDB ................. 42

2.8 Summary ......................................... 43

3 Improving the latency for thin streams ............. 45
  3.1 Experiments with thin stream mechanisms ....... 45
    3.1.1 Head-of-line blocking ...................... 51
  3.2 TCP Fairness and RDB ......................... 51
    3.2.1 RDB hiding loss events .................... 51
    3.2.1.1 Active queue management ................ 52
    3.2.2 Abusing and misusing RDB ................. 52
    3.2.2.1 Tests with senders abusing and misusing RDB 53
  3.3 PIF as a thin stream indicator ................ 55
  3.4 CC: A cause of reduced latencies ............... 56
  3.5 Summary ....................................... 57

4 RDB prototype v2 ................................ 59
  4.1 Classifying thin streams ....................... 59
### Table of Contents

5.3.2 graph_r ............................................... 95
5.3.3 streamzero ........................................... 95
  5.3.3.1 Why variate the ITT? ................................. 97
  5.3.3.2 Contributions ....................................... 99
5.3.4 analyseTCP ........................................... 99
  5.3.4.1 Contributions ...................................... 100
5.3.5 tcpprobe pdb .......................................... 103
5.3.6 Modifications to netem ................................ 104
5.4 Experiments ............................................ 105
  5.4.1 Reading the plot results ............................... 105
    5.4.1.1 Goodput ........................................ 105
    5.4.1.2 Throughput ...................................... 106
    5.4.1.3 Latency .......................................... 108
  5.4.2 Experiment 1 - Latency tests with uniform loss .... 109
    5.4.2.1 Test parameters .................................. 109
    5.4.2.2 Key results ...................................... 110
    5.4.2.3 Summary .......................................... 111
  5.4.3 Experiment 2 - Latency tests with greedy cross traffic 113
    5.4.3.1 Test parameters .................................. 113
    5.4.3.2 Key results ...................................... 114
    5.4.3.3 Summary .......................................... 120
  5.4.4 Experiment 3 - Latency tests with bundling limitations 121
    5.4.4.1 Test parameters .................................. 121
    5.4.4.2 Key results ...................................... 121
    5.4.4.3 Summary .......................................... 125
  5.4.5 Experiment 4 - Fairness experiments .................. 127
    5.4.5.1 Test parameters .................................. 127
    5.4.5.2 Key results ...................................... 128
    5.4.5.3 Summary .......................................... 129
5.5 Summary .................................................. 131

6 Conclusion ................................................. 133
  6.1 Summary .................................................. 133
  6.2 Contributions ........................................... 134
  6.3 Future work ............................................. 135

Bibliography .................................................. 137
  References (a) .............................................. 137
  Online References (b) ..................................... 141
  Internet Standards and Drafts (c) .......................... 143

Appendices
## Contents

### Appendix A  Experiments results  
A.1 Latency tests with uniform loss .................................................. 147  
A.2 Latency tests with greedy cross traffic .......................................... 153  
A.3 Latency tests with greedy cross traffic bundle limitation ................. 185  
A.4 Fairness experiments ................................................................. 239  

### Appendix B  RDBv2 implementation source code  
B.1 Congestion control changes in Linux ........................................... 265  
B.2 RDBv2 bundling implementation .................................................. 267  
B.3 TFRC simulations ................................................................. 272  
B.4 TFRC Congestion Control implementation .................................... 273  

### Appendix C  Patches  
C.1 RDB prototype v1 patch ............................................................... 283  
C.2 Netem with fixed loss ............................................................... 299  

### Appendix D  Comments from Ilpo Järvinen on Linux mailing list  

### Appendix E  Errata  

---

<table>
<thead>
<tr>
<th>Appendix A  Experiments results</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 Latency tests with uniform loss</td>
<td>147</td>
</tr>
<tr>
<td>A.2 Latency tests with greedy cross traffic</td>
<td>153</td>
</tr>
<tr>
<td>A.3 Latency tests with greedy cross traffic bundle limitation</td>
<td>185</td>
</tr>
<tr>
<td>A.4 Fairness experiments</td>
<td>239</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Appendix B  RDBv2 implementation source code</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1 Congestion control changes in Linux</td>
<td>265</td>
</tr>
<tr>
<td>B.2 RDBv2 bundling implementation</td>
<td>267</td>
</tr>
<tr>
<td>B.3 TFRC simulations</td>
<td>272</td>
</tr>
<tr>
<td>B.4 TFRC Congestion Control implementation</td>
<td>273</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Appendix C  Patches</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C.1 RDB prototype v1 patch</td>
<td>283</td>
</tr>
<tr>
<td>C.2 Netem with fixed loss</td>
<td>299</td>
</tr>
</tbody>
</table>

| Appendix D  Comments from Ilpo Järvinen on Linux mailing list |  
|---------------------------------------------------------------|--|

<table>
<thead>
<tr>
<th>Appendix E  Errata</th>
<th></th>
</tr>
</thead>
</table>
List of Figures

Chapter 2

2.1 G.114 – Determination of the effects of absolute delay by the E-model 9
2.2 Statistics from analysis of Anarchy Online server side dump .... 11
2.3 Illustration of slow-start and how loss events changes the CWND .... 18
2.4 The k, l space of nonlinear controls from the binomial algorithms ... 22
2.5 Example of TCP timestamps with TCP delayed acknowledgment ... 25
2.6 Example of TCP timestamps on packet reordering .................. 26
2.7 Illustration Linear Retransmission Timeouts ....................... 32
2.8 Timelines showing when fast retransmit is triggered ............. 34
2.9 Example of an Ethernet frame for a TCP packet with 100 bytes payload 36
2.10 Examples showing how RDB bundles the data of previously sent packets onto packets with new data. ......................... 37
2.11 Timeline of an RDB stream with packet loss ..................... 38
2.12 Call graph of parts of the TCP engine in the Linux kernel ......... 39
2.13 The TCP output queue ........................................... 40
2.14 The call sequence for outgoing data in the RDBv1 prototype implementation ........................................... 41
2.15 The call sequence for incoming packets in the RDB prototype implementation ........................................... 41

Chapter 3

3.1 Plot of ACK latencies showing the effect of queuing delay ........ 47
3.2 ACK latencies for thin stream mechanisms ........................ 49
3.3 The aggregated throughput for greedy streams competing with thick TCP and RDB streams ................................. 54

Chapter 4

4.1 The DPIFL with minimum ITTs 10 ms and 20 ms, for RTTs in range 10 – 160 ...................................................... 60
4.2 Example of TCP timestamps with TCP delayed acknowledgment ... 63
4.3 Example of RDB stream with SACK option enabled. ............ 64
4.4 Example of how the DSACK is used on an RDB stream with packet reordering. ........................................... 65
4.5 Call graph for TCP output engine, where the changes made for RDBv2 are marked in green ........................................ 66
4.6 Callgraph for the code in RDBv2 that performs the redundant bundling. 67
4.7 Congestion window growth for 20 thin streams with TCP Cubic using default kernel settings ........................................ 74
4.8 Congestion window simulation for 20 thin streams using TFRC-SP .... 75
4.9 The Linux kernel config menu with a new RDB option .................. 77
4.10 Call graph showing the relations between the TCP engine, the Linux CC framework, and the RDB module .......................... 79
4.11 Callgraph for RDBv2 ........................................ 80

Chapter 5

5.1 Packet processing time in a network router ............................. 89
5.2 Testbed network setup ........................................ 92
5.3 Results showing the difference and variation in ACK latency and loss between static (100) and dynamic (100:15) ITT .................. 98
5.4 Goodput plot example ........................................ 106
5.5 Throughput plot example ........................................ 107
5.6 Latency plot example ........................................ 108
5.7 Experiment 1: Loss 2% ITT 30:3 .................................. 110
5.8 Experiment 1: Loss 2% ITT 50:5 .................................. 111
5.9 Experiment 1: Loss 10% ITT 30:3 .................................. 112
5.10 Experiment 2: 5 streams, ITT: 10:1 .................................. 115
5.11 Experiment 2: 5 streams, ITT: 10:1 .................................. 115
5.12 Experiment 2: 5 streams, ITT: 10:1 .................................. 116
5.13 Experiment 2: 5 streams, ITT: 10:1, RDB+TFRC .................. 117
5.14 Experiment 2: 5 streams, ITT: 30:3, TCP, RDB .................. 119
5.15 Experiment 2: 5 streams, ITT: 30:3, TCP, RDB .................. 119
5.16 Experiment 2: 5 streams, ITT: 30:3, TCP, RDB .................. 120
5.17 Experiment 3: 5 streams, ITT: 10:1, TCP, RDB .................. 122
5.18 Experiment 3: 5 streams, ITT: 10:1, TCP, RDB .................. 122
5.19 Experiment 3: 5 streams, ITT: 10:1, TCP, RDB .................. 123
5.20 Experiment 3: 5 streams, ITT: 10:1, RDB+TFRC ............... 124
5.21 Experiment 4: 3 streams, ITT: 5:1, RDB, RDB+TFRC .......... 124
5.22 Experiment 4: 5 streams, ITT: 30:3, TCP, RDB ............... 128
5.23 Experiment 4: 5 streams, ITT: 30:3, TCP, RDB ............... 129
5.24 Experiment 4: 10 streams, ITT: 30:3, RDB, RDB+TFRC ....... 130
5.25 Experiment 4: 10 streams, ITT: 30:3, RDB, RDB+TFRC ....... 130
List of Tables

Chapter 2
2.1 Examples of thin stream packet statistics based on analysis of packet traces. ........................................ 12

Chapter 3
3.1 Thin stream modification test setup ........................................ 46
3.2 Thin stream modifications test results .................................. 48
3.3 Greedy vs RDB misuser test setup ....................................... 55

Appendix A
A.1 Test setup for experiment 1 .............................................. 148
A.2 Test setup for experiment 2 .............................................. 154
A.3 Test setup for experiment 3 .............................................. 187
A.4 Test setup for experiment 4 .............................................. 240
List of Equations

Chapter 2
2.1 Fast Recovery ssthresh ........................................ 17
2.2 Equation based CC throughput formula .................. 21
2.3 Equation based CC formalized ............................. 21
2.4 RTO-timer calculation ........................................ 27
2.5 Jain’s Fairness Index .......................................... 28

Chapter 4
4.1 Classifying thin streams .................................. 60
4.2 TFRC send rate calculation ................................. 72
4.3 TCP-Friendly Rate Control: The Small-Packet (SP) Variant (TFRC-SP) send rate header accounting ...................... 73

Chapter 5
5.1 Cost approximation for processing a network packet .... 87
5.2 Cost approximation for processing a network packet .... 88
5.3 Queue length calculation based on BDP .................. 94
# List of Source code listings

## Chapter 2

- 2.1 `tcp_stream_is_thin` in `net/tcp.h` ........................................... 31
- 2.2 The code that tests if mFR should be used. ................................. 33
- 2.3 Excerpt from the function `tcp_data_queue` in `tcp_input.c` ....... 42

## Chapter 4

- 4.1 TFRC pseudocode for calculating the weights .......................... 70
- 4.2 TFRC pseudocode for calculating average loss interval (RFC5348) . 71
- 4.3 TFRC-SP pseudocode for calculating average loss interval (RFC4828) 71

## Chapter 5

- 5.1 Setup of rate control with `htb qdisc` ...................................... 91
- 5.2 Setup of network delay with `netem` ....................................... 92
- 5.3 Function in `streamzero` that generates pseudo-random numbers from 
a given mean and standard deviation. .................................. 97
- 5.4 Bash script used to run `tcpproberdb` .................................. 103

## Appendix B

- B.1 `tcp_is_cwnd_limited` from TCP New Reno that tests if the send rate 
is limited by the CWND .................................................... 265
- B.2 `tcp_is_cwnd_limited` in Linux kernel version 3.15 that tests if the 
send rate is limited by the CWND ........................................ 266
- B.3 `tcp_is_cwnd_limited` in Linux kernel version 3.16 that tests if the 
send rate is limited by the CWND ........................................ 267
- B.4 Excerpt from the function `tcp_write_xmit` in `tcp_output.c` ....... 267
- B.7 The new function in RDBv2 for classifying thin streams based on a 
dynamic packet in flight limit (DPIFL). .................................. 269
- B.8 A modification to `tcp_stream_is_thin` used in the experiments with 
different static packets in flight limit (SPIFL) values .................. 269
- B.11 Function that calculates the send rate according to equation 2.2 .... 272
- B.12 Function that calculates the TFRC-SP loss event rate based on pseudo 
code 4.3. ................................................................. 273
B.13 Entry points defined in struct tcp_congestion_ops for the Linux CC framework in *RDB prototype version 2 (RDBv2)* ........................................ 274
B.14 tcp_ack function in the TCP engine handling incoming ACKs .... 274
B.15 Function rdb_ack in RDBv2 ........................................ 275
B.16 Function rdb_check_rtx_queue_acked in RDBv2 ................. 276
B.17 Function rdb_tfrc_update_loss_history in RDBv2 .............. 277
B.18 Function rdb_tx_update_x in RDBv2 ................................ 278
B.19 Function tfrc_lh_calc_i_mean_sp in RDBv2 ...................... 279
B.20 Function tcp_rdbcong_avoid_tfrc located in rdb_cc.c .......... 280
B.21 Function tcp_rdbcong_avoid located in rdb_cc.c ................. 281
B.22 ssthresh implementation for TCP New Reno ...................... 281
B.23 ssthresh implementation in RDBv2 .............................. 281

**Appendix C**

C.1 Patch for *RDB prototype version 1 (RDBv1), Modified fast retransmit (mFR)* and *Linear Retransmission Timeout (LT)* for Linux kernel 2.6.23298 ................................................................. 298
C.2 Netem fixed loss patch .............................................. 302
C.3 iproute2 fixed loss patch ........................................... 307
List of command examples

Chapter 5

5.1 Running streamzero ........................................... 96
5.2 Running analyseTCP ........................................... 102
5.3 Example output from tcpproberdb ............................ 103
Glossary

**ACK latency** is the time between a data segment is first sent onto the network, until an **ACK** for the data segment is received. xiv, 4, 24, 45–47, 85, 86, 95, 98–100, 105, 108, 110–112, 114, 115, 117–120, 122, 123, 125, 135, 317

**acknowledgment** is the term used for packets that acknowledge that certain data has been received. In this thesis we refer to the TCP’s acknowledgment packets, which are regular TCP packets with the **ACK** flag in the TCP header, indicating the highest sequence number that has been received in order. xxxi, 13, 21

**additive-increase** is a phase where the send rate is increased additively. 17, 21

**application layer latency** is the latency from the time a data chunk is sent from the application layer on the sender side to the data chunk is delivered to the application layer on the receiver side. 28, 59, 85, 86, 114, 135

**application limited** streams are network streams that are limited by the application and not the network. xxviii, 11, 12, 35, 53, 55, 56, 265, 267

**congestion avoidance** is a phase after the initial growth phase where the **CWND** is increased slowly. Standard TCP follows the AIMD paradigm to control the **CWND**. xxvii, 16, 17, 56

**congestion collapse** is a condition in a packet switched network where the goodput is minimal or none-existing. 1, 16, 27

**congestion control** describes the mechanisms or algorithms used by a network protocol to control the send rate to avoid network congestion. xxxi, 1, 4, 21

TCP BIC was the default **CC** algorithm in the Linux kernel from version 2.6.8 till 2.6.19 when it was replaced by TCP Cubic. 29

TCP Cubic is the default **CC** algorithm in the Linux kernel, introduced in version 2.6.19 (7e3801). xiv, xxiii, 19, 29, 39, 56, 74, 76, 114, 128

TCP New Reno is an improvement on TCP Reno, and was the default **CC** in the Linux kernel until v2.6.8 was released in August, 2004. xix, xx, 19,
TCP Nice is a CC algorithm implemented in the Linux kernel. “TCP–Nice is an experimental congestion control mechanism that uses less than it’s fair share of bandwidth when there is congestion, much like nice does for CPU usage by processes in the Unix operating system.” (McDonald and Nelson [2006, a3]).

TCP Reno is a CC algorithm named after the BSD version “4.3BSD-Reno” where it was first implemented. xxiii, 16, 19

TCP Vegas is a CC algorithm that relies on how the RTT changes to control the CWND. 19

TCP Westwood is a CC algorithm that relies on changes in the throughput to control the CWND. 19

Congestion Window is the window specifying the amount of outstanding data a TCP sender host may have. xxxi, 17

Eifel detection algorithm describes both a flag-based, and a timestamp-based algorithm for unambiguously distinguish which data packet an incoming ACK is a response to. (RFC3522) 25, 61, 62

E-model is a standardized model for measuring the quality of speech. 8

exponential backoff is a mechanism where the RTO timer is increased exponentially, by being doubled on successive RTOs. 27, 31

fast recovery is a variation to the CC algorithms that handles the sending of new data after a fast retransmit. 17, 18, 20, 26, 29, 32

fast retransmit triggers a retransmission based on incoming dupACKs. xiii, xxiv, 13, 17, 18, 30, 32–35, 55, 112

first person shooter game is a game where the player controls an avatar in first person, moving around in a virtual world where the goal is to shoot other players. xxxi, 10

flight size is the amount of outstanding data for a stream in the network (as defined by RFC5681). For segment based accounting it is natural to express this as the number of segments/packets that have not been ACKed. 13, 35

goodput is the useful part of the throughput, i.e. the data that has not already been transmitted. 4, 28, 95, 105, 106, 116, 123, 129

greedy stream is a network stream that aims to achieve the maximum possible throughput. Such streams are network limited, with the exception of scenarios where the sender is limited by such as local I/O. xiii, xxviii, 7, 11,
Head-of-line blocking is when packets in a queue are held or blocked until the first packet in the queue is processed. In a fifo-queue, any newer packets are held until the oldest packet can be processed and removed. For protocols such as TCP, which have an in-order guarantee to the application layer, any out-of-order packets are held on the receiver side until the sequence number gap is filled. xxxii, 1

inter-arrival time xxxii, 32

Internet Protocol Security is a mechanism for securing network traffic at the IP layer. xxxii, 89

inter-transmission time iii, xxxii, 12, 13

Jain's fairness index is an equation for rating the fairness of network streams, most commonly by throughput. 28

Karn’s algorithm is used to improve the accuracy of the SRTT by excluding the RTT measurements for retransmitted packets in the calculation. 24, 25

Linear Retransmission Timeout is a modification on TCP in the Linux kernel, suggested and implemented in Petlund [2009, a4] and Evensen [2008, a5]. It was introduced in the official Linux kernel version 2.6.34 (Petlund [2010a, b1]) as tcp_thin_linear_timeouts. xx, xxviii, xxxii, 30

Linux CC framework is the pluggable module system for CC implementations in the Linux kernel. xiv, xx, 39, 56, 66, 77–79, 82, 83, 274

Linux kernel is the kernel we have implemented and tested RDBv2 on. Unless otherwise specified, we refer to the kernel version 3.16 iii, xiii, xiv, xix, xxiii–xxix, 2–5, 16, 19, 20, 23, 25, 27, 30, 31, 33, 35, 38, 39, 41, 43, 45, 53, 57–59, 65, 76, 77, 82, 88, 91, 94, 103, 104, 109, 113, 127, 133, 134, 266, 267

Massively multiplayer online role-playing game is a type of online game that combines the role-playing genre with the support for a large number of simultaneous players, also known as massively multiplayer online game. xxxii, 10

Age of Conan 10

Anarchy Online 10, 13
World of Warcraft is an MMORPG created by Blizzard Entertainment released in 2004. xxxiv, 10

**minimum RTT** is the minimum registered RTT within a certain interval. 60, 82

**Modified fast retransmit** is a modification on TCP in the Linux kernel, suggested and implemented in Petlund [2009, a4] and Evensen [2008, a5]. It was introduced in the Linux kernel version 2.6.34 (Petlund [2010b, b2]) as tcp_thin_dupack. xx, xxviii, xxxii, 30, 32

**multiplicative-decrease** is a phase where the send rate is reduced drastically. 17, 20, 21

**Nagle’s Algorithm** 22, 23, 53, 57, 86, 88, 96, 112, 316

**network limited** streams are network streams which send rate is limited by the TCP CWND. xxiv, 11, 57, 90

**offloading schemes** describes the various mechanisms used in networking for offloading certain tasks onto dedicated hardware. 88

**checksum offloading** is the offloading of a checksum calculations onto dedicated hardware such as a NIC. 88, 89

**Generic Receive Offload** is a generic implementation (not even restricted to TCP/IP) for merging incoming packets into larger segments. xxxi, 93

**Generic Segmentation Offload** is LSO implemented for protocols other than TCP. xxxii, 93

**Large Receive Offload** is a technique for reducing CPU overhead by having the NIC merge the data from multiple packets into larger segments before passing them on to the kernel. xxxii, 93

**Large Segment Offload** is a technique for offloading the segmentation of larger buffers onto the hardware of the NIC. xxxii

**TCP Offload Engine** is the offloading mechanisms implemented in NIC where parts of the processing of the TCP packets is moved off the CPU and into the integrated circuits of the NIC. xxxiv

**TCP Segmentation Offload** is LSO implemented for TCP. xxxiv, 88, 93

**One-Way Delay** The time a packet uses through a network from one host to another. xxxii

**pfifo** is a packet based queue where the first packet that comes in is the first packet to come out. 94, 105, 113, 127
**queuing delay** is the extra delay caused by the data spending time in a queue. xiii, 19–21, 23, 28, 46, 47, 57, 86, 109, 112, 114–116, 120, 135

**RDB prototype version 1** is the first RDB prototype implemented in Linux kernel version 2.6.22.1, and later ported to version 2.6.23.8 (Evensen [2008, a5]). The patch is included in code listing C.1. xx, xxxiii, 3

**RDB prototype version 2** is the second implementation of RDB which we present in this thesis. xx, xxxiii, 3

**RDBv2-CC** is the CC implemented as part of RDBv2 presented in this thesis. 59, 76, 78, 82, 83, 87, 89, 90

**RTO Restart** is a mechanism proposed for TCP and SCTP, aimed at providing fast loss recovery for connections with small amounts of outstanding data. xxxiii, 30

**RTO timer** is the timer value for the RTO clock. xxiv, 13, 17, 21, 23, 26, 27, 30–32, 35, 94

**send buffering delay** is the extra delay caused by data segments being buffered in the senders TCP output queue. 86, 87, 117, 127

**Skype** is a popular voice-over-IP service. 8

**slow-start** is an algorithm which is used to control the growth of TCP's CWND. xiii, 16–18, 31, 33, 56, 76, 82, 312

**Smoothed RTT** is an estimation of the RTT calculated by continually adjusting the estimated RTT value based on new RTTMs. This gives a stable RTT value that fluctuates in a smooth manner. xxxiii, 23

**ssthresh** The slow start threshold for the CWND. When the CWND is greater or equal to this value, congestion avoidance is used. 17, 82

**Tail loss probe** is a TCP mechanism introduced in the Linux kernel version 3.10 Dukkipati [2013, b3]. xxxiii, 30

**TCP delayed acknowledgment** is a mechanism in TCP where the ACK packets are delayed for a certain amount of time unless they can be piggy-backed on data packets. xiii, 23–26, 35, 62, 63, 72, 316

**TCP engine** is the part of the Linux kernel that handles the data transferred by TCP. xiii, xiv, xx, 3, 16, 39–41, 45, 52, 59, 61, 65, 77–79, 81–83, 90, 127, 133–135, 274

**TCP friendly** describes how a network flow should behave similarly, in terms of aggressiveness, to a TCP stream in the same network conditions. 21, 29, 59, 69, 83, 85, 87, 105, 129, 313, 314
TCP output engine is the part of the Linux kernel that handles the data to be sent with TCP. See 2.7.1. xiv, 40, 57, 66, 68, 88

TCP output queue is a linked list of SKBs that keeps track of the data that is either unsent, sent or un-ACKed. xxvii, 3, 15, 24, 25, 33, 35, 36, 38–40, 42, 52, 53, 56, 57, 61, 65, 66, 68, 81, 86–89, 130, 276, 309

TCP Receive Window is the advertised receive window of one side of a TCP connection. xxxiii, 14

TCP timestamps is an extension to TCP (RFC7323) that defines two header fields containing the sender timestamp and the echo reply timestamp which is used by the receiver side to indicate which incoming packet that resulted in the reply. xiii, 24–26, 61–63

tcp_early_retrans is a TCP option in the Linux kernel. This mechanism can be enabled through the sysctl variable net.ipv4.tcp_early_retrans. xxxi, 30

TCP_RTO_MIN is the minimum RTO value for a TCP connection. 94

tcp_thin_dupack is a TCP option introduced in Linux kernel version 2.6.34 (Petlund [2010b, b2]). This mechanism is also referred to as Modified fast retransmit. It can be enabled through the socket option TCP_THIN DUPACK or through the sysctl variable net.ipv4.tcp_thin_dupack. xxvi, 32, 59, 96

tcp_thin_linear_timeouts is a TCP option in the Linux kernel. This mechanism can be enabled through the socket option TCP_THIN_LINEAR_TIMEOUTS or through the sysctl variable net.ipv4.tcp_thin_linear_timeouts. See Linear Retransmission Timeout. xxv, 59, 96

thin stream is a network stream that sends small amounts of data relative to greedy streams, such as produced when transferring files. These types of streams are application limited and often send segments smaller than one MSS. iii, xix, xxix, 1–5, 12–14, 27–32, 36, 43, 45–47, 51, 53, 55–61, 68, 72, 73, 76, 83, 85, 87, 90, 91, 93–95, 97, 105, 110, 113, 114, 116–118, 120–123, 127, 131, 133–135, 269, 311, 312, 316, 317

throughput is the total amount of data that is transferred from the sender to the receiver. The throughput may include the packet header size in addition to the payload for each packet. iii, xxiv, xxv, 1, 4, 7, 9, 11, 19, 20, 28–30, 51, 53, 54, 69, 72, 73, 76, 78, 82, 83, 87, 90, 95, 105, 106, 116, 120, 123, 129, 134

Tools

Analysetcp is a tool for analyzing tcpdump trace files. 86, 99–101

graph_r is a collection of python scripts used for plotting the results. The
source code is available at https://bitbucket.org/bendikro/graph_r. 95

**netem** is a network emulation tool in Linux that can be used to introduce artificial delay and loss in a network. xix, 4, 46, 73, 91, 92, 104, 105, 109, 113

**ns-2** is version 2 in the network simulator series of discrete-event network simulators. 91

**ns-3** is version 3 in the network simulator series of discrete-event network simulators. 91

**R** is a programming language for statistical computing. xxix, 95

**rpy2** is a python wrapper around the programming language R. 95

**sshscheduler** is a python script for running network tests. The source code is available at https://github.com/bendikro/sshscheduler 95

**Streamzero** is a client and server program developed for the purpose of testing thin streams. 73, 95, 99

**tcpdump** is a tool for saving traffic information in a network node. xxix, 85, 86, 99, 105, 106

**tcpprobe** is a Linux kernel module written by Stephen Hemminger to gather different properties of a TCP stream at runtime. xxix, 103

**tcpproberdb** is a Linux kernel module based on tcpprobe. 103

**tcptrace** is a tool for analyzing pcap traces produced by tcpdump. 100

**wireshark** is a tool for analyzing pcap traces produced by tcpdump. 100

**Unreal Tournament 2003** is a FPS-game 10

**Warcraft III** is a game created by Blizzard Entertainment released in 2002. 10
Acronyms


AIMD Additive Increase Multiplicative Decrease xxiii, 17, 18, 29, 30, 57, 83

AQM Active queue management 52

BDP bandwidth-delay product 19, 91, 93

CAGR compound annual growth rate 8


DCCP Datagram Congestion Control Protocol 7, 8

DPIFL dynamic packet in flight limit xiii, xix, 60, 67, 68, 76, 83, 109, 113, 121, 127, 269

DSACK Duplicate Selective Acknowledgments xiii, 16, 61, 63, 65, 90, 135

dupACK Duplicate Acknowledgment xxiv, 15, 17, 18, 25, 30, 32–35, 51, 52, 55, 61–64, 100, 107

ECN Explicit Congestion Notification 28, 52, 69

ER Early retransmit 30, 35, 45, 49, 51, 57, 133, Glossary: tcp_early_retrans

FACK Forward acknowledgment 16, 33

FPS-game first person shooter game xxix, xxx, 10, 12, Glossary: first person shooter game

GRO Generic Receive Offload xxx, 93, Glossary: Generic Receive Offload
GSO  Generic Segmentation Offload xxxii, 93, Glossary: Generic Segmentation Offload

HOL blocking  Head-of-line blocking xxxii, 1, 2, 30, 33, 47, 51, 57, 105, 109–112, 131, Glossary: Head-of-line blocking

IAT  inter-arrival time xxxii, 32, Glossary: inter-arrival time

IPSec  Internet Protocol Security xxxii, 89, Glossary: Internet Protocol Security

ITT  inter-transmission time iii, xiii, xxxii, 12, 13, 46, 47, 53, 60, 61, 72, 73, 76, 83, 86, 87, 94–97, 99, 105, 106, 110, 114, 116–118, 120, 121, 123, 127, 128, 131, 137, Glossary: inter-transmission time

ITU  International Telecommunication Union 8

LFN  long fat network 19

LRO  Large Receive Offload xxxii, 93, Glossary: Large Receive Offload

LSO  Large Segment Offload xxvi, xxxii, Glossary: Large Segment Offload

LT  Linear Retransmission Timeout xx, xxviii, xxxii, 30, 31, 42, 45, 49, 51, 55, 57, 76, 99, 133, 298, Glossary: Linear Retransmission Timeout

mFR  Modified fast retransmit xix, xx, xxviii, xxxii, 30–33, 35, 42, 45, 49, 51, 55, 57, 76, 99, 133, 298, Glossary: Modified fast retransmit

MIMD  Multiplicative Increase Multiplicative Decrease 16

MMORGP  Massively multiplayer online role-playing game xxvi, xxxii, 10, 36, Glossary: Massively multiplayer online role-playing game

MSS  Maximum Segment Size xxviii, xxxiii, 14, 19, 23, 36, 52, 53, 72, 88, 128, 133, 313

MTU  Maximum Transmission Unit 12

NAT  Network Address Translation 2, 8

NIC  Network interface card xxvi, 88, 89, 91, 92

OWD  One-Way Delay xxxii, Glossary: One-Way Delay

PIF  packet in flight 13, 30–33, 35, 42, 53–55, 57, 59, 60, 73, 76, 83, 90, 103, 109, 110, 116

Acronyms

**RDBv1**  RDB prototype version 1  xiii, xx, xxxiii, 3, 5, 38, 39, 41–43, 45, 51, 57, 59, 65, 69, 76, 82, 83, 99, 298, Glossary: RDB prototype version 1


**RED**  Random early detection 52

**RTO**  retransmission timeout  xxiv, xxvii, xxviii, 13, 17, 18, 26, 27, 31, 32, 34, 35, 69, 112

**RTOR**  RTO Restart  xxxiii, 30, 35, Glossary: RTO Restart

**RTP**  Real-time Transport Protocol 7–9

**RTT**  round-trip time  xiii, xxiv–xxvii, 10, 13, 14, 19–21, 23, 24, 29, 31, 32, 46, 47, 51, 55, 59, 60, 70, 72, 73, 83, 92, 94, 105, 112, 133, 134

**RTTM**  round-trip time measurement  xxvii, 19, 23–26, 82

**RTTVAR**  Round-trip time variation 26

**RWND**  TCP Receive Window  xxxiii, 14, 19, 35, Glossary: TCP Receive Window

**SACK**  Selective Acknowledgments  xiii, 15, 16, 20, 33, 35, 61, 63, 64, 90

**SCTP**  Stream Control Transmission Protocol  xxvii, 1, 2, 35

**SG**  Scatter-Gather 88


**SMSS**  The sender sides  MSS 17

**SPIFL**  static packets in flight limit  xix, 55, 59, 67, 68, 76, 83, 109–111, 269

**SRTT**  Smoothed RTT  xxv, xxxiii, 23, 24, 31, Glossary: Smoothed RTT


**TFRC**  TCP-Friendly Rate Control  20–22, 56, 69, 70, 72, 76, 81, 82, 116, 117, 123, 128, 133, 280, 281, 313

**TFRC-SP**  TCP-Friendly Rate Control: The Small-Packet (SP) Variant  xiv, xvii, xix, 72, 73, 75, 76, 78, 82, 83, 134, 273

**TLP**  Tail loss probe  xxxiii, 30, 35, 45, 51, 57, 133, Glossary: Tail loss probe
TOE  TCP Offload Engine xxxiv, Glossary: TCP Offload Engine

TSO  TCP Segmentation Offload xxxiv, 88, 93, Glossary: TCP Segmentation Offload

UDP  User Datagram Protocol iii, 1–3, 7–9, 88, 95

VNC  Virtual Network Computing 8

VoIP  Voice over IP iii, 8, 9, 11, 14, 20, 72

WOW  World of Warcraft xxxiv, 10, Glossary: World of Warcraft
Chapter 1

Introduction

Today’s public Internet has had a tremendous growth, from the first ideas of a global network in the 1960s, to the birth of TCP/IP in the 1980s, eventually leading to an explosion in the number of users starting in the mid-1990s continuing till today (Internetworldstats.com [2014, b4]).

Transmission Control Protocol (TCP) (RFC793), the most common protocol used on the Internet today (John and Tafvelin [2007, a6]), has mechanisms to control the send rate to prevent users from overflowing the network with too much data. V. Jacobson’s work on congestion control (CC) for TCP in the late 1980s is, by many, recognized as a primary reason that enabled the Internet to grow at such a speed and size as it has (Bhatti et al. [2008, a7]; Bansal and Balakrishnan [2001, a8]).

The development of CCs and retransmission mechanisms for TCP has mainly focused on stability (fairness) and handling the transfers of the bigger and bigger amounts of data through the network, i.e., throughput (Stewart et al. [2011, a9]). This has left interactive applications that value latency over throughput in a bad spot. Many of the applications with strict latency requirements produce network traffic with thin-stream characteristics, meaning they send smaller and fewer packets compared to greedier streams. Due to the design of the mechanisms that are designed to prevent congestion collapse, such interactive applications suffer from higher latencies - we argue unnecessarily, or unfairly. In this thesis we focus on how to improve the performance for these types of applications.

1.1 Background and motivation

Alternative transport protocols to TCP and User Datagram Protocol (UDP) have emerged, that aim to replace them for certain uses. An example is Stream Control Transmission Protocol (SCTP), which should be ideal for services that find UDP to be too basic, but do not require or want all of the functionality that TCP provides. With SCTP, ordering is optional, which eliminates the issue of Head-of-line blocking (HOL blocking) in TCP, which is beneficial for many types of interactive applications. With a variety of optionally negotiable features it could have the potential of replacing both UDP and TCP for many use cases.
A major problem is that many firewalls in home gateways and middleboxes, only support UDP and TCP, and do not let protocols such as SCTP through (RFC3257). Also, while OSes such as Linux, Solaris and FreeBSD and have SCTP included, Windows and OSX do not have native implementations. Until protocols such as SCTP are better supported in major OSes, home gateways, and middleboxes, they can not be used for services whose traffic must pass through firewalls and Network Address Translations (NATs).

This gives a “Chicken or the egg” situation, where alternative protocols cannot gain wide adoption without better support by the network nodes. Meanwhile, manufacturers of operating systems and network middleboxes seem unwilling to invest resources into supporting new protocols until they are forced by the consumers.

A big challenge with deploying improvements to the Internet is the lack of any centralized control of all the nodes. As changes to nodes in the Internet must not break existing functionality, updates to protocols such as TCP must be backwards compatible. This lays heavy restrictions on what kind of changes that can be made to TCP.

Even with a clear trend showing an increase of audio and video streaming traffic, studies on the ratio of UDP/TCP could not find a clear systematic trend showing a relative increase of UDP usage at the expense of TCP (Lee, Carpenter, and Brownlee [2010, a10]). This suggests that the common belief that UDP would be the obvious choice for streaming services might not be correct.

As TCP is still widely used, and is the de facto standard for many services that could benefit from the better latency provided by other protocols, we wish to look at how to improve the latency for such services using TCP.

Thin streams using TCP suffer from high latencies caused by the in-order guarantee that TCP provides. When packets are lost, the mechanisms for retransmitting the lost data cause considerable delays. Any data transmitted after the lost segment are subjected to HOL blocking which means that multiple data segments may be delayed due to only one lost packet.

In this thesis we present the continued work on the sender side TCP modification Redundant Data Bundling (RDB) for the Linux kernel. By enabling a more aggressive (re)transmission mode for thin streams, the per-packet latencies can be considerably improved. The modifications are made to maintain compatibility with TCP, which should allow for easy deployment into existing networks.

1.2 Problem statement

The Internet is a packet-switched network providing best-effort delivery of data packets. One of its strengths lies in how easily extendable the network is, and how robust the transfer of data is when one node in the network goes down. A weakness is that it heavily relies on the users to behave in a good manner. There is no centralized governance controlling how users behave, that can reprimand users that do not follow the “rules”, partly because there really are no rules.

This is why controlling the traffic using an end-to-end CC is important. Due to the lack of any such mechanisms in UDP, trying to improve the performance for time-dependent traffic on protocols utilizing CCs is the best solution for the
1.3. Research Method

network users as a whole. This will help the users that currently use TCP for such traffic, and reduce the incentives for application developers to choose UDP over TCP.

Mechanisms have been developed to improve the situation for interactive applications, one of them being RDB, which works by piggy-backing (bundling) already sent data in packets with new (unset) data. The first RDB implementation, which we refer to as *RDB prototype version 1 (RDBv1)*, showed great potential on improving the latency for thin streams, but it left certain issues unanswered. Uncertainties remain about the fairness of the RDB mechanism towards competing traffic. The lack of any mechanisms to limit aggressiveness, as well as the potential for abuse, is not sufficiently addressed in RDBv1, and is laid out as potential future work in the conclusion (Evensen [2008, a5]).

An implementation specific issue with *RDB prototype version 2 (RDBv2)* also remains, regarding how the data contained in the socket buffers (SKBs) of the TCP output queue are modified by the mechanism. The operations required to perform the manipulations of the SKBs were deemed too intrusive in regards to data integrity.

Based on the earlier work on RDB, the goal of this thesis is to continue the study of improving the latency for thin stream traffic generated by interactive applications over TCP. With the previously mentioned issues of RDBv1 in mind, we specifically aim to:

- Develop an RDB implementation that is less intrusive than RDBv1, both in regards to data integrity of the buffers in the TCP output queue, and to the Linux kernel’s TCP engine code. Streamlining the implementation, by better organizing the code into separated logical segments, is important to simplify the work of developing and extending the functionality as well as to make future patch submissions feasible.
- Investigate how to detect packet loss that is hidden from the current TCP implementation due to the redundant data introduced by RDB.
- Evaluate mechanisms that limit redundant data bundling to situations where it is most needed. This is to balance the aggressiveness of RDB against latency gains.
- Ensure that streams utilizing RDB are TCP-fair.

1.3 Research Method

The work in this thesis follows the *design* paradigm described in *Computing As a Discipline* by the ACM Task Force (Comer et al. [1989, a11]). This entails the process of stating the requirements (1) and specification (2) for the system we intend to create, before we design and implement the system (3), followed by evaluating (4).

We have written a prototype implementation of RDB in the Linux kernel, referred to as RDBv2, and experimentally evaluated the mechanism in a lab testbed, with a focus on the problems addressed in this thesis. We then analyse and present
Chapter 1. Introduction

the results based on traffic traces and run time information from the hosts on a multitude of different test setups.

Experiments

The experiments are performed in a testbed consisting of hosts running Debian Linux. We have set up the experiments with different configurations of sender hosts to test how the RDBv2 mechanism works in different scenarios.

We have run a set of latency tests with uniform loss rate enforced by `netem` to isolate the changes to the latency results by avoiding any external influence by competing network streams. The next set of experiments are set up with competing greedy and thin stream traffic to create a more realistic network environment, as well as to see how the RDB mechanism affects the other network streams. The last set of experiments, which we call fairness experiments, are designed to test how network streams produced by RDBv2 behave towards competing streams in respect to fairness when potential “evil-doers” try to abuse the mechanism to (unfairly) gain advantages over other competing network streams.

Data analysis

We have analysed the problem area in order to identify suitable metrics by which to evaluate the mechanisms. Then we have analysed the results of the experiments using these metrics. We calculate the ACK latency\(^1\), the per-packet latency for the TCP streams, and compare the results of the different network streams. By calculating the goodput and throughput from the packet traces, we can compare the amount of data that the competing streams transfer through the network. We use this to try to identify unfair behavior of the RDB mechanism.

1.4 Main Contributions

The main contributions of this thesis can be summarized as follows:

- Implemented a TCP congestion control (CC) module for the Linux kernel, that enables a sender host to send redundant data in data packets already scheduled for transmission.
- Evaluated the thin-stream classification method in the Linux kernel and suggested improvements.
- Experimentally evaluated the RDB implementation with regards to:
  - The tradeoff between latency and aggressiveness in terms of increasing the packet size, and hence the throughput.
  - How it gives RDB streams significantly better latency without being unfair towards competing traffic.

\(^1\)ACK latency is the time between a data segment is first sent onto the network, until an ACK for the data segment is received.
• How it prevents the RDB mechanism from being abused to obtain an advantage over competing traffic.

1.5 Outline

The thesis is structured as follows: In chapter 2 we look at the background for the different interactive applications that produce thin streams, and the mechanisms for improving the latency for such streams in the Linux kernel. We also present RDBv1, the TCP modification that our work is based on.

In chapter 3 we go into detail about what causes the increased latencies for thin streams, and aspects of the RDBv1 that we will try to improve upon. In chapter 4 we present RDBv2, the re-implementation of RDB, and in chapter 5 we evaluate the modifications by running experiments and presenting the results. Chapter 6 concludes the thesis with a summary of our findings and an outline on topics for future research.
Chapter 2

Thin streams

TCP is the most widely used protocol on the internet, and is the underlying engine for most common tasks such as, web browsing (HTTP), file transfers (FTP) and email (SMTP) (John and Tafvelin [2007, a6]).

The main focus of TCP’s development has been on maximizing the throughput, i.e., move as much data as possible through the network as fast as possible. What matters on the receiving side when transferring data, such as a picture, is that the data is received correctly, and the time it takes to transfer the entire image, from the first packet is received, till the last packet is received. This is where TCP shines.

Time-dependent applications on the other hand, which usually send smaller segments of data, may depend on a minimal latency for each transferred data segment. For such use cases, TCP shines somewhat less.

2.1 Transport layer protocols for interactive applications

The conventional wisdom is to avoid TCP for interactive real-time applications. This is due to the overhead of TCP’s reliability guarantee causing too much end-to-end delays (Brosh et al. [2010, a12]), which degrades the perceived quality of service as we see in section 2.2. The right choice of transport protocol highly depends on the type of application and its latency requirements, as well as the type of traffic it generates.

UDP is often used as an alternative to TCP, which gives better control over the network traffic. Because UDP provides a best-effort service with limited guarantees (only error checking), it is necessary to manually implement the required functionality. Speed, and better control over the network traffic, can be gained by using UDP, but it is costly to implement the extra functionality, and to do thorough testing to make sure it works correctly. As UDP does not offer CC, it is not well suited for greedy stream transfers of large amounts of data, unless some CC mechanism is used on top, such as Datagram Congestion Control Protocol (DCCP).

There are alternatives available that build upon UDP, such as Real-time Transport Protocol (RTP), which is designed for delivery of data, over IP networks, with real-time characteristics like interactive audio and video RFC3550. It provides fea-
tures similar to TCP, such as timestamps, and sequence numbers to identify lost or out-of-order segments. A CC is also defined for RTP based on DCCP RFC5762.

However, any UDP solution still suffers from the problem of passing through firewalls and networks behind NAT, which is the reason why TCP often is used as a fallback when UDP can not be used (Brosh et al. [2010, a12]; Guo et al. [2006, a13]).

2.2 Interactive applications with strict latency requirements

Many interactive multimedia applications utilize computer networks, where the user experience depends greatly on the network latency. Examples are Voice over IP (VoIP) software like Skype and Ventrilo, remote desktop control like Virtual Network Computing (VNC), and online multi-player games. All of these are examples of time critical services where the quality is highly dependent on timely arrival of network packets.

According to Cisco [2014, b5] forecasts, online gaming will have a compound annual growth rate (CAGR) of 34% from 2013 till 2018. They also forecast that internet video such as YouTube (short videos), Hulu (longer videos such as tv series) and Netflix (streaming), will have a CAGR of 29%, and by 2018, grown to more than \( \frac{3}{4} \) of all internet consumer traffic. From 1993 till 2013, the international VoIP transported by VoIP carriers has had a CAGR of 13% (TeleGeography Report [2013, b6]).

2.2.1 VoIP

Based on research using the E-model, the International Telecommunication Union (ITU) recommends an upper limit of 400 ms end-to-end (one-way) delay for conventional speech ([ITU-T] [2003, b7]). Figure 2.1 shows that the quality begins to degrade at around 200 ms one-way delay. At about 300 ms one-way delay some users are getting dissatisfied and at 400 ms, many are getting dissatisfied. At 500 ms nearly all the users are dissatisfied. The ITU states that most interactive applications will not be affected negatively if the end-to-end delay is below 150 ms. However, because of the lack of standardized or agreed-upon assessment tools for non-speech applications, the effects of latency cannot be stated as clearly.

Video-conferencing, such as Skype, which is simply an extension of VoIP that includes video as well as audio, has the same latency requirements to VoIP. In addition to the requirements for audio quality, users will also evaluate the user experience based on the video quality, and how well it matches the audio, i.e., how synchronized they are.

2.2.2 Video streaming

Video streaming is a service that has grown tremendously the last years, pushed by providers such as YouTube, Netflix and HBO. Cisco forecasts that by 2018
2.2. Interactive applications

Figure 2.1: G.114 – Determination of the effects of absolute delay by the E-model ([ITU-T] [2003, b7])

over half of all internet video traffic will be content delivery network traffic (Cisco [2014, b5]). While streaming content such as movies and series does not have the same strict latency requirements as VoIP, video streaming is still very vulnerable to variations in the throughput. Video that stops playing to wait for the next video frames to arrive, is highly disruptive for the user experience. Such stalling issues are dealt with by buffering parts of the video by requesting a given length of the video in advance. Dobrian et al. [2011, a14] find that the importance of different quality metrics, like time spent buffering, buffering event rate and rendering rate (frames per second), depend on the type of content that is streamed, like short or long video and if it is streamed live.

While one may argue that on-demand video streaming does not have strict latency requirements, as the video can simply be buffered and played with a delay. If the viewers do not mind the delay, this is true, but in situations such as live soccer matches, a delay may reduce the viewing experience to a great deal. Imagine viewing the live video with a 10 seconds delay and having to listen to the neighbors’ reactions on important game events before you can see it yourself.

A common belief or expectation has been that streaming traffic would use UDP, or protocols that rely on UDP, such as RTP. In light of the increase in audio and video streaming traffic in the later years, Lee, Carpenter, and Brownlee [2010, a10] studied the ratio of \( \frac{UDP}{TCP} \), but could not find a clear systematic trend showing a relative increase of UDP usage at the expense of TCP.

Despite the shortcomings of TCP it is still widely used for video streaming, where reports suggest that already in 2006, as much as 50% of video streaming on the Internet was carried by TCP (Guo et al. [2006, a13]). Cisco [2014, b5]
Chapter 2. Thin streams

forecasts, that by 2018, 79% of the global consumer internet traffic will be IP video traffic.

2.2.3 Online games

There are a multitude of different online games where the gameplay quality depends on the network in a varying degree. When the game traffic is delayed too much, the players experience what is called lag.

M. Claypool and K. Claypool [2005, a15] categorizes online games into three categories based on how dependent they are on the latency of the transferred data:

- **First person shooter games** (**FPS-games**) like Half-Life and Quake where the player navigates a virtual world in a first person perspective.

- **Role-playing games**, like World of Warcraft (**WOW**), Age of Conan and Anarchy Online, where the player usually controls an avatar in third person view.

- **Real-time strategy games** and construction games, like Warcraft III, with an omnipresent control.

They find that for first person shooter games, 100 ms one-way delay is a maximum threshold for a good user experience. For role-playing games like WOW and Anarchy Online, the threshold is around 500 ms, and for real-time strategy games it is around 1000 ms.

The **first person shooter games** (**FPS-games**) have very strict latency requirements, as the actions the players perform, like aiming and shooting a moving target, must be completed within a strict time frame. Quax et al. [2004, a16] test the effects of latency on the gameplay quality in the FPS-game Unreal Tournament 2003., and find that from 60 ms round-trip time (**RTT**) and on, the players experience the delay as disturbing.

**Massively multiplayer online role-playing games** (**MMORGPs**) can have thousands of players interacting at the same time, requiring continuous updates of game data. The information sent from the player is everything the user does, like starting to move, which direction the player moves, when the player stops, if the player shoots and the direction of the shot. Position info for the player must be sent to the server, and position info for other players must be sent back. All this information must be sent quickly for the game play to be smooth, and only small delays or loss of this information can be noticed by the players.

Griwodz and Halvorsen [2006, a1] investigate the data traffic of the game Anarchy Online. They analysed a one-hour server-side trace from a game server located in the US, that contains 175 TCP connections. In figure 2.2 we see the results of the analysis. Subfigure 2.2.(a) shows that quite a lot of the connections experience latencies above 500 ms (quadrant A), the threshold for a good user experience. Subfigure 2.2.(b) shows the number of packets per RTT the connections have.
2.3. Characterizing different network streams

Applications performing bulk data transfer such as FTP, where the primary goal is to transfer as much data in as little time as possible, produce what is called greedy streams. By continually pushing the network to transfer the data as fast as possible, the network becomes the limiting factor, hence we call them network limited streams.

Applications that have a finite bandwidth requirements, produce network streams that are application limited, meaning the throughput is limited by how much data the application produces, and not by how much data the network is able to transfer. RFC2861 defines the term application limited period as “when the sender sends less than is allowed by the congestion or receiver windows” and network limited period as “any period when the sender is sending a full window of data.”.

One may argue that any stream that is not greedy, i.e., not trying to achieve the maximum possible throughput, is application limited, making it an inherent property of the application. With that definition, a stream may be both application limited and network limited.

With the increase in bandwidth capacity, both in the Internet and in consumer homes, more applications that were previously network limited, such as VoIP, are no longer so.
Chapter 2. Thin streams

Table 2.1: Examples of thin stream packet statistics based on analysis of packet traces. (Petlund [2009, a4])

<table>
<thead>
<tr>
<th>Application</th>
<th>Protocol</th>
<th>Payload size (bytes)</th>
<th>Packet inter-arrival time (ms)</th>
<th>Avg bandwidth requirement (pps) (bps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Casa (sensor network)</td>
<td>TCP</td>
<td>175 93 572</td>
<td>7287 307 305 299898 305 299898</td>
<td>0.137 269</td>
</tr>
<tr>
<td>Windows Remote Desktop</td>
<td>TCP</td>
<td>111 8 1417</td>
<td>318 159 1 12254 2 3892 3 145 4497</td>
<td></td>
</tr>
<tr>
<td>VNC (from client)</td>
<td>TCP</td>
<td>5 1 106</td>
<td>34 8 0 5451 0 517</td>
<td>29.412 17K</td>
</tr>
<tr>
<td>VNC (from server)</td>
<td>TCP</td>
<td>827 2 1448</td>
<td>38 0 0 3557 0 571</td>
<td>26.316 187K</td>
</tr>
<tr>
<td>Skype (2 users)</td>
<td>UDP</td>
<td>111 11 316</td>
<td>30 24 0 20015 18 44</td>
<td>33.333 37K</td>
</tr>
<tr>
<td>Skyive (2 users)</td>
<td>TCP</td>
<td>236 14 1267</td>
<td>34 40 0 1671 4 80</td>
<td>29.412 69K</td>
</tr>
<tr>
<td>SSH text session</td>
<td>TCP</td>
<td>48 16 752</td>
<td>323 159 0 76610 32 3616</td>
<td>3.096 2825</td>
</tr>
<tr>
<td>Anarchy Online</td>
<td>TCP</td>
<td>98 8 1333</td>
<td>632 449 7 17032 83 4195</td>
<td>1.582 2168</td>
</tr>
<tr>
<td>World of Warcraft</td>
<td>TCP</td>
<td>26 6 1228</td>
<td>314 133 0 14855 0 3785</td>
<td>3.815 2046</td>
</tr>
<tr>
<td>Age of Conan</td>
<td>TCP</td>
<td>80 5 1460</td>
<td>86 57 0 1375 24 386</td>
<td>11.628 12K</td>
</tr>
<tr>
<td>BZFlag</td>
<td>TCP</td>
<td>30 4 1448</td>
<td>24 0 0 530 0 151</td>
<td>41.667 31370</td>
</tr>
<tr>
<td>Halo 3-8 players</td>
<td>UDP</td>
<td>247 32 1264</td>
<td>36 33 0 1403 32 182</td>
<td>27.778 60223</td>
</tr>
<tr>
<td>Halo 3-6 players</td>
<td>UDP</td>
<td>270 32 280</td>
<td>67 66 32 716 64 69</td>
<td>14.925 35888</td>
</tr>
<tr>
<td>World in Conflict (from server)</td>
<td>TCP</td>
<td>365 4 1361</td>
<td>104 100 0 315 0 300</td>
<td>9.615 31K</td>
</tr>
<tr>
<td>World in Conflict (from client)</td>
<td>TCP</td>
<td>4 4 113</td>
<td>105 100 16 1022 44 299</td>
<td>9.524 4443</td>
</tr>
<tr>
<td>Test Drive Unlimited</td>
<td>UDP</td>
<td>80 34 104</td>
<td>40 33 0 298 0 158</td>
<td>25.000 22912</td>
</tr>
<tr>
<td>Tony Hawk’s Project 8</td>
<td>UDP</td>
<td>90 32 576</td>
<td>308 163 0 4070 53 2332</td>
<td>3.247 5812</td>
</tr>
</tbody>
</table>

2.3.1 What is a thin stream?

Application limited streams is a very broad class of streams, which leads us to further classifying the types of streams in this category. We do this by looking at what we call the thickness of a stream, which is how much data is sent and how often.

Applications with strict latency requirements, have in common that they often produce application limited traffic with thin-stream characteristics. This is traffic consisting of small packets with a relatively high inter-transmission time (ITT), as we see from the examples in table 2.1.

In table 2.1 we see payload size and packet ITT statistics for many different applications and games that send traffic with thin-stream characteristics. We see that typical characteristics of the packets is the small payload size, often between a few tens to a few hundred bytes in total which is well below the Maximum Transmission Unit (MTU) of 1500 bytes for IP datagrams over Ethernet (RFC894). We also see that the ITT of the different applications is varying to a great degree, from an average of 20-30 ms at the lowest, to many hundreds and even thousands at the highest.

Lang, Branch, and Armitage [2004, a17] analyse the traffic for the FPS-game Quake3. They find that the ITT of the traffic from the server to the clients is very regular. Independent of how many clients are connected the server sends a packet to each client every 50 ms. The IP datagram size would vary depending on the
number of clients connected. For the data traffic from clients to the server, the ITT varies in the range of 10-60 ms depending on the computer hardware (graphics card), and the map the players were playing on. The IP datagram size is in the range 50-70 bytes and does not vary depending on the computer or number of players connected.

In the experiments presented in Griwodz and Halvorsen [2006, a1] on Anarchy Online, they find that most of the retransmissions are caused by retransmission timeouts (RTOs), which suggests that most of the game traffic has few packets in flight (PIFs) since they are unable to trigger a fast retransmit.

2.3.2 Identifying thin streams

While identifying thin streams might seem like a manageable task, it is not so simple. Applications producing thin streams are very diverse, but so is the traffic they produce. Fuchs [2014, a18] study how to characterize, identify, and classify thin streams. They present the following possible characteristics and metrics for identifying thin streams:

- **PIFs or flight size**
  The PIFs is the number of outstanding packets or packets in transit, i.e., packets that are in transit to the receiving host, or whose acknowledgments (ACKs) are in flight on its way back to the sender.
  In many cases, the PIFs are similar to the flight size, where flight size is the amount of outstanding data in the network (as defined in RFC5681). When doing segment based accounting, contrary to bytes based accounting, the flight size is the number of packets that have not been ACKed.
  However, in some cases, especially in thin-stream scenarios, they are not at all the same. For a stream with an ITT of 100 ms, where the RTT is always exactly 100 ms, and the RTO timer is 350 ms, the PIFs will most of the time be 1. As long as no packets are lost, the PIFs and flight size will both be 1. However, when a packet is lost, the flight size will increase to 2 when the sender sends a new packet exactly 100 ms after the previous packet was sent. The PIFs will still be 1. 100 ms later, after a new packet is sent, the flight size will increase to 3, but the PIFs is still 1.

- **Packet inter-transmission time**
  The ITT together with the RTT is for the most part what controls the PIFs. The ITT only tells you what the application needs to send within a specific time period. An application producing 100 bytes of data, with an ITT of 5 ms on a connection with 150 ms RTT, will at most have 30 PIFs. Compared to the applications in table 2.1, this stream would not be considered thin. If however, the link has an RTT of 10 ms, the PIFs would be 2, which does not seem very high. A greedy stream on a good link will have have tens of PIFs at a minimum, but if we move that same application to a link with a very limited capacity, the PIFs could reach levels as low as streams we would consider thin.
• **Payload and packet size that often is below the Maximum Segment Size (MSS)**
  As all greedy streams will fill up each packet with the MSS of data, this is possibly one of the best indicators for thin streams. However, there are still scenarios where it does not work well. For example, an application may produce relatively small amounts of data (compared to greedy streams) in a bursty fashion, such as 3 * MSS, once every RTT. In total it would not be sending much data, and with most of the time not sending any data it might be considered thin, even when most packets are full MSSs.

• **Stream duration**
  For interactive applications such as games or VoIP, one can expect a certain duration of the network stream. Using the duration requires a preliminary goal for which types of applications you want to include in the class. HTTP transactions for example, often have a very short duration, but they do not transfer much data in total. Dukkipati, Mathis, et al. [2011, a19] find that the average HTTP response from Google’s servers was 7.5kB, which corresponds to about 5-6 TCP segments. While few would argue that latency is not important for HTTP transactions, it is special case in terms of thin-stream classification, at least in comparison to the applications in table 2.1.

### 2.4 Overview of TCP

TCP provides important services that the underlying IP, in its unreliable nature, does not. In short, these features are:

• **Connection oriented**
  The hosts set up a connection before sending data. They may also negotiate and agree on different parameters during the connection establishment.

• **Stream oriented**
  The application sends and receives a stream of data and does not need to know how the data is transferred by the underlying network layers. The data is delivered to the application layer in the same order was it was sent.

• **Reliable transmission**
  In case of packet loss or packet reordering, the receiving side waits until any gaps in the order av filled before delivering the data to the application layer. The transmitted data is verified by checksums to guarantee that the data is correct. If the sender notices loss it will retransmit the data.

• **Flow control**
  The receiver side notifies the sender of how much data it will accept by setting the *TCP Receive Window (RWND)* field in the TCP header.

• **Congestion control and avoidance**
  The senders way of controlling the send rate to avoid overflowing the network with more traffic than can be handled. This is arguably the most critical
and complex part of TCP in regards to how well the network will work when shared by multiple network streams.

### 2.4.1 Data flow

To meet the requirement for reliable transmission, the TCP header in each packet contains a field for the sequence number of the payload and the length. This is used to keep track of what parts of the data stream has been sent and what has been verified as received by the receiving end.

The header has a acknowledgment field used to tell the sender what has been successfully received. This cumulative acknowledgment tells the sender that all the data up to the sequence number in the ACK field has bee received successfully. If a packet is lost, packets will arrive out-of-order on the receiver side, which means that there is a gap in the sequence space of the data. The receiver will keep this data in the incoming buffers and immediately reply with an ACK packet where the acknowledgment field contains the sequence number of the first byte starting the gap of missing data in the sequence space. When the sender host receives an ACK where the acknowledgment field contains a sequence number that has already been ACKed by a previous ACK packet, it counts it as a Duplicate Acknowledgment (dupACK).

When a data segment that fills a hole in the sequence space arrives at the receiver side, it will immediately reply with an ACK packet where the acknowledgment field contains the sequence number of the first byte that has not been received in-order, i.e., the next expected byte.

**SACK**

*Selective Acknowledgments (SACK)* is an extension to TCP, which allows the receiver side to provide more detailed information to the data sender, about which data segments have been received (*RFC2018*). By using the optional parts of the TCP header, the data receiver can provide a set of sequence number pairs defining one or more data segments that have arrived out-of-order.

When a packet is lost, creating a gap in the sequence space of received data, the (cumulative) acknowledgment field of any packet sent in return must contain the last byte received in-order + 1. This applies even when newer data is received. *Selective Acknowledgments (SACK)* does not change this, but with by providing pairs of sequence numbers in the SACK header option, the sender will also know which segments have been received, and hence, which do not need a retransmission. While the SACK option is only advisory, and permits the data receiver to discard the data blocks that cannot be cumulatively ACKed, doing so is discouraged unless the receiver is forced to, such as when buffer space is limited. Because the SACK option is advisory, the data sender may not discard any data in the TCP output queue that was reported in a SACK-block. Only when data is ACKed by the cumulative acknowledgment field, may it be removed.
SACK is widely used on the Internet, where analysis of the traffic on Google’s Web servers, found that 96% of the connections negotiated SACK (Dukkipati, Mathis, et al. [2011, a19]).

**DSACK**

RFC2883 defines an extension to SACK that specifies Duplicate Selective Acknowledgments (DSACK)-blocks which are SACK-blocks with sequence numbers that are lower than the cumulative sequence number of the acknowledgment field. This enables the data receiver to report about duplicate packets or data segments to the sender. In the Linux kernel, the TCP option tcp_dsack is enabled by default.

**FACK**

Forward acknowledgment (FACK) aims at improving the performance of TCP Reno, when multiple packets are lost. FACK relies on SACK information to keep track of which blocks are held by the receiver. In addition, it separately accounts for the amount of outstanding retransmitted data. By considering any gaps in the reported SACK blocks as an indication of loss, FACK’s packet accounting depends on in-order delivery. Due to this, the Linux kernel disables FACK when reordering is detected.

While Mathis and Mahdavi [1996, a20] present FACK as a CC, the Linux kernel does not implements the FACK functionality as a separate CC, but as a part of the core TCP engine. Through the sysctl option net.ipv4.tcp_fack, FACK may be disabled.

### 2.4.2 Congestion Control

The CC mechanism in TCP, is the mechanism responsible for keeping the data flow in control to avoid overflowing the network, and in turn causing congestion. The principles of the CC is based on controlling the network resources, with the goal of an optimal fair allocation for the network as a whole. The first CC mechanisms for TCP were developed out of necessity, after the congestion collapse that hit the Internet in the mid 1980s (Jacobson [1988, a21]). The stability of today’s Internet is often attributed to the CC and congestion avoidance mechanisms made famous by Van Jacobson (Bhatti et al. [2008, a7]; Bansal and Balakrishnan [2001, a8]; Mcdonald and Nelson [2006, a3]), along with further improvements in modern versions (Floyd and Fall [1999, a22]).

#### 2.4.2.1 Basis for the modern congestion control in TCP

The basis for the modern CC mechanisms in TCP consists of the following (as described in RFC5681)

- **Slow-start**
  
  The first phase of a TCP stream’s development is slow-start. This phase follows the *Multiplicative Increase Multiplicative Decrease (MIMD)* algorithm
where the stream will build the Congestion Window (CWND) exponentially until it reaches the network limit. It keeps building the CWND until it experiences loss or the CWND has reached the ssthresh.

- **Congestion avoidance**
  After the initial slow-start phase, the congestion avoidance phase controls the CWND. This mechanism is used as long as the CWND is greater than ssthresh.
  This phase follows the Additive Increase Multiplicative Decrease (AIMD) algorithm described in Jacobson [1988, a21] where the CWND is grown at a slow rate (additive-increase) and reduced to half the send window size (multiplicative-decrease) if case loss occurs. When an RTO is triggered, the CWND is adjusted as shown in subfigure 2.3.(a).

- **Fast retransmit**
  In the case of packet loss, the sender must retransmit the data at some point. To avoid having to wait for the RTO timer to trigger, the fast retransmit algorithm will use Duplicate Acknowledgments (dupACKs) to detect loss. If a packet is lost, the receiver notices a gap in the sequence numbers of the received packets. For each subsequent packet received that is not in sequence on that connection, the receiver replies with a dupACK, until the missing segment is filled. With fast retransmit, a retransmission is triggered when three consecutive dupACKs are received. To avoid triggering a retransmission in case of packet duplication, or when packets arrive out-of-order (caused by different network routes), the limit is set to three dupACKs (RFC5681).

- **Fast recovery**
  The goal of this mechanism, is to allow the sender to back off in a more gentle way when sporadic loss occurs. Instead of going to slow-start, as is done after an RTO, the sender may continue to transmit at a lower rate, as long as packets are coming through the network.
  Immediately after a retransmission is initiated by fast retransmit, the ssthresh must be set to no more than what is given in equation 2.1, and the CWND set to ssthresh plus 3 * SMSS. When the first ACK that acknowledges new data is received, the CWND is set to ssthresh. Given that the CWND is set to ssthresh when new data is ACKed, congestion avoidance is used to increase the CWND further on.
  Subfigure 2.3.(b) shows an example of how fast recovery works.

\[
ssthresh = \max(\text{FlightSize}/2, 2 \times \text{SMSS})
\] (2.1)
Figure 2.3: Illustrations of Slow-start, and how an RTO, and three dupACKs are handled in the AIMD phase.
2.4. Overview of TCP

2.4.2.2 Congestion Window

The CWND is used to control how much data a host can send, by specifying the upper limit of outstanding data the sender may have. The measurement unit for the CWND can be either segments or bytes, and, while the RFC5681 specification is for the most part general in respect to the units for the CWND, some equations are defined with bytes in mind. In the Linux kernel, the CWND is calculated in segments of size MSS, which is the unit we will use in this thesis unless otherwise specified.

The CWND, together with the advertised RWND, is what controls the send rate of a connection. By continuously calculating the TCP send window (sliding window) to be the minimum value of the CWND and the RWND, both the capacity of the network, and of the receiver side host, is taken into account when adjusting the send rate.

2.4.2.3 Different types of congestion control mechanisms

TCP Reno was the first CC algorithm to support all the four mechanisms described in section 2.4.2.1 RFC2001. TCP New Reno, which improves upon TCP Reno by better handling multiple loss within the same window, was the default CC in the Linux kernel until version 2.6.8 (Released August, 2004), when it was replaced by TCP Cubic. TCP Cubic uses a different algorithm to calculate the CWND growth making it better suited for networks with large bandwidth-delay product (BDP), commonly known as long fat network (LFN) (Ha, Rhee, and Xu [2008, a23]). These traditional CC mechanisms are loss-based, meaning that they primarily rely on loss detection for estimating the amount of congestion in the network. An inherent problem with the loss based CC, is that they must cause loss to find the limitations of the network. This also causes loss for other network streams using the same resources.

Research has been done on trying to find alternative CCs for TCP, that rely on other metrics than loss, or loss alone, to measure congestion. One example is TCP Westwood, which aims for a better handling of sporadic and random loss. By avoiding the drastic reduction of the CWND that TCP New Reno does, it takes into account the estimated available bandwidth when adjusting the CWND. Tests have shown that TCP Westwood provides considerably better throughput on wireless networks (Casetti et al. [2002, a24]).

Delay based congestion control

TCP Vegas is a delay based CC, that uses the calculated queuing delay to adjust the CWND. TCP Vegas implements a more accurate RTT calculation compared to TCP Reno, which is required for the queuing delay calculations. It compares the measured throughput with the expected throughput by using the baseRTT, the smallest round-trip time measurement (RTTM) within the last RTT, and the current CWND (which is assumed to be analogous to the bytes in transit). The calculated diff is the difference between the current CWND and the desired CWND,
where the size of the $\text{diff}$ indicates the amount of queuing delay or network congestion.

**Variations on the CWND reduction**

Alternative mechanisms, to control the CWND reduction in the fast recovery phase, have been created. RFC3517 defines “A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP”, an algorithm that uses SACK information to increase performance by letting senders recover more effectively when losing multiple segments within a window. “Rate halving” is another mechanism, where the sender alternates between sending retransmissions and new data during the recovery RTT. Dukkipati, Mathis, et al. [2011, a19] proposes *Proportional Rate Reduction*, which improves upon the standard fast recovery in cases of bursty loss and losses at the end of short flows, where the application has no more data to send. *Proportional Rate Reduction* became the standard recovery mechanism in Linux kernel version 3.2.

### 2.4.2.4 Equation based Congestion Control

Historically, the development of end-to-end CC and TCP goes hand in hand, which has left other protocols in the shadows, so to speak. Nevertheless, the need of CCs for non-TCP streams has been emerging, ever since the use of latency-sensitive applications started to manifest itself as a major “player” on the Internet.

The need to define how a CC should behave in a formal way has lead to a new type of CCs. Equation based CCs is a class of CCs, where the send rate is adjusted based on a function on some property, such as loss even rate. TCP is known to constantly and aggressively probe the network to find the maximum send rate. The primary goal of the equation based CCs is to avoid this, by finding and maintaining a steady send rate, that does not fluctuate as much as TCP’s, while still reacting to network congestion.

**TCP-Friendly Rate Control**

Floyd et al. [2000, a25] presents *TCP-Friendly Rate Control (TFRC)*, which is a CC for non-TCP streams operating in a best-effort network, such as the Internet. It is designed for applications that not only value maximum throughput, and it specifically aims to reduce the variations in throughput compared to TCP.

Streaming media and VoIP are such examples, where stable throughput is very important to maintain a smooth playback of video or audio. TFRC is designed to be “reasonably fair when competing for bandwidth with TCP flows,” (RFC5348) where a reasonably fair stream is defined as “if its sending rate is generally within a factor of two of the sending rate of a TCP flow under the same conditions”. Most importantly, TFRC is designed to avoid the drastic reduction of the CWND of TCP’s multiplicative-decrease phase, when loss is detected.
To achieve this, TFRC uses a control equation (equation 2.2) presented by Padhye et al. [1998, a26], which is a TCP response function describing the steady-state sending rate of TCP.

\[
T = \frac{s}{R\sqrt{\frac{2bp}{3} + t_{RTO}(3\sqrt{\frac{3bp}{8}})p(1 + 32p^2)}}
\]  

(2.2)

\(T\) is the upper send rate (bytes/s), \(s\) the packet payload size, \(R\) the RTT, \(p\) the loss event rate, \(b\) the maximum number packets acknowledged by a single TCP acknowledgment, and \(t_{RTO}\) the current RTO timer in seconds.

Instead of “blindly” adjusting the send rate (CWND) when loss occurs, TFRC uses equation 2.2 to calculate the send rate based on the rate of lost packets (loss event rate), the RTT, and the packet size. The receiver has the responsibility of calculating the loss event rate and report this to the sender regularly in feedback packets. The loss event rate is then used by the data sender to adjust the sending rate. The reason these calculations are made on the receiver, is that the loss calculations are not sensitive to lost ACK or feedback packets.

To reduce oscillation in the queuing delay, it is recommended to reduce the send rate when the queuing delay increases. This is done by maintaining an RTT estimation of the square root of the RTT, and comparing that to the square root of the most recent RTT sample.

2.4.2.5 Binomial Congestion Control Algorithms

Bansal and Balakrishnan [2001, a8] introduce a class of congestion control algorithms they call binomial algorithms. They generalize TCP’s additive-increase and multiplicative-decrease using two properties \(k\) and \(l\), where an algorithm is TCP-compatible (TCP friendly) when it satisfies \(k + l = 1\).

Additive-increase is generalized by increasing inversely proportional to the power \(k\) of the current window, and multiplicative-decrease, by decreasing proportional to the power \(k\) of the current window. This is formalized as

\[
I: \omega_t + R \leftarrow \omega_t + \frac{\alpha}{\omega_t^k}; \alpha > 0 \\
D: \omega_t + \delta \leftarrow \omega_t - \beta \omega_t^l; 0 < \beta < 1
\]  

(2.3)

where \(I\) is the increase after receiving ACKs for a window within an RTT, \(D\) is the decrease of the window after a loss/congestion event, \(\omega_t\) is the window size at time \(t\), \(R\) is the RTT, and \(\alpha\) and \(\beta\) are constants or functions of the current window
size. Equation 2.3 generalizes the linear control algorithms:

\[ k = 0, \quad l = 1 : \text{AIMD (TCP)} \]
\[ k = -1, \quad l = 1 : \text{MIMD (multiplicative increase/multiplicative decrease)} \]
\[ k = -1, \quad l = 0 : \text{MIAD} \]
\[ k = 0, \quad l = 0 : \text{AIAD} \]

Figure 2.4 shows a) the \( k,l \) space of nonlinear controls \( b) \) where it corresponds with the four linear algorithms, and \( c) \) the \( k+l = 1 \) line which is the set of TCP–compatible controls.

The idea is that by adjusting the \( k \) and \( l \) properties, the CC can be made to better fit the needs of applications such as audio and video streaming, where a drastic reduction of the send rate on loss events is very damaging.

An important difference with the binomial CC compared to other equation based CCs such as TFRC is that it does not make use of a calculated loss rate.

### 2.4.3 Nagle’s algorithm

Nagle’s Algorithm is a mechanism proposed by John Nagle, to address The small-packet problem described by RFC896. The problem, described as early as 1960s, is caused by the sender application doing multiple send calls to the kernel with very small segments. In a scenario, such as an SSH session, depending on the implementation, typing a single character could cause the application to do a send
system call. This would result in a TCP to be sent, where the header size (of around 40 bytes) is 40 times larger than the payload, giving a overhead of 4000%.

To reduce the number of transferred packets, Nagle’s Algorithm causes the data from the application layer to be buffered in the kernel for a certain period, before being sent. As long as there are outstanding packets, Nagle’s Algorithm will buffer data until a full-sized packet (defined by the MSS) can be sent.

**The Minshall variation on Nagle’s algorithm**

The Linux kernel uses a variation of Nagle’s Algorithm defined in Minshall [1999, c17]. This modification aims at fixing a problem that can occur when a sender using Nagle’s Algorithm interact with a receiver using TCP delayed acknowledgment. If an application sends $1.5* \text{MSS}$ worth of data to the kernel, the kernel will send one full-sized packet, and delay the rest of the data. Not until the kernel either receives more data from the application layer to fill a full packet, or the first packet is ACKed, will the rest of the data be transmitted. With TCP delayed acknowledgment enabled, the data receiver will delay the ACK until more packets arrive or the delay-limit is reached, which causes grave delays.

The modification to Nagle’s Algorithm therefore proposes to buffer data only when there is an outstanding packet smaller than the MSS. In this case, all the $1.5* \text{MSS}$ worth of data from the application will be sent immediately. This is because when testing for the second segment, there are no less-than MSS-sized packets in transit.

The variation to Nagle’s Algorithm is enabled by default in the Linux kernel.

**2.4.4 Delayed ACKs**

TCP delayed acknowledgment is a mechanism designed to reduce the amount of packets in the network by sending fewer ACKs. Without TCP delayed acknowledgment, the receiver host will send an ACK-reply to the sender for each incoming data packet. By allowing the receiver to introduce a delay of up to 500 ms (RFC1122), it can reduce the amount of pure-ACKs (packets that have no payload data) by ACKing the data from multiple packets received within a certain time frame. If the receiver is also sending data packets (within the time frame of the threshold), it can avoid sending pure-ACKs by piggy-backing the ACK on the data packets.

**2.4.5 RTT measurements**

An important property for the CC algorithms is the RTT, which is used for calculating the RTO timer.

Due to variations in RTTMs, caused by such as queuing delay, TCP delayed acknowledgment, or packets being routed on different paths (causing reordering), the RTT (the most current RTTM) is not used directly. To avoid rapid fluctuations, the sender maintains an estimation of the RTT, called the Smoothed RTT (SRTT) (RFC6298), computed using the algorithm described by Jacobson [1988, a21].
The smoothing works by weighting the last RTTM to $\frac{1}{8}$ on each recalculation: 
\[
SRTT = SRTT \times \frac{7}{8} + RTTM \times \frac{1}{8}.
\]

### 2.4.5.1 Retransmission ambiguity problem

To produce as accurate measurements of the RTT as possible, it is important to rule out ambiguous RTTMs, caused by events like packet loss or reordering.

Without TCP delayed acknowledgment, the sender can expect one ACK per packet sent. Anomalies, such as an ACK that covers two SKBs in the TCP output queue, can be an indication of packet loss. The exception would be if reordering has occurred. In such circumstances, the RTTM for the lost (oldest) packet may be ignored, such that only the measurement for the last sent segment is used.

However, with TCP delayed acknowledgment enabled, both the ACK latency, and the number of ACK packets, may vary in an unpredictable manner. The data sender may deduce that TCP delayed acknowledgment is enabled on the receiver, based on fluctuations in the RTTM, but even then, it cannot know if a certain ACK was delayed or not.

In the case of a retransmitted data segment, it is difficult to know if the received ACK for that (retransmitted) segment is a reply on the original packet, that was not lost, but delayed due to TCP delayed acknowledgment, or it being routed on a different (and slower) network path (RFC3522). If the received ACK is in fact a reply on the original packet, and the sender uses the send-time of when the data segment was retransmitted, the calculated RTT would be smaller than the true RTT. This is what is known as the *Retransmission ambiguity problem* (Karn and Partridge [1988, a27]).

To improve the accuracy of the SRTT, Karn’s algorithm suggests to ignore the RTTMs for retransmitted segments altogether in the calculation of the SRTT (Karn and Partridge [1988, a27]). For the reasons explained, RFC6298 states that Karn’s algorithm must be used when taking RTT samples.

### 2.4.5.2 TCP Timestamps

The TCP timestamps extension to TCP (RFC7323), can be used to improve the accuracy of the RTTM. By using two new optional header fields, *sender timestamp* and *echo-reply timestamp*, it is possible for the sender host to identify which sent packet resulted in a specific ACK packet. On each outgoing packet, the sender host sets the *sender timestamp* header field (TSva1). When the receiver side replies with an ACK packet, it will set the *echo-reply timestamp* field (TSecr) with the value from the incoming packet’s *sender timestamp* field.

RFC7323 specifies that, when using TCP delayed acknowledgment, the ACK’s TSecr field should reflect the oldest segment that is cumulatively ACKed. As illustrated in figure 2.5, we see that the value in the *echo-reply* field (TSecr = 1) of the ACK packet, reflects the timestamp (TSva1) of segment A, even when segment C is cumulatively ACKed. This is to ensure that the sender measures the effective RTT, which includes the extra delay caused by TCP delayed acknowledgment. TSRecent is a local variable on the receiver side, that always reflects the timestamp value to be used in the TSecr field of packets sent in return.
2.4. Overview of TCP

Based on billions of sampled connections from Google’s Web servers, Dukkipati, Mathis, et al. [2011, a19] found that 12% negotiated TCP timestamps. While TCP timestamps is enabled by default in the Linux kernel, it is disabled by default in MS Windows, which explains the low amount of timestamp-enabled connections in their results.

**Eifel-detection-algorithm**

Instead of solving the retransmission ambiguity problem using Karn’s algorithm, by discarding any RTTMs of retransmitted segments, TCP timestamps can be used.

The timestamp-based Eifel detection algorithm (RFC3522), enables a sender to unambiguously distinguish which packet an ACK is a response to. This is done by using the *sender timestamp* values of the packets (SKBs) in the TCP output queue, and the *echo-reply timestamp* of the incoming ACKs.

Figure 2.6 shows how the TCP timestamps are set when a packet is reordered. When $C$ arrives before $B$, a dupACK is sent in reply. When $B$ arrives, filling the hole between $A$ and $C$, the receiver can cumulatively acknowledge up to $C$, causing the ACK to cover two packets. Without TCP timestamps, that would be an ambiguous situation if a retransmission for packet $B$ was sent before the ACK arrived, as explained in section 2.4.5.1. However, with TCP timestamps enabled, due to the requirement that the TCP timestamps must reflect the earliest packet (with lowest sequence number) that is ACKed, the result is that the $TSecr$ value of the ACK that cumulatively acknowledges $C$, reflects packet $B$’s timestamp. Therefore, the sender knows that packet $B$ was not lost, but a reordering occurred.

Had $B$ been lost, the ACK on $C$ would be sent after the retransmission of $B$ arrived. In that case, the value of the $TSecr$ field would reflect the timestamp of the retransmitted $B$ segment, and not the original packet.

---

**Figure 2.5: Example of TCP timestamps with TCP delayed acknowledgment** (From RFC7323)

<table>
<thead>
<tr>
<th>TS.Recent</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, TSval = 1</td>
<td>1</td>
</tr>
<tr>
<td>B, TSval = 2</td>
<td>1</td>
</tr>
<tr>
<td>C, TSval = 3</td>
<td>1</td>
</tr>
<tr>
<td>ACK(C), TSecr = 1</td>
<td>1</td>
</tr>
</tbody>
</table>
Figure 2.6: Example of TCP timestamps on packet reordering (Customization of the example from RFC7323)

2.4.6 Retransmission timeout

In the most basic form, TCP detects data loss by the lack of ACKs. If no ACKs arrive within a given threshold, an RTO is triggered, which indicates to TCP that the packet was lost and needs a retransmission. The greater the RTO timer value, the longer it takes for a retransmission to be initiated, and the data to arrive correctly at the receiver.

The RTO timer value is calculated so that it should only trigger when necessary. If fast recovery fails, i.e., the retransmitted data is not ACKed before the RTO is triggered, the RTO timer must be large enough to give the retransmitted packet time to be ACKed. This indicates a value greater than \( RTT \times 2 \). The calculation must also take into account that the receiver end may have TCP delayed acknowledgment enabled which allows it to delay sending the ACK for up to 500 ms (RFC1122).

RFC2988 specifies the following calculation for the RTO timer:

\[
\text{RTO} = G + 4 \times \text{RTTVar}
\]

Where \( G \) is the interrupt timer granularity in seconds, \( K \) is 4, and Round-trip time variation (RTTVAR) is the variations of the RTTMs. RFC2988 specifies that, if the computed RTO timer is less than one second, it should be set to one second, and refer to research that suggests that a large minimum RTO timer is necessary to
2.5. Fairness

\[
SRTT = R \\
RTTVAR = R/2 \\
RTO = SRTT + \max(G, K \times RTTVAR) \\
RTO = \text{MIN}(1000, RTO)
\] (2.4)

avoid spurious retransmissions.

However, many TCP implementations deviate from the standard, such as the Linux kernel, where they have chosen a minimum limit of 200 ms (Sarolahti and Kuznetsov [2002, a28]), instead of the 1000 ms specified in RFC2988. Other OSes have the minimum RTO timer value even lower, like FreeBSD, where it is set to 30 ms (FreeBSD 9 source code [2014, b8]).

2.4.7 Exponential Backoff

As part of the RTO timer calculation, RFC1122 specifies that the implementation must include exponential backoff, which means that for successive RTOs on the same segment, the RTO timer value must be exponentially increased. This makes the sender wait even longer before retransmitting packets that have not been ACKed, which is meant to have a positive effect on a congested network. However, performance analysis on the exponential backoff algorithm has shown contradictory results (Kwak, Song, and Miller [2005, a29]). Mondal and Kazmanovic [2008, a30] even argue to remove the exponential backoff algorithm from TCP altogether, claiming that it can be done without causing instability issues in the Internet.

A time-dependent thin stream will contribute less to congestion than a greedy stream, due to the relatively few packets it sends. Therefore, the result of the exponential backoff can be said to cause a disproportionate decrease in the performance (increased latency) for thin streams, relative to the amount of traffic and congestion produced compared to a greedy stream.

2.5 Fairness

While the primary goal of the CC algorithms is to reduce congestion, and ultimately prevent congestion collapse, the algorithms should strive for fairness. In short, the available network resources should be fairly distributed among the users. Despite of how simple it sounds, the topic of network fairness is a complex one.

Any evaluation of network protocols, being changes to existing protocols, or proposals of new protocols, must consider the key issue of fairness. One must consider how fair it is against other streams, using the same, or similar protocols, or a mix of different protocol types.
2.5.1 Measuring fairness

A popular way of measuring fairness in networks is Jain’s fairness index (equation 2.5) (Bhatti et al. [2008, a7]). While the index is generic and may be used with any kind of resource, a common metric is throughput. The fairness index is calculated by using the end-to-end throughput for all streams that share a path, or parts of a path, through the network. The fairness index ranges from 0 to 1, where the index is 1 when all streams are allocated an equal share.

\[
Fairness = \frac{\left(\sum_{j=1}^{n} S_j\right)^2}{\left(n \sum_{j=1}^{n} S_j^2\right)}
\]  

(2.5)

The throughput-based fairness-index applies only to greedy streams of infinite length. What is not considered is when network streams have different requirements, and different lengths. If a stream is satisfied with less than what the network could maximally provide, the fairness index states that the resource allocation is unfair (Fuchs [2014, a18]).

If the fairness measure index can not consider the different requirements for different streams, how can it tell what is fair? It is a major issue when we consider greedy streams competing with thin streams, where widespread CC algorithms measure fairness by throughput, and primarily try to optimize the throughput for a maximum link-utilization.

2.5.2 Fairness metrics

There is no agreement in the network community on the goals for measuring fairness (RFC5166), understandably, as it all depends on the use case and applications. There are a multitude of fairness metrics that apply to different applications (use-cases) to a varying degree. Some relevant metrics for thin streams are:

- **Throughput** is the most common fairness metric, which can be measured on the network routers as link-utilization and as per-connection transfer times (e.g. the total time used to transfer a file). The throughput may also be distinguished from the goodput as the useful data coming through the network.

- **Delay** can be measured on the routers in the network as queuing delay, or per flow as packet delay, or per data segment which includes retransmissions. It could also be measured as the application layer latency, the time from the data was sent to the kernel on the sender, to when it enters the (user-space) application on the receiver side. While thin streams care about the per-packet latency, greedy streams only care about the per-packet latency in regards to the effect it has on the per-connection transfer time.

- **Packet loss rates** can be measured either for the network as a whole or per stream or flow. This can include congestion events such as Explicit Congestion Notification (ECN). Greedy streams care about packet loss only
2.5. Fairness

in regards to how it effects the per-connection transfer time, while time-
dependent thin streams are more likely to care about every lost packet due to
the increased latency for the data-segment in the lost packet.

- Robustness to misbehaving users considers how well the CC algorithm han-
dles misbehaving users that try to exploit or bypass the algorithm to gain an
advantage over competing streams.

2.5.3 TCP-Friendliness

TCP-friendliness is how well a network stream plays with a TCP stream. By defi-
nition, TCP is TCP friendly, but the different TCP-variants are not equally friendly,
so apparently there are degrees to the friendliness.

With more and more protocols appearing in competition with TCP, TCP-
friendliness has arisen in the network community as a goal for behavior of network
streams. Considering the success of TCP on the Internet, it is understandable that
the network community is reluctant to change the rules.

With the continuous development of TCP until today, where different prop-
ties and characteristics of the TCP streams changes, evaluating TCP-friendliness
is not a precise science. Floyd and Fall [1999, a22] define it as: “a flow is TCP-
friendly if its arrival rate does not exceed the arrival of a conformant TCP connec-
tion in the same circumstances.”

The reason most such definitions include a mention about similar conditions
or environment, is that TCP, or at least its earlier versions, are not completely fair
against competing TCP streams with differing RTTs. This has been a well known
problem with the CCs like TCP New Reno and those before, where Floyd and Fall
[1999, a22] show, by simulation, that the throughput of a stream using these CCs
varied inversely proportional to the connections RTT. When two TCP streams with
differing RTTs compete over the same bottleneck, the high-RTT stream will lose
due to this.

This is what TCP BIC, that TCP Cubic builds upon, addressed. By letting
the growth rate depend on the time between two consecutive congestion events,
declared as when it enters fast recovery, TCP BIC’s growth function is indepen-
dent of the RTT. Therefore, the two streams in the previous example will develop
approximately the same window size, even when they have different RTTs.

A TCP friendly stream must react similarly to TCP when loss occurs within a
window, by reducing its CWND to at least half, and increase the CWND by a con-
stant rate of one packet per RTT at most. This AIMD response gives a maximum
throughput described by equation 2.2, which must not be exceeded.

2.5.4 Fair allocation of what among what?

An issue with the fairness maintained by the AIMD algorithm, known as flow-rate
fairness, is that it treats all streams in a network as separate entities.

Consider a scenario with a network with five hosts sharing a bottleneck gate-
way, where each host has one TCP stream (that requests the same amount of band-
width) where the sum of requested bandwidth for all streams surpasses the bottle-
neck rate. According to flow-rate fairness the hosts would be getting a fair share of the network resources, when over time, the rate of the streams converges against an equilibrium (Jacobson [1988, a21]).

However, if one of the hosts uses four TCP streams instead of one, that one host will get half of the network resources instead of $\frac{1}{4}$. Simply by using multiple connections, a host can gain an unfair advantage over the other hosts, some would say, but the flow-rate fairness is still maintained in this scenario.

This is described in-depth by Briscoe [2007, a31], where the entire idea of flow-rate fairness is “debunked” - flow-rate fairness does even answer the right questions, allocate the right thing or allocate between the right entities, they say.

### 2.6 Mechanisms for improving Latency

Historically the development on TCP’s CC has focused mainly on improving throughput performance on network links that have become faster and faster by the years (Stewart et al. [2011, a9]).

The main reasons why thin streams are suffering when used with TCP can be attributed to a following:

- **The few PIFs** cause the thin streams to not trigger dupACKs as fast as greedy streams. This means that it takes longer for them to initiate retransmits for lost packets.
- **Due to TCP’s in order guarantee**, one lost packet will cause a HOL blocking which results in severe increase in the application layer delay.
- **The exponential increase of the RTO timer** will in cases where retransmissions are lost, due to severe congestion, or simply bad luck, increase the latencies for thin streams.
- **AIMD’s the harsh reaction to loss** by halving the CWND after a fast retransmit hits thin streams hard.

In recent years, some work has focused primarily on improving the latency for TCP streams. We will look closer at some of these mechanisms:

- **Linear Retransmission Timeout (LT)**
- **Modified fast retransmit (mFR)**
- **Early retransmit (ER)**
- **Tail loss probe (TLP)**
- **RTO Restart (RTOR)**
- **Redundant Data Bundling (RDB)**

Petlund [2009, a4] investigate how to improve the latency for time-dependent thin streams using TCP. They implement three modifications to the Linux kernel, that in various ways attempt to improve the latency; LT, mFR, and RDB.
To limit how aggressive the mechanisms are, and ensure they only affect thin streams, LT and mFR are only used if the stream is considered \textit{thin}, as defined by the function \texttt{tcp\_stream\_is\_thin} (see code listing 2.1). This function verifies that the connection has less than four PIFs, and is not in the initial slow-start phase.

<table>
<thead>
<tr>
<th>Linux kernel source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 /* Determines whether this is a thin stream (which may suffer from increased latency). Used to trigger latency-reducing mechanisms. */</td>
</tr>
<tr>
<td>2 static inline bool tcp_stream_is_thin(struct tcp_sock *tp) {</td>
</tr>
<tr>
<td>3 return tp-&gt;packets_out &lt; 4 &amp;&amp; !tcp_in_initial_slowstart(tp);</td>
</tr>
<tr>
<td>4 }</td>
</tr>
</tbody>
</table>

The function in the Linux kernel that determines if a stream is thin

Code Listing 2.1: tcp\_stream\_is\_thin in net/tcp.h

### 2.6.1 Linear Retransmission Timeouts

The exponential backoff mechanism will double the RTO timer each time an RTO is triggered. First when an ACK is returned for a segment that was not retransmitted, will the RTO timer be recalculated. This is because it can be unclear which data packet was ACKed - the initial data packet or one of the retransmissions. As the RTO timer calculation depends on the RTT, the calculation is delayed until the reception of an ACK which gives a good RTT estimation.

To improve the situation for thin streams, the LT option disables exponential backoff for up to 6 timeouts (RTOs) as long as the stream is thin. On the seventh RTO exponential backoff is resumed. When LT is triggered, instead of being doubled, the RTO timer is set to the SRTT + rttvar, which is the smoothed maximum value of the medium deviation (of the RTT) within the last RTT period.

Figure 2.7 illustrates the difference between LT and exponential backoff. As this mechanism will have effect only when retransmissions are lost, it will have the most effect in heavily congested scenarios where subsequent RTOs occur. However, it will also have effect when subsequent RTOs are lost due to bad luck.

The LT modification was included in Linux kernel version 2.6.34 (Petlund [2010a, b1]).
Chapter 2. Thin streams

Figure 2.7: Illustration of the Linear Retransmission Timeouts mechanism. (Petlund [2012, a32])

2.6.2 Modified fast retransmit (mFR)

By default fast recovery is triggered when three dupACKs have been received. In the case of thin streams, the high inter-arrival time (IAT) makes it more likely that an out-of-order packet is caused by packet loss. With three dupACKs required to trigger fast recovery, it requires the sender to send three packets after a packet is lost to generate the three dupACKs.

Subfigure 2.8.(a) illustrates how the loss of one packet affects a stream when fast retransmit is triggered after three dupACKs. When packet P1 is lost, the receiver will retransmit the packet when receiving the third dupACK triggered when P4 is received. The consequence is that packets P2-P6 must be delayed on the receiver side before being delivered to the application layer.

With thin streams, that often have very few PIFs, the fast retransmit may never be triggered. This is also an issue for short-flows, such as HTTP transactions, that only send few packets in total. Tests on Google’s Web servers showed that fast recovery accounted for 25% of the retransmission on short-flows, and 50% on bulk video traffic (Dukkipati, Mathis, et al. [2011, a19]), which means that a surprisingly large amount of the retransmissions are triggered by RTOs. Balakrishnan et al. [1998, a33] also find similar results for HTTP traffic, where more than 50% of the retransmissions by a busy web server are triggered by RTOs.

The mFR mechanism was developed to improve the latency for thin streams in such cases. With the tcp_thin_duapack option enabled, the required number of dupACK to trigger a fast retransmit, is reduced from three to one for thin streams.

Subfigure 2.8.(b) illustrates what can happen when a stream sends only one packet per RTT. It does not receive the third dupACK before the RTO timer is
2.6. Mechanisms for improving Latency

triggered, causing grave delays. In subfigure 2.8.(c) we see how this situation is improved using the mFR option. By triggering fast retransmit after the first dupACK, fewer packets are delayed due to HOL blocking.

However, in addition to the PIF limit and the test for initial slow-start phase, there are other criteria that restricts when mFR may trigger a fast retransmit. In code listing 2.2 we see the code that tests if mFR may trigger a fast retransmit. In addition to calling tcp_stream_is_thin, and testing for SACK support, it also requires no unsent data being available in the TCP output queue, as well as the return value of tcp_dupack_heuristics being more than 1. With FACK enabled by default in the kernel, this return value is tp->packets_out, the number of outstanding segments according to the FACK accounting.

Linux kernel source

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>`if ((tp-&gt;thinDupack</td>
</tr>
<tr>
<td>2</td>
<td><code>    return true;</code></td>
</tr>
<tr>
<td>3</td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

The code in the Linux kernel that tests if mFR can be used for a stream.

Code Listing 2.2: The code that tests if mFR should be used.

The mFR mechanism was included in Linux kernel version 2.6.34 (Petlund [2010b, b2]).
Figure 2.8: Timelines showing when fast retransmit is triggered

(a) Fast retransmit triggered after three dupACKs

(b) Fast retransmit is first triggered after an RTO is triggered

(c) Fast retransmit triggered after one dupACK
2.6.3 Early retransmit

ER is a mechanism for TCP and SCTP (RFC5827), that is similar to mFR in that it tries to improve the latency for streams that do not receive enough dupACKs to trigger a fast retransmit. Like mFR, it allows the sender to lower the threshold for the number of required dupACKs to trigger a fast retransmit. Where mFR allows a fast retransmit on the first dupACK as long as there are less than four PIFs, the segment based ER implemented in the Linux kernel allows a fast retransmit only when a) the flight size is less than four, and b) there is either no unsent data in the TCP output queue, or the advertised RWND does not allow new segments.

ER was included in the Linux kernel version 3.5 (Cheng [2012, b9]) and is enabled by default (Linux kernel IPv4 variables [2014, b10]). If the mFR mechanism is enabled, ER is automatically disabled by the Linux kernel.

2.6.4 Tail Loss Probe

TLP is designed to help in cases where losses occur at the end of a packet burst (Dukkipati, Cardwell, et al. [2012, c23]). If the last packet or packets are lost, the sender will never receive a dupACK, and is forced to wait for an RTO before retransmitting. TLP will send probe segments to get extra ACKs from the receiver, and in the case of loss, dupACKs. A probe segment is sent when the probe timeout (PTO), which is always set to be lower or equal to the RTO timer, signals that an ACK is overdue, i.e., should have been received, and if the connection satisfies the criteria: a) there is outstanding unacknowledged data, and b) it is otherwise idle, i.e., not receiving any ACKs or is limited by CWND/RWND or is application limited.

It is also a requirement that the connection has negotiated SACK to use TLP. If there there is any unsent data available, and the RWND allows it, the probe segment will send new data. Otherwise, the probe segment will retransmit the most recently sent segment.

TLP was included in the Linux kernel version 3.10 (Dukkipati [2013, b3]) and is enabled by default (Linux kernel IPv4 variables [2014, b10]).

2.6.5 RTO Restart

RTOR is a mechanism for TCP and SCTP that provides faster loss recovery for connections with small amounts of outstanding data, such as short-lived or application limited connections Hurtig et al. [2014, c24]. It specifically aims to improve the latencies in situations where fast retransmit cannot be used.

The standard algorithm for managing the RTO timer specifies that it should be reset on each incoming ACK (RFC6298), which in the worst case scenario can give a loss recovery time of $RTO + RTT + delACK$, where $delACK$ is additional delay on the ACK added by TCP delayed acknowledgment.

RTOR suggests to lower the RTO timer by setting it to $RTO - T_{earliest}$, where $T_{earliest}$ is the time elapsed since the earliest of the currently outstanding segments was sent. This alternative RTO timer calculation should only be used when the number of outstanding segments is less than four.


2.6.6 Redundant Data Bundling

RDB is a technique which aims to reduce the latency by resending data even before a packet loss is detected. RDB utilizes the free space in packets that have less payload than the MSS, by bundling older un-ACKed data onto packets with new.

Figure 2.9 shows an example of an Ethernet frame containing a TCP segment with less data than allowed in a full MSS. By using the “free” space in the ethernet frames, one can argue that RDB does not generate extra workload for the nodes in the network. The idea is that when a packet is lost, the data is bundled with the next packet that may already be on its way. This way of bundling data can be regarded as a retransmission mechanism that retransmits data prior to detecting packet loss, unlike regular retransmission mechanisms, that retransmit based on indications that loss has occurred.

While TCP can handle packet reordering, and, with the help of sequence numbers, is able to order the data properly before passing it to the application layer, the data in each TCP packet must be sequential. This constraints the RDB mechanism to only bundle the data of adjacent SKBs that are earlier in the TCP output queue.

Figure 2.10 illustrates how the data is bundled in the TCP segments.

![Figure 2.9: Example of an Ethernet frame for a TCP packet with 100 bytes payload. The headers depicted contain only the necessary information with only the mandatory TCP options.](image)

In figure 2.11 we see a short packet sequence timeline, where a thin stream using RDB, avoids retransmissions due to redundant data being bundled with the segments containing new (unsent) data.

**Sender side modification**

What makes RDB somewhat special, is that it changes significantly how the data is transferred by TCP, while still being a sender-side modification only. No changes are necessary on receiver hosts to be able to handle the RDB packets, which makes it easy to deploy and use in today’s networks.

In an MMORGP scenario, where multiple clients communicate with servers hosted by the game company, it will be beneficial even if only the game server uses
Figure 2.10: Examples showing how RDB bundles the data of previously sent packets onto packets with new data.

RDB. The traffic sent from the server will benefit from RDB, while the unmodified TCP traffic from the clients will not.
Chapter 2. Thin streams

2.7 RDB prototype v1

RDB started out as an idea in Søgård Paaby [2006, a35], to extend the retrans_-collapse function in the Linux kernel. The function tcp_retrans_try_collapse in tcp_output.c is used to collapse (merge) SKBs in the TCP output queue before retransmission. It will try to merge as many SKBs as possible. In the Linux kernel version 2.6.15, which they were studying, the function tcp_reM
trans_try_collapse was only called once to merge two adjacent SKBs, so the idea was to modify this to do this multiple times, as the current Linux kernel code does.

Building upon this idea, Evensen [2008, a5] implemented a more aggressive bundling mechanism that also bundles old data when transmitting packets with new data. These modifications, which we will refer to as the RDBv1, modifies the buffers in the TCP output queue by prefixing the payload of previous buffers to each new buffer. RDBv1 was implemented in the Linux kernel version 2.6.22.1 and later ported to version 2.6.23.8 (Evensen [2008, a5]).

The RDBv1 patch, which is included in code listing C.1, was sent to the Linux net-dev mailing lists in 2009 for feedback. No more work was done on
the RDBv1 patch after that. The implementation details will be briefly described in section 2.7.2.

2.7.1 TCP-engine in Linux

The part of the Linux kernel that implements TCP is called the TCP engine. The main parts of the TCP engine code in the Linux kernel is implemented in the following files:

- **tcp.c**
  Contains initialization code for a TCP connection and the entry points for data coming from user space.

- **tcp_input.c**
  Handles incoming packets and processes the events that are triggered by ACKs.

- **tcp_output.c**
  Processes outgoing packets contained in the TCP output queue and sends the SKBs to the IP-layer for transmission.

- **tcp_cong.c**
  Contains the TCP New Reno CC code as well as the base code for the Linux CC framework.

- **tcp_<CC>.c**
  Contains alternative CC implementations, (where CC is replaced with) such as TCP Cubic, vegas, highspeed, westwood and veno.
SKBs and the TCP output engine

The TCP output engine stores the outgoing data for each socket in a linked list called the TCP output queue as shown in figure 2.13. The data to be transferred is stored in SKBs which is a struct containing next and previous pointers as well as the necessary information needed to build the network packets.

The TCP output queue has two functions: a) to handle the send rate mismatch between the application layer and TCP, and b) to store the sent packets, in case of a needed retransmission, until they are acknowledged.

The elements are ordered by the sequence numbers where sk_write_queue points to the first list element that contains the oldest un-ACKed data, whereas sk_send_head points to the data that has not yet been sent. If they point to the same buffer, no un-ACKed data is present in the queue. If only sk_send_head is NULL, all the data in the queue has been sent and is waiting to be acknowledged.

When ACKs are received, the function tcp_clean_rtx_queue is called to traverse the TCP output engine to remove SKBs if possible.

2.7.2 RDB prototype 1 (RDBv1)

As depicted in figure 2.14, data from the application layer enters the TCP engine in tcp_sendmsg. tcp_sendmsg is modified to call the function tcp_trans_merge_prev after the new data has been inserted into the TCP output queue. tcp_trans_merge_prev does the work of modifying the new SKBs by placing older data into the beginning of the data area. For the incoming packets, tcp_clean_rtx_queue is modified to clean up after the changes made to the TCP output queue, as shown in figure 2.15.
A prerequisite for the RDB technique, is that the receiver end will check the incoming packets by using the end sequence number, and not the start sequence number. If only the start sequence number was to be compared with the expected incoming sequence number, it would classify RDB packets as retransmitted packets and ignore them.

The function in the Linux kernel’s TCP engine that handles incoming data packets, tcp_stream_is_thin, is shown in code listing 2.3.

In code listing 2.3 line 10, we see that it first checks if the starting sequence number is in sequence by testing if it equals rcv_nxt. If that is not the case it will check if the incoming SKB contains new data, by testing if end_seq is greater than the rcv_nxt (line 10). This ensures that the parts of the data that is new in an RDB packet is still processed even when start sequence number is not the expected sequence number for new data.
Linux kernel source

```c
static void tcp_data_queue(struct sock *sk, struct sk_buff *skb) {
    if (TCP_SKB_CB(skb)->seq == tp->rcv_nxt) {
        // This is an in sequence segment
        queue_and_out:
        ....
        return
    }
    if (!after(TCP_SKB_CB(skb)->end_seq, tp->rcv_nxt)) {
        /* A retransmit, 2nd most common case. Force an immediate ack. */
        // Does not contain any new data
        ....
        out_of_window:
        tcp_enter_quickack_mode(sk);
        inet_csk_schedule_ack(sk);
        return
    }
    /* Out of window. E.g. zero window probe. */
    if (!before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt + tcp_receive_window(tp)))
        goto out_of_window;
    tcp_enter_quickack_mode(sk);

    if (before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt)) {
        /* Partial packet, seq < rcv_next < end_seq */
        // This is where RDB packets containing both new and old data are handled
        ....
        tcp_dsack_set(sk, TCP_SKB_CB(skb)->seq, tp->rcv_nxt);
        ....
        if (!tcp_receive_window(tp))
            goto out_of_window;
        goto queue_and_out;
    }
}
```

The initial tests in tcp_data_queue on the incoming data.

Four punctuations indicate code lines that were removed. Comments added for clarity are marked in this color.

Code Listing 2.3: Excerpt from the function tcp_data_queue in tcp_input.c

### 2.7.3 Issues and critique of RDB

Concerns have been raised as to the aggressiveness of RDB, and how it could potentially affect other competing network streams. The RDBv1 has a sysctl option (tcp_rdb_max_bundle_bytes) that limits the number of bytes to bundle for each packet, but it does not use the tcp_stream_is_thin employed in mFR and LT to bundle only when a stream has less than 4 PIFs.

Concerns were also raised about how the SKBs in TCP output queue are modified by adding old data to new SKBs. These complex changes to the SKBs, which are performed when new data enters the kernel, as well as before retransmissions, may lead to issues like silent data corruption, and would require an audit for every...
place where changes to the data are made to be sure the data is always correct (see email D.1).

2.8 Summary

In this chapter we have introduced thin streams and interactive applications that produce network streams with thin-stream characteristics. We have looked at key properties of TCP, as well as different types of CCs, and how they relate to measuring fairness. Further, we briefly discussed some TCP mechanisms implemented in the Linux kernel aimed at improving the latency for thin streams, and finally introduced the RDB mechanism and described the RDBv1 implementation.

In light of the concerns raised about the RDB mechanism’s aggressiveness from section 2.7.3, we will explore how to improve the situation for thin streams further in chapter 3.
Chapter 3

Improving the latency for thin streams

In this chapter we look at the how thin streams perform on the current thin-stream mechanisms in the Linux kernel. We confirm the performance issues of thin streams with a set of preliminary experiments, before we go into detail about RDB, and identify the key issues with the RDBv1 implementation that we want to solve. After reviewing the state of RDBv1 we decided on a few goals on what we wanted to improve:

• **TCP Fairness**
  Look at how to make RDB as fair as possible towards competing traffic while still achieving better latency.

• **Protection against abuse**
  Look at how to avoid any abuse of RDB to gain an advantage over other streams.

• **Code quality and design**
  Look at how the code can be better integrated with the Linux kernel, by separating as much of the code as possible out of the base code of the TCP engine.

### 3.1 Experiments with mFR, LT and ER+TLP

We have performed a set of preliminary experiments with the current mechanisms in the Linux kernel aimed at reducing latency for thin streams. We have used two sender hosts, one sending thin streams, and one sending greedy traffic, with the test properties from table 3.1. The host sending thin streams is set up to run all combinations of the mechanisms LT, mFR, and ER + TLP. The testbed setup used for the experiment is explained in detail in section 5.2.

The plots show the aggregated ACK latency \(^1\) for thin streams versus greedy

---

\(^1\)ACK latency is the time between a data segment is first sent onto the network, until an ACK for the data segment is received.
Table 3.1: Thin stream modification test setup

<table>
<thead>
<tr>
<th>Test</th>
<th>Figure</th>
<th>Name</th>
<th>Streams</th>
<th>Cong</th>
<th>RTT</th>
<th>ITT</th>
<th>Payload</th>
<th>Mechanisms</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Figure 3.2.(a)</td>
<td>Thin TCP</td>
<td>21</td>
<td>Cubic</td>
<td>150</td>
<td>100:15</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Figure 3.2.(a)</td>
<td>Greedy</td>
<td>10</td>
<td>Cubic</td>
<td>150</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>2</td>
<td>Figure 3.2.(b)</td>
<td>Thin TCP</td>
<td>21</td>
<td>Cubic</td>
<td>150</td>
<td>100:15</td>
<td>120</td>
<td>LT</td>
</tr>
<tr>
<td>2</td>
<td>Figure 3.2.(b)</td>
<td>Greedy</td>
<td>10</td>
<td>Cubic</td>
<td>150</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>3</td>
<td>Figure 3.2.(c)</td>
<td>Thin TCP</td>
<td>21</td>
<td>Cubic</td>
<td>150</td>
<td>100:15</td>
<td>120</td>
<td>DA</td>
</tr>
<tr>
<td>3</td>
<td>Figure 3.2.(c)</td>
<td>Greedy</td>
<td>10</td>
<td>Cubic</td>
<td>150</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>4</td>
<td>Figure 3.2.(d)</td>
<td>Thin TCP</td>
<td>21</td>
<td>Cubic</td>
<td>150</td>
<td>100:15</td>
<td>120</td>
<td>LT/DA</td>
</tr>
<tr>
<td>4</td>
<td>Figure 3.2.(d)</td>
<td>Greedy</td>
<td>10</td>
<td>Cubic</td>
<td>150</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>5</td>
<td>Figure 3.2.(e)</td>
<td>Thin TCP</td>
<td>21</td>
<td>Cubic</td>
<td>150</td>
<td>100:15</td>
<td>120</td>
<td>ER+TLP</td>
</tr>
<tr>
<td>5</td>
<td>Figure 3.2.(e)</td>
<td>Greedy</td>
<td>10</td>
<td>Cubic</td>
<td>150</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>6</td>
<td>Figure 3.2.(f)</td>
<td>Thin TCP</td>
<td>21</td>
<td>Cubic</td>
<td>150</td>
<td>100:15</td>
<td>120</td>
<td>LT/ER+TLP</td>
</tr>
<tr>
<td>6</td>
<td>Figure 3.2.(f)</td>
<td>Greedy</td>
<td>10</td>
<td>Cubic</td>
<td>150</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>7</td>
<td>Figure 3.2.(g)</td>
<td>Thin TCP</td>
<td>21</td>
<td>Cubic</td>
<td>150</td>
<td>100:15</td>
<td>120</td>
<td>DA/ER+TLP</td>
</tr>
<tr>
<td>7</td>
<td>Figure 3.2.(g)</td>
<td>Greedy</td>
<td>10</td>
<td>Cubic</td>
<td>150</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>8</td>
<td>Figure 3.2.(h)</td>
<td>Thin TCP</td>
<td>21</td>
<td>Cubic</td>
<td>150</td>
<td>100:15</td>
<td>120</td>
<td>LT/DA/ER+TLP</td>
</tr>
<tr>
<td>8</td>
<td>Figure 3.2.(h)</td>
<td>Greedy</td>
<td>10</td>
<td>Cubic</td>
<td>150</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

The ITT of 100:15 means that the ITTs are generated by a function that produces a uniform distribution with an average of 100 and a standard deviation of 15. The greedy streams are denoted with an ITT and payload of “NA” i.e., not applicable.

Table 3.1: Thin stream modification test setup

streams (See section 5.1.1.1 for details about why and how the ACK latency is measured).

Figure 3.1 shows the first test where all mechanisms are disabled. The orange line indicates the range of extra delay for the thin-stream packets caused by queuing delay in the bottleneck. With an RTT of 150 ms added by netem, the minimum ACK latency is 150 ms which is why the x-axis starts at 150.

Thin-stream packets with ACK latencies that fall after the sharp curve (around the 250 ms mark), have been lost. This is also indicated by the blue line being almost horizontal between 250 and 350 on the x-axis, meaning no ACK latency values in that range.

The vertical dotted lines are placed on every ITT interval (100 ms). Looking closely at the intersection between the ITT-lines and the blue line, we can see small curves. To get a better look at this pattern we have plotted the rest for the results (figure 3.2) using only the upper parts of the y-axis (0.65 – 1).
3.1. Experiments with thin stream mechanisms

Figure 3.1: Plot of ACK latencies showing the effect of queuing delay

In figure 3.2 we can see how much impact packet loss can have on time-dependent thin streams. The distinct stair pattern in the plots illustrates how the latency of the packets are dependent on the data in earlier packets due to HOL blocking. With a loss rate of round 2.5% (see table 3.2) we see that around 10-13% of the packets are affected by increased latency due to packet loss. The stair pattern corresponds well with the vertical dotted lines that show the ITT intervals. The number of steps in the stair pattern reflects how many packets are affected by a loss, which is dependent on the relation between the RTT and ITT. In this case, where the RTT is 150 ms, and the ITT is 100 ms, we see a clear trend where up to 6 packets are affected. We go into further in details on this effect in section 5.4.2.3.
## Table 3.2

<table>
<thead>
<tr>
<th>Test</th>
<th>Name</th>
<th>Data packets</th>
<th>Retrans</th>
<th>Payload B</th>
<th>Bytes Loss %</th>
<th>DupACKs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Thin TCP</td>
<td>61876</td>
<td>1813</td>
<td>7.8 M</td>
<td>2.86</td>
<td>6208</td>
</tr>
<tr>
<td>1</td>
<td>Greedy</td>
<td>118361</td>
<td>3191</td>
<td>171.2 M</td>
<td>2.69</td>
<td>20671</td>
</tr>
<tr>
<td>2</td>
<td>Thin TCP</td>
<td>61881</td>
<td>1717</td>
<td>7.8 M</td>
<td>2.75</td>
<td>6010</td>
</tr>
<tr>
<td>2</td>
<td>Greedy</td>
<td>118198</td>
<td>3032</td>
<td>170.9 M</td>
<td>2.56</td>
<td>20872</td>
</tr>
<tr>
<td>3</td>
<td>Thin TCP</td>
<td>61843</td>
<td>1692</td>
<td>7.8 M</td>
<td>2.71</td>
<td>5902</td>
</tr>
<tr>
<td>3</td>
<td>Greedy</td>
<td>118513</td>
<td>2925</td>
<td>171.4 M</td>
<td>2.47</td>
<td>20416</td>
</tr>
<tr>
<td>4</td>
<td>Thin TCP</td>
<td>62048</td>
<td>1633</td>
<td>7.8 M</td>
<td>2.54</td>
<td>5741</td>
</tr>
<tr>
<td>4</td>
<td>Greedy</td>
<td>118187</td>
<td>2999</td>
<td>171.0 M</td>
<td>2.53</td>
<td>20579</td>
</tr>
<tr>
<td>5</td>
<td>Thin TCP</td>
<td>63153</td>
<td>1645</td>
<td>7.8 M</td>
<td>2.56</td>
<td>5104</td>
</tr>
<tr>
<td>5</td>
<td>Greedy</td>
<td>117997</td>
<td>2845</td>
<td>170.6 M</td>
<td>2.41</td>
<td>20094</td>
</tr>
<tr>
<td>6</td>
<td>Thin TCP</td>
<td>62232</td>
<td>1708</td>
<td>7.8 M</td>
<td>2.70</td>
<td>5115</td>
</tr>
<tr>
<td>6</td>
<td>Greedy</td>
<td>118546</td>
<td>2940</td>
<td>171.5 M</td>
<td>2.48</td>
<td>20896</td>
</tr>
<tr>
<td>7</td>
<td>Thin TCP</td>
<td>63072</td>
<td>1568</td>
<td>7.8 M</td>
<td>2.44</td>
<td>4823</td>
</tr>
<tr>
<td>7</td>
<td>Greedy</td>
<td>117906</td>
<td>2810</td>
<td>170.6 M</td>
<td>2.38</td>
<td>20242</td>
</tr>
<tr>
<td>8</td>
<td>Thin TCP</td>
<td>63114</td>
<td>1662</td>
<td>7.8 M</td>
<td>2.59</td>
<td>5066</td>
</tr>
<tr>
<td>8</td>
<td>Greedy</td>
<td>117952</td>
<td>2898</td>
<td>170.6 M</td>
<td>2.45</td>
<td>20475</td>
</tr>
</tbody>
</table>

Table 3.2: Thin stream modifications test results
3.1 Experiments with thin stream mechanisms

Figure 3.2: ACK latencies for thin vs greedy streams using the mFR, LT and ER mechanisms
Chapter 3. Improving the latency for thin streams

Figure 3.2
From the plots with ER and TLP enabled we see that the thin-stream latencies are somewhat lower, where the 90\textsuperscript{th} percentile is consistently at 250 ms, whereas the plots without ER and TLP have the 87 – 88\textsuperscript{th} percentile at 250 ms.

For LT to have an affect, it requires loss of retransmitted packets. Losing retransmissions will naturally have a very negative effect on the latency, but is also a less likely event than losing only the first transmission. Therefore, we can not expect to see considerable effect from LT in these conditions.

The goal of this experiment is not to perform an in-depth analysis on the effects of the thin-stream mechanisms, and how they compare to each other. We show these results to illustrate the problem that none of the current mechanisms solve, namely the HOL blocking effect caused by retransmission latency.

### 3.1.1 Head-of-line blocking

Due to the in-order guarantee of TCP, any data segments arriving after a lost packet are held back on the receiver, also known as HOL blocking. To uphold the ordering guarantee, the data that is held may not be sent to the application layer until the lost data segment has been retransmitted.

The stair pattern in the plots illustrates how a packet loss not only affects the latency of the lost packet, but due to HOL blocking, a range of packets coming after. These grave delays, can be alleviated by increasing the effectiveness of the traditional re-transmission mechanisms, such as mFR. However, the extra delay of minimum one RTT (for the dupACK to return and the retransmission to arrive) is still very high, which leaves great potential for improvements.

### 3.2 TCP Fairness and RDB

With no mechanisms to control how much and how often data is bundled, RDBv1 was deemed too aggressive, and rightfully so. We therefore started out with the task of finding the sweet spot for bundling data so that we get a best possible latency improvement without affecting competing streams negatively.

It is of vital importance that RDB cannot be misused to get an advantage over other streams by ignoring any signs of congestion. At a minimum, RDB should meet the criteria that an RDB stream can not get an advantage throughput-wise over a competing greedy TCP stream.

#### 3.2.1 RDB hiding loss events

An interesting effect with RDB is that it hides the loss of a packet, if the next packet containing the lost data is transferred successfully. The amount of loss required for TCP to detect a loss depends on the number of previous packets that are bundled. If each new data segment from the application is 700 bytes, the data of only one packet can be bundled with each sent packet. If only one packet is lost, the next packet will hide the loss, but if two packets in a row are lost, the data in the next packets will not be in sequence and hence trigger dupACKs. If the data segments from the application are 400 bytes, as depicted in figure 2.10, there is room
to bundle the data of two earlier packets in the TCP output queue. If the frames in subfigure 2.10.(a) and subfigure 2.10.(b) are lost, S1 is still transmitted in subfigure 2.10.(c), but if that frame is lost as well, the data in frame insubfigure 2.10.(d) will cause a gap in the sequence numbers.

If an application sends segments of 100 bytes to the kernel, and the MSS is 1460 bytes, there will be room to bundle the data of 13 previously sent packets. This will require 14 lost packets in a row, for a gap in the sequence space to occur on the receiver side, resulting in a dupACK.

3.2.1.1 Active queue management

When the incoming buffer queue of a router in a network is constantly full the network is congested (at least that path in the network). When the router has no more room it simply drops any incoming packets, which is known as the tail drop algorithm, the traditional solution for the most basic routers.

Active queue management (AQM) is the smarter alternative to tail drop, where the queuing discipline used by the router drops packets according to some algorithm.

Explicit Congestion Notification

Some routers with AQMs may also support ECN, an extension to the TCP and IP protocols. In an ECN scenario, a router that is not yet congested, but finds the traffic flow becoming too heavy, may set the ECN flag on packets before forwarding them on the correct path. This will work with regular TCP as well as with an RDB stream, as the ECN mark will be detected by the receiving end host which echoes back the ECN mark on the ACK-reply. On receiving the ACK, the sending host will be able to do adjustments to the send rate accordingly.

Loss based AQMs

Queuing disciplines like Random early detection (RED), works by dropping incoming packets before the router is too congested. This induced loss is a congestion indication signal to the sender node that there is congestion, so that it can make adjustments to prevent overflowing the network.

For RDB streams we encounter a problem with these types of AQMs. A TCP stream will react to the loss of a single packet, but with an RDB stream, the loss would be hidden by the redundant data in the next packet resulting in the congestion signal from the AQM being completely ignored. In this case RDB would get an advantage over other streams.

3.2.2 Abusing and misusing RDB

The potential of abusing RDB to gain an advantage over other network streams, or to sabotage for other streams on the network, is significant. As RDB hides some of the loss from the TCP engine, the CWND could grow faster than a regular TCP stream and not back off as fast as other streams would when they detect loss.
RDB is meant to help thin streams, but when used on a greedy stream where the application adapts the send calls to the kernel to improve the potential of bundling redundant data, it could get an advantage throughput-wise. Therefore, care has to be taken to prevent any abuse of RDB by applications that do many send calls with less than half an MSS per call.

The kernel receives the data to be transferred through the system call send. Normally the kernel will delay the sending of the data to avoid smaller packets being transmitted. This is because of the Nagle’s Algorithm setting which is enabled by default in the Linux kernel. If the user does two calls to send with 400 bytes each, the first call will result in a new SKB containing the data being added to the TCP output queue. On the second call to send it checks if there is unsent data in the queue, and if there is free space in the newest SKB (which we know it is in this example), the new data will be appended to the newest SKB in the queue.

By disabling Nagle’s Algorithm the kernel will not delay sending the data, but instead send the SKB with the new data segment as soon as possible, as long as the send window allows it. Therefore two consecutive calls to send with a small amount of data will result in two packets to be transmitted.

3.2.2.1 Tests with senders abusing and misusing RDB

We ran tests to verify how greedy RDB streams will affect competing greedy streams. We have named these RDB streams Thick to indicate that these streams are not requesting a maximum amount of bandwidth as traditional greedy streams. With an ITT of 10 ms and payload of 400 bytes they are “only” requesting 40 KB/s $(40000 = 100 \times 400)$ which is more than the thin-stream applications in table 2.1, but they would still be application limited in our testbed if there were only a few simultaneous streams.

Subfigure 3.3.(a) shows the result of 6 greedy TCP streams competing with 6 RDB streams. The RDB streams do calls to send with 400 bytes every 10 ms, which gives room to bundle the data of two previously sent packets for each packet sent. Each packet transferred on the wire will contain 1200 bytes, where 800 of them are redundant.

When the RDB streams are limited by maximum three PIFs, they end up not bundling any data, and as a result, get the same throughput as the regular TCP streams. As soon as the limit is raised, the RDB streams get substantially better throughput than the greedy streams.

If the RDB streams in subfigure 3.3.(a) lose two consecutive packets, the data of the first and second lost packet is still received in the packet coming next, which contains all the lost data as well as some new data. Neither the receiver, nor the sender, will know that the data was lost, and the result is that the senders will not back off as the other streams will, due to detecting loss. It is clear that the RDB streams get an unfair advantage by not reacting to loss events.
PIF (−1) is the threshold for how many packets in flight are allowed when data is bundled, i.e., “PIF:4” means a maximum of three PIFs.

Figure 3.3: The aggregated throughput for greedy streams competing with thick TCP and RDB streams
3.3 PIFs as a thin-stream indicator

As mentioned in section 2.6, the TCP modifications, LT and mFR, are only active on thin streams, assured by the function `tcp_stream_is_thin` in code listing 2.1.

The limit of maximum three PIFs, which is hardcoded in the function `tcp_stream_is_thin`, is justified because with three or fewer PIFs the stream will not receive the three required dupACKs to trigger a fast retransmit within one RTT.

For testing if a stream will have problems triggering a fast retransmit, it works fairly well, but that it also the limited scope it tests for. Using a static packets in flight limit (SPIFL) as a metric for classifying thin streams in general, is however, to some extent flawed. The main reason is that the PIFs of a flow is highly dependent on the RTT. Therefore, two application limited streams that send the exact same amount of data, may be classified differently, even though their strict latency requirements are the same. In a network with latencies as low as a few milliseconds, the PIFs limit would allow streams, that would be considered greedy in a network with higher latencies, to use the thin-stream modifications. For this reason, the hardcoded SPIFL must be set very low, to minimize the potential for non-thin streams to use the mechanisms.

While many of the applications listed in table 2.1 will frequently have less than four PIFs on a connection with 70 ms RTT\(^2\), some of them will have more. These streams will fall outside the limited scope of the current test, while still, to some degree, suffering from the same problems.

\(^2\)The median RTT of the connections measured on Google’s test servers in 2010 is 70 ms (Dukkipati, Refice, et al. [2010, a36])

Table 3.3

<table>
<thead>
<tr>
<th>Test</th>
<th>Name</th>
<th>Type</th>
<th>Streams</th>
<th>Cong</th>
<th>RTT</th>
<th>ITT</th>
<th>Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TCP</td>
<td>Thick TCP</td>
<td>6</td>
<td>Cubic</td>
<td>150</td>
<td>10</td>
<td>400</td>
</tr>
<tr>
<td>1</td>
<td>TCP</td>
<td>Greedy</td>
<td>6</td>
<td>Cubic</td>
<td>150</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>2</td>
<td>RDB PIF:4</td>
<td>Thick RDB</td>
<td>6</td>
<td>Cubic</td>
<td>150</td>
<td>10</td>
<td>400</td>
</tr>
<tr>
<td>2</td>
<td>RDB PIF:4</td>
<td>Greedy</td>
<td>6</td>
<td>Cubic</td>
<td>150</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>3</td>
<td>RDB PIF:20</td>
<td>Thick RDB</td>
<td>6</td>
<td>Cubic</td>
<td>150</td>
<td>10</td>
<td>400</td>
</tr>
<tr>
<td>3</td>
<td>RDB PIF:20</td>
<td>Greedy</td>
<td>6</td>
<td>Cubic</td>
<td>150</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>4</td>
<td>RDB PIF:100</td>
<td>Thick RDB</td>
<td>6</td>
<td>Cubic</td>
<td>150</td>
<td>10</td>
<td>400</td>
</tr>
<tr>
<td>4</td>
<td>RDB PIF:100</td>
<td>Greedy</td>
<td>6</td>
<td>Cubic</td>
<td>150</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Table 3.3: Greedy vs RDB misuser test setup
3.4 Congestion control: A cause of reduced latencies

A major issue for time-dependent, application limited TCP streams, is that the congestion avoidance technique of drastically reducing the CWND is very harmful to the latency. It is not only an issue for thin streams, but also with applications such as video streaming (Tan and Zakhor [1999, a37]).

In scenarios where a thin stream is competing with greedy streams, the greedy streams are to blame for the reduced latencies of the thin stream. This is because of how the CC of the greedy streams work, where itcauses a repeated filling and draining of the queues on the bottleneck node in the network path (Stewart et al. [2011, a9]). By continually trying to utilize any available capacity on the path, by increasing the send rate until losses occur, they ensure that other streams that share a link to the bottleneck node also lose packets.

This might be considered fair practice when the competing traffic a) has the same behavior, and b) when the competing traffic does not care about per-packet latency. But for interactive applications producing thin streams, the consequences are drastic, and unfair. Even when the thin streams are only requesting a small share of the total link capacity, they are forced to reduce their send rate.

Reducing the negative effect the CC has on such network streams has been one of the driving forces behind equation based CCs such as TFRC and binomial CC. By growing in a less aggressive manner compared to slow-start, they argue that it is fair to reduce the send rate in a slower manner.

What is the problem, exactly?

The most obvious problem is that the thin streams, who are only transmitting a very limited amount of data to begin with, are forced to lower their send rate due to the aggressive behavior of greedy streams. Due to the lowered send rate, some of the data must be queued in the TCP output queue for a certain time, causing increased latencies.

The function in the CC that controls how the CWND grows is implemented in the entry point cong_avoid in the Linux CC framework (code listing B.13 line 15). The implementation for TCP New Reno can be seen in Code listing B.1, but the logic is similar for other CCs like TCP Cubic. The function tests if the connection is in slow-start (Code listing B.1 line 16), and if so, grows the CWND accordingly. If not in slow-start mode, meaning congestion avoidance mode, the CWND grows in a slower manner.

Some CC implementations, such as TCP New Reno and TCP Cubic also perform a test at the beginning, by calling tcp_is_cwnd_limited (Code listing B.1 line 12). This seemingly harmless test is what causes problems for thin streams.

The function tcp_is_cwnd_limited tests if the connection’s send rate is currently limited by the CWND. If the connection does not utilize the available CWND, it presumably does not need a bigger CWND. While this is reasonable for a greedy stream, as it will prevent the stream from building up a CWND that the network cannot support, it causes issues for thin streams. When a thin stream
loses a packet, the AIMD algorithm will reduce the CWND drastically, causing
data to be queued in the TCP output queue instead of being sent immediately.

One might wonder if it really is such a big problem, as the CWND would
eventually grow back for the thin streams, and again allow them to send the data
immediately without any queuing delay. While this is true, the lowered send rate
increases the latency, causing reduced service quality.

Recent changes to the Linux kernel’s TCP

In Linux kernel version 3.16, the function tcp_is_cwnd_limited was rewritten,
which changes the situation for thin streams. The two commits e114a7 and ca8a22,
which we believe can be attributed to the effects we observe, cause increased la-
tencies for thin streams.

The implementation in Linux kernel version 3.15 (code listing B.2 would, at
call time, simply test if the number of PIFs was not less than the CWND. The first
commit (e114a7) aims to improve the accuracy of when tcp_is_cwnd_limited
reports a stream as limited by the CWND. An important change is that it moves the
PIF < CWND test into the CWND validation mechanism (tcp_cwnd_validate)
performed in the TCP output engine. Therefore, the CCs no longer test if the
stream is network limited at the time it is asked to grow the CWND.

The second commit (ca8a22) tries to fix a problem with the first commit, where
the CWND could be larger than necessary. The result is that after the CWND is
reduced due to packet loss, the CWND does not grow back to the previous level.
This prevents the sender from maintaining the number of PIFs it needs to transmit
the data segments without being delayed in the TCP output queue. When the
CWND does not grow, more data is buffered in the TCP output queue leading to
bigger packets being sent, which effectively enforces a behavior similar to Nagle’s
Algorithm.

Whether the effect on thin streams is unintended, meaning it can be regarded
as a bug, or if this is really how the kernel developers intended it, is yet to be
found out. This behavior has been confirmed up to Linux kernel version 3.19-rc7,
the latest release at the time of writing. The implementation of tcp_is_cwnd_limited
in Linux kernel version 3.16 is shown in code listing B.3.

3.5 Summary

In this chapter we have looked at the latency for thin stream, and what causes
increased latencies for such streams. We have presented the results from a set of
preliminary experiments, that tests thin streams using the mechanisms mFR, LT,
and ER+TLP against competing greedy traffic. From the results we see how the
loss of a packet reduces the latency for many of the following packets due to HOL
blocking.

We have gone into details about different issues with the RDBv1 implementa-
tion as well as topics on how to detect if a stream is thin. We have described
how the traditional CCs for TCP control the CWND can cause increased latencies
for thin streams, as well as how changes to the CC in Linux kernel version 3.16 further increases the latencies for thin streams.
Chapter 4

RDB prototype v2

As discussed in section 3.2, RDBv1’s bundling scheme is considered too aggressive. The increased network data for a thin stream using RDB may be considered a problem, however, compared to greedy streams, the extra load on the network is minimal. However, the problem of RDB hiding loss is strictly necessary to address, as it prevents the TCP engine and the regular CCs from controlling the stream in response to network congestion.

For this reason, we have investigated how to make RDB less aggressive, where our main goal has been to ensure that RDBv2 behaves in a TCP friendly manner towards competing network traffic, while still being effective in reducing the application layer latency for thin streams.

In this chapter, we address the issues described in chapter 3, and present how we tried to solve them in RDBv2. We first look at the thin-stream classification mechanism described in section 3.3, and how it can be improved to more precisely identify thin streams. Further, we go into detail about how to detect the hidden loss in an RDB stream, before we describe the re-implementation of the bundling code and how it differs from the RDBv1 implementation. Finally, we explain the background for the CC mechanism we have implemented in RDBv2, which we refer to as RDBv2-CC.

4.1 Classifying thin streams

The Linux kernel “defines” a stream as thin by the number of PIFs being less than four (see code listing 2.1), and uses this to decide whether to use the mechanisms tcp_thin_dupack and tcp_thin_linear_timeouts described in section 2.6.

Using the same test to restrict RDB would certainly make RDBv2 less aggressive than RDBv1, but that would result in no redundant bundling in many circumstances where it could be beneficial, e.g., when the RTT is so high that the PIFs exceed three very easily.

As described in section 3.3, the use of the static PIF value is somewhat flawed, as the underlying network environment is not taken into account. If a PIF limit should make sense, it must be calculated so that an application is treated similarly under different network environments. The SPIFL, compiled by the function tcp_stream_is_thin, will be stricter to a stream with a higher RTT, even when
the sending rate remains constant. Therefore, the stream’s RTT should be taken into account when limiting based on PIFs.

By basing the limitation on the application’s sending rate, it is easier to make a distinction between the applications we believe deserve to benefit from RDB, and those who do not. We have therefore chosen to do this by basing the thin-stream classification on a maximum number of PIFs relative to the RTT. By dynamically calculating the PIF limit as a factor of the flow’s RTT, we ensure that network streams are treated more equally under differing network environments. This dynamic limit, which we call dynamic packet in flight limit (DPIFL), is calculated according to equation 4.1,

$$\text{DPIFL} = \frac{\text{RTT}_{\text{min}}}{\text{ITT}_{\text{min}}}$$  \hspace{1cm} (4.1)

where $\text{RTT}_{\text{min}}$ is the minimum RTT value observed, and $\text{ITT}_{\text{min}}$ is a static limit defining the minimum allowed ITT.

Figure 4.1 shows the DPIFL plotted for RTTs ranging from 10 to 160 ms, with minimum ITTs of 10 ms and 20 ms. The ratio between DPIFL-10 and DPIFL-20 is 1 : 2, such that DPIFL-10 allows for the double number of PIFs compared to DPIFL-20 on the same RTT.

### 4.2 Loss detection

The fact that RDB hides loss events, as described in section 3.2.1, causes it to not react well to losses in many situations. It gives the streams an unfair advantage,
which goes against the TCP-friendliness concept described in section 2.5.3. We have therefore investigated how to detect loss events that RDB hides from the TCP engine.

When the TCP data sender receives a dupACK, it must consider if it was caused by reordering or by loss. With RDB, the potentially lost or reordered data is included with the next packet. Therefore, dupACKs are not sent when sporadic loss of only one packet occurs, as explained in section 3.2.1. The probability of receiving dupACKs depend solely on how many (already sent) data segments are being bundled with each packet. If each RDB packet includes the bundled data of four previously sent packets, the loss of five consecutive packets is required for a gap in the data to appear on the receiver side, resulting in a dupACK being sent.

We have investigated the following techniques for detecting packet loss in an RDB stream:

- **Multiple packets ACKed**
  When an ACK covers multiple SKBs, it is a good indicator that a packet has been lost. This test alone cannot distinguish between a lost data packet and a lost ACK. If reordering occurs, this technique may also fail. However, for thin streams, the high ITT itself reduces the chances of packets arriving out-of-order.

- **TCP time stamps**
  We investigated if TCP timestamps could be used to improve the loss detection accuracy, similarly to how the Eifel detection algorithm, described in section 2.4.5.2, distinguishes between reordering and loss.

- **DSACK**
  The DSACK extension to SACK, enables the receiver to accurately inform the sender of duplicate received data segments. We have looked at how this mechanism behaves in a RDB scenario, and how this could be used for loss detection.

In the next sections, we will go into detail about these two techniques for detecting loss.

### 4.2.0.2 ACKs covering multiple segments indicating loss

One idea for detecting lost packets is to look at the received ACKs. If a sender can expect to receive one ACK for every data packet sent, it can presume that any deviations to this is caused by data packets, or ACKs on the reverse path, being lost. If an incoming ACK acknowledges multiple SKBs in the TCP output queue, it must be caused by:

- a data packet being lost, resulting in all or some of the redundant data in the next packet effectively not being redundant after all.
• an ACK on the reverse path being lost, causing the ACK on the next packet to advance \texttt{snd_una}\footnote{\texttt{snd_una} is a variable in the TCP socket struct used to keep track of the greatest un-ACKed sequence number.} by two segments.

There are only two flaws in this technique. One being packet reordering, which can cause the first ACK to be a dupACK, and the second ACK to acknowledge two segments. In that scenario, it would be incorrect to consider the second ACK, that acknowledges two packets, as an indication of loss. The other problem is caused by lost ACK packets. Distinguishing between loss of data packets, and loss of ACK packets on the reverse path can be difficult, and with this technique alone, it is not possible.

### Delayed ACKs and RDB

A regular TCP stream must consider that TCP delayed acknowledgment may be enabled on the receiver. Receiving ACKs that span (acknowledge) multiple segments, is therefore a part of the normal workflow, and should not be treated as a possible loss signal. Due to this, the loss detection technique described in section 4.2.0.2 cannot be used for regular TCP.

However, delayed ACKs are used only when data is received in sequence, as seen on line 3 in code listing 2.3. When RDB bundles previously sent data, the sequence number does not equal the expected sequence number, and TCP will perform a quick ack by calling \texttt{tcp_enter_quickack_mode} (see code listing 2.3 line 15 and 22). Therefore we know that when sending an RDB packet, an ACK will be sent for each data packet delivered to the receiver.

### 4.2.0.3 TCP Timestamps

Similar to how the Eifel detection algorithm solves the \textit{retransmission ambiguity problem}, we looked at how TCP timestamps can be used to identify lost packets in an RDB stream.

Figure 4.2 shows a scenario where an RDB stream bundles one older packet with each data packet. When packet \textit{B} is lost, the next ACK sent in reply, acknowledges segment \textit{C} using \textit{C}'s TS\text{val} in the \textit{echo-reply} field. Due to the requirement that the \textit{echo-reply} field of an ACK, always must reflect the earliest data segment that \textit{this} ACK acknowledges, the \textit{echo-reply} field in dupACKs will never reflect the packet that triggered it. This is because dupACKs do not acknowledge any new data.

When a dupACKs is received, with the \textit{echo-reply} field reflecting segment \textit{C}, we can speculate that the previously missing ACK was simply caused by a re-ordering, and not by a lost packet. However, if there is more outstanding data that contains segments newer than \textit{C}, it is an ambiguous situation, as the \textit{echo-reply} does not tell you if the incoming packet had old data (before \textit{C}), or contained data newer than the highest in-sequence segment. Had the \textit{echo-reply} of dupACKs reflected the timestamp of the incoming packet that triggered it, it would be possible to distinguish between the two cases.
4.2. Loss detection

Figure 4.2: Example of TCP timestamps with TCP delayed acknowledgment (From RFC7323)

**4.2.0.4 DSACK**

As described in section 2.4.1, the DSACK extension to SACK enables the receiver to accurately inform the sender of duplicate received data segments. The consequence is that when DSACK is enabled on an RDB stream, the redundant data bundled with newer packets causes the receiver to populate the SACK option field with the range of already received data. This can be used to identify lost packets.

Figure 4.3 shows an RDB stream that bundles one previous segment with each data packet. As we can see, each ACK’s SACK field is populated with the DSACK range of the redundant segment.

Figure 4.4 illustrates how it looks when reordering occurs. Due to reordering of one packet, the third data packet arriving on the receiver side contains no old (already received) data, and consequently, the ACK sent in return does not contain any SACK information. This will only happen if reordering has occurred, or if one packet was lost. If an ACK is lost, the data sender will notice that an ACK was skipped, but due to the DSACK range in the next ACK, it can conclude that the data had already been received, and hence, it must be the ACK that was lost.

The SACK option, along with the DSACK extension, will help us solve the ambiguous case that the TCP timestamps could not. By providing a DSACK range when the packet that triggered the dupACK contains old data, and a regular SACK range when the data is newer than the highest in-sequence segment,
Figure 4.3: Example of RDB stream with SACK option enabled.

we can unambiguously identify which packet triggered the dupACK in the case of reordering.
4.3 Implementing redundant bundling

Re-implementing the bundling code was necessary, because the initial implementation (RDBv1) turned out to be more intrusive to the core of the Linux kernel’s TCP engine than strictly necessary.

In a response on the Linux net-dev mailing list, Ilpo Järvinen suggested a different approach than what was used in RDBv1. Instead of modifying the payload of the SKBs in the TCP output queue when receiving data from user space, he suggested to create the SKBs containing redundant data on-the-fly at send time (email D.1). This approach would avoid any modifications to the data in the TCP output queue, and allow cleaner entry points into the codebase.

The part of RDBv2 that implements bundling is a re-implementation of the functionality implemented in RDBv1. While, functionally, there is little difference between these implementations, RDBv2 has been structured differently, and organized into separate files which makes it easier to read and understand.
Figure 4.5: Call graph for TCP output engine, where the changes made for RDBv2 are marked in green.

For convenience, the code listings for the implementation is placed in the appendix B, and is referred to in the descriptions of each function.

### 4.3.1 Entry point for sending custom SKBs

The code in RDBv2 responsible for modifying the SKBs, is modeled after the suggestion by Ilpo Järvinen (see email D.1). Instead of modifying the SKBs directly in the TCP output queue, RDBv2 creates a new SKBs on the fly, and copies redundant data from the SKBs in the TCP output queue into the linear page buffer of the new SKB.

All the call-paths in the program flow for sending new data (TCP output engine, tcp_output.c) ends up in the function tcp_write_xmit (see figure 4.5).

The function tcp_write_xmit (see code listing B.4) is called to write packets with unsent data to the network. It will keep sending packets from the TCP output queue as long as they fit into the send window, and the argument for the push_one parameter is 0.

In the function tcp_write_xmit, we have added a call to a new entry point (pkt_send) in the Linux CC framework (see code listing B.13 line 25). As shown in code listing B.4, the call to the new entry point is added just before the call to tcp_transmit_skb (Line 13). If the return value of the pkt_send function is zero, the packet has been successfully sent, and the default call to tcp_transmit_skb is skipped (Line 19). If the return value is not zero, the code proceeds to execute the standard call to tcp_transmit_skb (Line 24).
4.3. Implementing redundant bundling

4.3.2 Performing redundant data bundling

The code that tests if an RDB packet may be created, as well as creates and sends the packet, is implemented in rdb_skb.c. The function do_rdb in rdb_cc.c is called through the entry point pkt_send, as shown in figure 4.10. Following comes a short explanation of the main tasks of the most relevant functions for performing the bundling.

**do_rdb()**

This function (see code listing B.5) is called by the pkt_send entry point, and does the following:

- Calls run_rdb (Code listing B.6) which tests if the RDB socket option is set, and that the stream is thin by calling tcp_stream_is_thin_dpif.

- Calls redundant_data_bundling (Code listing B.9) which tests if it is possible to bundle. If so, it creates a new SKB with redundant data.

- Calls tcp_transmit_skb to transmit the RDB packet.

**run_rdb()**

This function performs the initial test to see if the stream is currently eligible for redundant data bundling. It will first test if the TCP_THIN_RDB socket option has been set (see line 2 in code listing B.6). Next it will test whether to use DPIFL or SPIFL to classify the stream is thin or not.

If DPIFL is enabled, it will use the function tcp_stream_is_thin_dpif (code listing B.7), which implements the DPIFL test described in section 4.1.
If DPIFL is not enabled, it will use the function `tcp_stream_is_thin_spif` (code listing B.8), which is a modified version of `tcp_stream_is_thin` (code listing 2.1). To easily perform tests with different SPIFL values, this function reads the PIF limit from a sysctl variable instead of using a hardcoded limit. The second difference is that we have removed the call to `tcp_in_initial_slow_start`, which restricts the thin-stream mechanisms to streams that have experienced packet loss. This is because we want to bundle data even before loss has occurred.

**redundant_data_bundling()**

The function `redundant_data_bundling` (Code listing B.9) tests if it is possible to bundle any redundant data, and if so, returns a new SKB containing redundant data followed by the new data to be transmitted.

- First, it calls `rdb_can_bundle_check` (Code listing B.10) which traverses the TCP output queue, and returns the first SKB (with the oldest data) that can be bundled.

- If a valid SKB is returned by `rdb_can_bundle_check`, it calls `create_rdb_skb` which creates the new SKB and returns it. The resulting SKB will contain some new data, as well as the data from one or more older SKB, such as depicted in figure 2.10.

**rdb_can_bundle_check()**

`rdb_can_bundle_check` (Code listing B.10) goes through the TCP output queue to find the first (oldest) SKB containing data that can be bundled. Starting at the SKB that the TCP output engine tries to send, it traverses the queue in reverse order and tests if the data of each of the SKBs can be included in the new SKB. If there is room left for parts of the data in an SKB, it will save the offset into the data buffer of that SKB, which is passed into the function `create_rdb_skb` that creates the SKB and copies the data.

To control the amount of bundling in each SKB, this function also uses two sysctl variables, `sysctl_tcp_rdb_max_bundle_bytes` and `sysctl_tcp_rdb_max_bundle_skbs`. These variables can optionally be set before a test, which will be used for all the RDB streams in the test. If any of these variables are not zero, they will limit the maximum number of bundled bytes per RDB packet, or the maximum number of SKBs to bundle from.

### 4.3.3 Bundling on retransmission

To measure the effect of bundling on retransmissions, a separate set of tests would be required, so that the results of bundling on retransmissions could be compared to the tests without bundling on retransmissions. To reduce the number of experiments, we decided to only test bundling when transmitting new data.
The RDBv2 code presented here does not implement the pkt_send function hook on retransmission, however, the changes required to do so is simple, and very similar to the code on line 13 in code listing B.4.

4.4 RDB Congestion control background

In addition to the functionality required to bundle data, the RDBv2 implementation contains a CC that aims at solving a range of issues. Dealing with the issue of RDB hiding loss events (described in section 3.2.1) is strictly necessary to make it TCP friendly. Also, preventing abuse, by implementing functionality to limit when bundling may occur, is a strict requirement to ensure TCP-friendliness. Solving these main tasks could best be done by creating a new CC module.

4.4.1 TFRC-like congestion control

TFRC, as specified in RFC5348, requires the receiver side to calculate the loss rates and report this to the sender side in feedback packets. One advantage to this is robustness to loss of the feedback packets, as this would not influence the loss rate calculations.

An important benefit with the initial RDB implementation, RDBv1, is that only sender side modifications are required. By introducing CC functionality that requires receiver side modifications, we would not be able to keep the TCP compatibility which is one of the main goals we have set for RDBv2. RFC5348 also specifies sender-based variants, where the loss rate calculation is moved to the sender side, which is what we have chosen to base our implementation on.

The goal of TFRC is to provide a CC that has less variation of the throughput compared to TCP. Where TCP reacts drastically to loss by either halving the CWND, or in the case of an RTO, set it to one, TFRC seeks to avoid these drastic reductions to the send rate by calculating the average loss over a certain period of time. In the case of just a single lost packet, the CWND is not reduced as much as for TCP. While TCP will not reduce the CWND for multiple lost packets within the same window, it will not treat losses in different windows any differently.

TFRC, on the other hand, divides the sequence space into loss intervals, which makes a loss history that is used to calculate the allowed send rate. In addition to packet losses, congestion events, such as ECN marked packets, can be used to build the loss history. From the packet history, a loss event rate \( p \) is calculated, which in turn determines the speed at which TFRC responds to congestion.

4.4.1.1 Loss History

Where TCP sequence numbers are based on the number of bytes being sent as payload, TFRC packets are given a sequence number which is incremented by one for each packet. The length of a loss interval is specified in RFC5348 as:

“If a loss interval, A, is determined to have started with packet sequence number S_A and the next loss interval, B, started with packet sequence..."
number $S_B$, then the number of packets in loss interval $A$ is given by $(S_B - S_A)$.

This approach enables the sender to calculate how serious the congestion is by looking at how many packets have been lost over time.

Like TCP, TFRC will treat multiple lost packets within the same send window differently. If a lost packet is within the same RTT as the lost packet that started the current interval, it will include it in the current interval instead of creating a new one.

The number of loss intervals contained in the loss history affects how quickly the loss event rate is adjusted when the loss rate changes. The more loss intervals used, the slower the loss event rate will adjust. For TFRC, it is specifies that eight loss intervals should be used.

### 4.4.1.2 Calculating average loss interval

A sequence of weights are used to ensure that the loss event rate changes smoothly. From RFC5348, the weights are calculated by pseudo code 4.1, such that the first four (of eight) weights are 1.0 (meaning no change), and the next weights reduce the significance of the values to be weighted more and more the older they are.

The average loss interval is calculated according to pseudo code 4.2, where the weights are applied to the loss intervals. Two calculations of the average loss interval are made, with and without the current loss interval, i.e., the loss interval that is currently open. The maximum of the two calculated values is used as the mean value, such that the current loss interval will only be used if it is large enough to increase the average loss interval.

---

**Pseudo Code**

Weights $w_0$ to $w_{(n-1)}$ are calculated as:

```plaintext
If (i < n/2) {
    w_i = 1;
} Else {
    w_i = 2 * (n-i)/(n+2);
}
```

Thus, if $n=8$, the values of $w_0$ to $w_7$ are:

1.0, 1.0, 1.0, 1.0, 0.8, 0.6, 0.4, 0.2

---

Code Listing 4.1: TFRC pseudocode for calculating the weighting for the average loss intervals (RFC5348)
4.4. RDB Congestion control background

<table>
<thead>
<tr>
<th>Pseudo Code</th>
</tr>
</thead>
</table>
| I_tot0 = 0;
I_tot1 = 0;
W_tot = 0;
for (i = 0 to k-1) {
    I_tot0 = I_tot0 + (I_i * w_i);
    W_tot = W_tot + w_i;
}
for (i = 1 to k) {
    I_tot1 = I_tot1 + (I_i * w_(i-1));
}
I_tot = max(I_tot0, I_tot1);
I_mean = I_tot/W_tot;
The loss event rate, p is simply:

\[ p = 1 / I_{mean} \]

Code Listing 4.2: TFRC pseudocode for calculating average loss interval (RFC5348)

<table>
<thead>
<tr>
<th>Pseudo Code</th>
</tr>
</thead>
</table>
| I_tot0 = 0;
I_tot1 = 0;
W_tot = 0;
for (i = 0 to n-1) {
    I_tot0 = I_tot0 + (I_i * w_i);
    W_tot = W_tot + w_i;
}
for (i = 1 to n) {
    I_tot1 = I_tot1 + (I_i * w_(i-1));
}
if the current loss interval I_0 is "short"
    then I_tot = I_tot1;
else I_tot = max(I_tot0, I_tot1);
I_mean = I_tot/W_tot;
The loss event rate, p:

\[ p = 1 / I_{mean} \]

Code Listing 4.3: TFRC-SP pseudocode for calculating average loss interval (RFC4828)

4.4.1.3 Calculating the send rate

As the unit of the calculated send rate from equation 2.2 is bytes/s, and the sender is limited by the CWND, the send rate is converted into a CWND value by equa-
Chapter 4. RDB prototype v2

4.2

\[ RTT_{sec} = \frac{10^6}{srtt_{\mu s}} \quad X_{RTT} = \frac{X_{sec}}{RTT_{sec}} \]  

\[ CWND = \frac{X_{RTT}}{psize} = \frac{X_{sec} \cdot srtt_{\mu s}}{10^6 \cdot psize} \]

We first divide the send rate in seconds \((X_{sec})\) by the number of RTTs in a second \((RTT_{sec})\) to get the send rate per RTT \((X_{RTT})\). We then divide \(X_{RTT}\) by the size of the packets \((psize)\) to get the corresponding CWND value. For the \(psize\) we use the same value as used for packet payload size in the throughput calculation (Equation 2.2).

The variable \(b\) in equation 2.2 denotes the maximum number of packets that can be acknowledged by a single ACK. Due to TCP delayed acknowledgment, this will normally be 2, but RFC5348 recommends using 1, because TCP is allowed to send an ACK for every packet. Considering that RDB effectively disables TCP delayed acknowledgment, we are safe to use 1 for the calculations.

4.4.1.4 TFRC Small-Packet variant

RFC4828 defines TCP-Friendly Rate Control: The Small-Packet (SP) Variant (TFRC-SP), a modification of TFRC for applications that send small packets, where the design goal is to achieve the same bandwidth as a TCP stream sending packets of size MSS. RFC4828 specifies that a minimum ITT of 10 ms should be enforced to prevent a stream from sending too many small packets. This limit is based on the requirement of VoIP applications, that do not send packets more often than 10 ms. Also, this limit is justified by the lack of applications used in the current Internet that require to send small packets more often. To accommodate applications sending time-dependent data, TFRC-SP proposes a set of adjustments that will help streams with thin-stream characteristics.

Loss Event Rate

TFRC competes with TCP by counting the loss event rate, where one loss event is one or more lost packets within a window of data. A thin stream that sends multiple smaller packets, competing with a standard TCP stream (sending with the same bit rate, but with fewer MSS sized packets), can no longer achieve fairness with the TCP stream by counting the loss event rate as defined in RFC5348.

The loss measurement for a thin stream must be able to detect when a stream consistently loses multiple packets within an RTT. Therefore, TFRC-SP requires that, for loss intervals that are longer than two RTTs, only one loss event is counted. For shorter intervals, the lost or marked packets are accounted for by calculating the interval length as \(N/K\), where \(N\) is the number of packets in the interval, and \(K\) is the number of loss events in the interval.
4.4. RDB Congestion control background

Taking packet headers into account

By sending multiple smaller packets instead of fewer larger ones, a stream will cause extra load on the nodes along the network path. To account for this, TFRC-SP proposes to include the packet header size when calculating the throughput, as shown in equation 4.3.

\[ X := \frac{X \times s_{true}}{s_{true} + H}, \]  

(4.3)

\(X\) denotes the send rate, \(s_{true}\) is the average packet payload size, and \(H\) is the expected per-packet header size. Unless the exact header size is easily available, a default size of 40 bytes can be used for both IPv4 and IPv6, which is justified as a rough fairness is sufficient.

4.4.1.5 TFRC-SP simulations

Before implementing the TFRC-SP concept into the CC of RDBv2, we decided to do some simple simulations to get an idea of how (well) it works. We made a simple python script that implements the throughput calculation from equation 2.2 (as seen in code listing B.11), as well as the calculation of the average loss interval from pseudo code 4.3 (as seen in code listing B.12).

The simulations were performed with 20 individual streams, with the loss rates 0.1%, 0.2%, 0.5%, 1%, 2%, and 5%. For comparison, we ran testbed experiments with uniform loss induced with netem, with the same loss rates as the simulations. We ran the testbed experiments for 5 minutes, with the same number of simultaneous streams, using an RTT of 150 ms, which is also used in the experiments detailed in section 5.4. How the testbed experiments are setup and run is explained in detail in chapter 5. To generate the thin streams, we used Streamzero with an ITT of 50 ms, which should give each stream around 3 PIFs (150/50).

Simulation Results

From the results of the testbed experiments shown in figure 4.7, we see a clear trend where the CWND stabilizes at a low value around 2-4 as the loss rates grow. This illustrates the problem described in section 3.4 well, where the CWND does not grow more than what the requested send rate requires.

The results from the TFRC simulations in figure 4.8 show a very different picture compared to the testbed experiments of TCP. While the simulation with the highest loss rate of 5% shows that the streams are granted a low CWND, equal to the CWND values of the testbed experiments, the simulations with lower loss rates show that the CWND steadily increases as loss rates decrease. This is exactly what we want for the thin streams - a heavy reduction of the CWND when high loss rates are encountered, and a less aggressive reduction on lower loss rates.
Figure 4.7: Congestion window growth for 20 thin streams with TCP Cubic using default kernel settings
Figure 4.8: Congestion window simulation for 20 thin streams using TFRC-SP
4.4.2 Main tasks of the RDB congestion control

The RDBv2-CC aims to solve the following tasks:

- **Deal with the “hidden loss events” issue** caused by RDB described in section 3.2.1. We handle this issue by using the technique described in section 4.2.0.2 for detecting the hidden loss. The detected loss events are recorded in the TFRC loss history described in section 4.4.1.1, and used in the loss event rate calculation described in section 4.4.1.2.

- **Alleviate the issue of CWND growth for thin streams** described in section 3.4. This is solved by using the implementation of the function tcp_is_cwnd_limited from Linux kernel version 3.15, which is less strict than the version implemented in version 3.16.

- **Alleviate the issue of aggressive CWND reduction harming thin streams** described in section 3.4. We try to solve this by adjusting the CWND based on the TFRC throughput calculation designed to produce a more steady throughput rate. On every loss event, the CWND is adjusted according to the send rate calculated by equation 4.2.

- **Ensure that users do not have incentives for misusing RDB** by choosing RDB instead of standard TCP, for the purpose of increasing throughput at the expense of competing traffic. To make RDBv2 less attractive for throughput hungry streams, the CWND growth rate is reduced by lowering the slow-start threshold compared to CCs like TCP Cubic and TCP New Reno.

- **Prevent abuse of RDB** as described in section 3.2.2. This issue is dealt with by limiting bundling to a maximum number of PIFs similarly to the limitation imposed for the mechanisms mFR and LT. However, instead of a SPIFL of maximum 3 used by tcp_stream_is_thin (code listing 2.1), a DPIFL is used, which is a PIF limit dynamically calculated using equation 4.1. While the RDBv2 implementation can set the ITT_{min} manually, the default value is set to 10 ms, which is the minimum allowed ITT for a stream using TFRC-SP (RFC4828).

4.5 RDB-CC Implementation

RDBv2 has been implemented as a pluggable congestion control module. As the implementation spans multiple files, they have been placed in a new subdirectory net/ipv4/rdb/. The implementation of TFRC-specific functionality has been placed in net/ipv4/rdb/lib/.

4.5.1 Kernel Module

RDBv1 was for the most part implemented in the files tcp.c, tcp_input.c and tcp_output.c in the Linux kernel source tree. After starting to implement RDBv2, we eventually found the need to modify parts of the CC. We decided to
implement RDBv2 entirely as a kernel module, and only modify the kernel base files with the necessary changes for the entry points needed to execute the RDB code.

A major advantage of implementing RDB as a kernel module is that the code can be re-compiled much quicker without having to recompile the rest of the kernel. Another advantage is that the changes to the code can be loaded into a running kernel without rebooting.

The RDB implementation we present here, which we refer to as RDBv2, has been implemented as a kernel module configurable through the Linux kernel configuration, as shown by figure 4.9. In the figure, RDB is configured as a module, but it can also be compiled as part of the kernel, indicated by a star, like CUBIC TCP.

### 4.5.2 Congestion Control framework

The Linux CC framework is provided by the Linux kernel for implementing CCs as pluggable kernel modules. This makes it possible to easily separate the code for each CC implementation from the core of the TCP engine.

Code listing B.13 shows the entry points defined by the Linux CC framework, where sshthresh and cong_avoid are the only functions that the CC modules are required to implement (Code listing B.13, 11 and line 15).
4.5.3 RDBv2 implementation overview

The RDBv2-CC performs the following tasks:

- **Loss detection** based on processing incoming ACKs.
- **Congestion avoidance** performed through the cong_avoid hook.
- **ssthresh adjustment** after a CWND reduction.

The RDBv2 implementation is split across the following files:

- **rdb.c**
  Contains the initialization code for kernel module.

- **rdb_cc.c**
  Contains the CC code.

- **rdb_skb.c**
  Implements the functionality that tests if it is possible to bundle data in a SKB, and the functions to allocate and copy the redundant data into a new SKB.

- **lib/rdb_tfrc.c**
  This is the entry point for the TFRC-SP functionality utilized by the RDBv2-CC in rdb_cc.c.

- **lib/loss_history.c**
  This is where the functionality for keeping track of the loss event history, and the calculation of the loss event rate.

- **lib/tfrc_equation.c**
  This contains the function tfrc_calc_x that implements the throughput calculation from equation 2.2. Functionally, this is a direct copy of the file net/dccp/ccids/lib/tfrc_equation.c.

Figure 4.10 shows how the TCP engine calls the functions in RDBv2, through the Linux CC framework in code listing B.13. The code that performs the data bundling, by modifying the SKBs, is called from do_rdb through the entry point pkt_send. The other three functions, tcp_rdb_event, tcp_rdbcong_avoid_tfrc, and tcp_rdb_ssthresh, are part of the RDBv2-CC.
4.5. RDB-CC Implementation

Figure 4.10: Call graph showing the relations between the TCP engine, the Linux CC framework, and the RDB module. The node containing a star indicates a call to a function that further calls the functions pointed to by the star node.
Figure 4.11: Callgraph for RDBv2
4.5. RDB-CC Implementation

4.5.3.1 Processing ACKs

To calculate packet loss, the RDBv2 must analyse the state after each incoming ACK. While there is a specific hook for ack accounting pkts_acked, this hook is triggered during the cleanup process of the TCP output queue, after modifying the socket info. However, the hook cwnd_event (Code listing B.13 line 19), provides the necessary entry points we need. A requirement for RDB ACK-analysis is that it must be performed before the default ACK handling mechanism (tcp_ack) cleans up the TCP output queue.

Code listing B.14 shows an excerpt from the function tcp_ack, where we see the two events, CA_EVENT_FAST_ACK (Line 11) and CA_EVENT_SLOW_ACK (Line 15), being triggered, which in turn call cwnd_event.

Detecting loss

The function rdb_ack (see code listing B.15) is called for each incoming ACK before the ACK has been processed by the TCP engine. It calls the function rdb_check_rtx_queue_acked (Code listing B.16) which traverses the TCP output queue to find how many packet were ACKed.

If any packets are detected as lost, the counter rdb_ack_loss_count is incremented with the lost packet count. If TFRC is enabled, it further calls the function rdb_tfrc_update_loss_history, which updates the loss interval with the new information.

Of the loss detection techniques described in section 4.2, only the simplest one has been implemented in RDBv2, where loss is detected by ACKs that cover multiple packets. This detection technique is sufficient for our testbed experiments, because we can be certain that no packet reordering occurs, and no ACK packets can be lost on the reverse path. To limit the scope of the thesis, we therefore leave investigating this further as future work.

Performing TFRC calculations

After processing each incoming ACK, rdb_ack calls the function rdb_tfrc_update_loss_history (Code listing B.15 line 19), which updates the loss history (described in section 4.4.1.1). If loss is detected, the function rdb_tfrc_handle_loss is called to update the loss history, before the average loss interval is recalculated with tfrc_lh_calc_i_mean_sp (Code listing B.17 line 13). If no loss is detected, the current (open) loss interval is updated with rdb_tfrc_lh_update_i_mean.

Further, rdb_tfrc_update_loss_history will call the function tfrc_calc_x (Code listing B.17 line 27) which is the implementation of the throughput calculation from equation 2.2. Finally, on line 36 in code listing B.17, rdb_tfrc_update_loss_history calls the function rdb_tx_update_x (shown in code listing B.18), which performs the task of calculating the new CWND value based on the send rate, according to equation 4.2 (as described in section 4.4.1.3).
4.5.3.2 Congestion avoidance

The CC hook cong_avoid is called by the TCP engine to grow the CWND. RDBv2 has two implementations, where tcp_rdbcong_avoid_tfrc is used when the TFRC loss response mechanism is enabled, and tcp_rdbcong_avoid is used otherwise.

Both the implementations will perform the same basic tasks:

• **Update minimum RTT** (of the entire duration of the connection) if latest RTTM is smaller. The minimum RTT saved in cong->minrtt_us and used by tcp_stream_is_thin_dpif.

• **Restrict the CWND** if loss has been registered. When the ACK processing performed by rdb_ack detects a loss, the CWND will be updated. If RDBv2-CC is used with TFRC enabled, the CWND is updated with the value calculated from the TFRC-SP throughput. If TFRC is not enabled, it will instead call the function tcp_enter_cwr. This function is used by the TCP engine to enter CWND reduction state.

• **Increase the CWND** if no loss has occurred. The CWND is increased using tcp_reno_cong_avoid, the cong_avoid handler for TCP New Reno.

Due to the negative effects caused by the changes to the function tcp_is_mCwnd_limited in Linux kernel version 3.16 and forward, we decided to use the implementation from Linux kernel version 3.15 in the functions called by the cong_avoid CC hook (see code listing B.2).

4.5.3.3 Modify slow-start threshold

The Linux CC framework defines the entry point ssthresh (Code listing B.13 line 11), which is used by the TCP engine to set the slow-start threshold. The ssthresh hook implementation for TCP New Reno, tcp_reno_ssthresh (code listing B.22), sets the ssthresh to half the current CWND. To make RDBv2 less favorable for greedy streams, the rate at which the CWND grows could be reduced further compared to standard TCP.

An alternative implementation of the ssthresh entry point is shown in code listing B.23, where the ssthresh is set to 1/4 of the CWND, instead of 1/2. This is the same as used by TCP Nice, which is a CC designed to be less aggressive (*nicer*) than standard TCP, and is meant for tasks such as background transfers (*Mcdonald and Nelson [2006, a3]*).

4.6 Summary

We have described the background on why we needed to develop the new RDB prototype implementation RDBv2, and how we chose to do it. RDBv1 was a modification to the core parts of the TCP engine, that implemented redundant bundling
and sysctl variables for restricting how much bundling to perform. By taking a different route, and developing RDBv2 as a CC module in the Linux CC framework, the RDB mechanism can be controlled more precisely by the custom CC algorithm we have implemented, referred to as RDBv2-CC. The main improvements RDBv2 brings over RDBv1 are:

- **DPIFL** is a dynamically adjusted PIFS limit, that is calculated based on the connection’s RTT and the maximum allowed ITT specified for the connection. This enables a more precise control of which streams are classified as thin streams, and hence, which should be treated differently. This is achieved by basing the classification on the sender application’s transmission patterns, instead of using a SPIFL that will differentiate between connections with different RTTs.

- **Loss detection** based on processing incoming ACKs. This solves the problem caused by RDB, where loss events are hidden from the TCP engine.

- **The congestion control mechanism RDBv2-CC**, which is based on TFRC-SP. Based on the loss calculations, it calculates a throughput rate that oscillates less compared to standard TCP mechanisms that utilize the AIMD paradigm. This helps thin streams to maintain a higher CWND than regular TCP, while still trying to be TCP friendly.

In chapter 5, we evaluate the RDBv2 implementation that we have presented in this chapter.
Chapter 5

Evaluation of RDB prototype v2

In this chapter we will present how we have set up the experiments, and how we have evaluated the results. We will first describe the metrics we have used for measuring the effects of the mechanisms we have developed. We will further describe the testbed and the test environment we have used, before we give some details on the tools we have developed and used to produce and analyse the results. Finally, we will present the experiments we have performed, and describe some key results from the experiment tests.

5.1 Metrics for evaluating RDB

We define three metrics to measure the effect of RDB. ACK latency, TCP-friendliness and resource penalty. The ACK latency quantifies the gain in latency from the viewpoint of the sender. The TCP-friendliness quantifies the penalty incurred on competing traffic by the RDB mechanism, and resource penalty quantifies the resource overhead of the modified protocol. The sections that follow will go into detail about the metrics.

5.1.1 Latency

As the main goal of RDB is to improve the latency for thin streams, the main metric we use for the evaluations is latency. The end goal is to improve the application layer latency, which is the time between a segment is sent from the application layer on the sender hosts, till the application layer on the receiver host can read the data segment. However, as we currently do not have a good way of measuring the application layer latency directly, we measure ACK latency instead.

5.1.1.1 ACK Latency

The ACK latency is the measurement of the time when the sender host sends the data for the first time, till the sender host receives an ACK for the data. This is measured by analyzing the tcpdump captures of the network data on the sender side. When packets are lost, the in-order guarantee of TCP causes any data with
a higher sequence number to be delayed until the lost data has been retransmitted. By measuring when the data is ACKed, we include this extra delay in the latency numbers. The ACK latency calculations are performed with Analysetcp (described in section 5.3.4).

5.1.1.2 ACK Latency vs Application layer latency

The ACK latency resembles the application layer latency, but they differ in a few aspects. The application layer latency is the measurement from the time when the data is passed from the user space application to the kernel on the sender side, to the moment the data is passed from the kernel to the user space application on the receiver side.

The ACK latency differs, by measuring the time between when the data segment is first sent onto the network by the sender host, till the sender receives an ACK for that segment. While this may sound like very different measures, they effectively provide the same measurement data, with one key difference, namely the send buffering delay.

The ACK latency is calculated based on the data captures provided by tcpdump, which is fetched from the packets in the link layer on the sender host. Therefore, any processing delays in the higher layers of the sender host network stack are not included in these numbers.

When data is received in order, nothing but processing delay should delay the deliverance of the data to the application layer. Therefore, the application layer latency and ACK latency measure the same metric on the receiver side. However, on the sender side they can differ significantly. This is because the data can be delayed in the TCP output queue before it is passed to the lower layers. As long as Nagle’s Algorithm is not disabled, it will cause delays for send calls with small data chunks. Another reason for delays is that the CWND prevents more data from being sent. This queuing delay is not reflected in the ACK latency data, but is an important part of the application layer latency measure.

For streams that send packets very seldom, this time difference may be negligible, as long as they are not limited by the CWND (and Nagle’s Algorithm is disabled), but as soon as the sender starts to buffer data in the TCP output queue, this delay can be significant.

We use the ACK latency because that is the most accurate metric that measures the per-data-chunk in order delivery latency that is possible to calculate with the data provided from the tcpdump output traces. To measure the application layer latency, we could use hooks in the send and read system calls to get the correct time stamps for when the data is sent and delivered. This would however require further work, and is out of the scope for this thesis.

5.1.1.3 Evaluating gains in latency

As we do not have tools for measuring the application layer latency, we need to compare the ACK latency values in the context of how the ITT values change. If an experiment shows a decrease in the ACK latency values, it is also necessary to take into account changes to the send buffering delay, which we indirectly can see
from the ITT numbers. As we know that the user space application for each stream in two similar tests will send the same amount of data to the kernel, we know that if one test has lower average ITT, and consequently larger average payload per packet, it means the send buffering delay was higher.

5.1.2 TCP-friendliness

TCP-friendliness, in this context, is how the RDB modifications affect the behaviour the streams, in regards to fairness towards competing streams using standard TCP. There are many aspects to consider in this regard.

As described in section 2.5.3, the traditional definition of a TCP-friendly network stream is one that does not outperform a TCP stream in terms of throughput. While this makes sense when the only goal is to obtain a fair bandwidth share, we must also consider the effect RDB has on the latency of competing TCP thin streams. However, it is also important to evaluate if the RDB mechanism behaves in a TCP-friendly manner in regards to throughput. The experiments presented in section 5.4.5 will therefore mainly consider the traditional throughput-based measure of TCP-friendliness.

5.1.3 Resources

By bundling extra data with the packets, RDB causes extra overhead on the network nodes that processes the packets. In scenarios where thin streams are limited by the CWND, the data gets buffered in the sender’s TCP output queue causing larger and fewer packets to be sent. By using RDBv2-CC, the CWND will not so easily become the limiting factor, enabling the sender host to transmit the data immediately, meaning more packets gets sent over the network. This also causes extra overhead for the network nodes.

5.1.3.1 Calculating the cost of sending network data

To calculate the cost of sending a network packet we separate the processing tasks for an application $a$ into:

- **Per-packet processing cost** $\alpha_a$ which is the processing cost for each packet independent of its size.

- **Per-byte processing cost** $\beta_a$ which increases linearly for each byte in the packet.

From Ramaswamy, Weng, and Wolf [2004, a38] we have the following equation:

$$c_{a,l} = \alpha_a + \beta_a \times l$$

(5.1)
Calculating the number of CPU instructions required to send a network message from user space is not easy, as it depends on the type of message and the kernel implementation. Protocols such as TCP, that provide services like reliable transmission through data checksums and in-order guarantee using sequence numbers, will require more processing than simpler protocols like UDP.

Measurements have shown that 12,000 instructions is a fair estimate (Gray [1988, a39]). According to Gray [2000, a40] a rule of thumb for calculating the cost of sending a message is 10,000 instructions per packet plus 10 instructions per byte, giving the following calculation (by equation 5.1) for a packet of 100 bytes:

\[
\alpha_a = 10,000\text{ins}, \ \beta_a = 10\text{ins}, \ \ l = 100 \\
11000\text{ins} = 10000 + 10 \times 100
\]  

In the case of TCP, the send call performed by the application leads to a context switch into kernel mode, where the data is copied to the TCP output queue, either by utilizing the free space of an existing SKB, or by creating a new SKB. Protocol specific operations are also performed, such as calculating the sequence numbers and updating the socket struct.

A large part of the per-message cost is not strictly the cost of sending a network packet, but the entire sequence of events triggered by the call to send from user space. Therefore we can expect that sending 1000 bytes by issuing 1000 calls to send, with 1 byte for each call, will cost much more than doing 1 call with 1000 bytes. If Nagle’s Algorithm is enabled, we can expect only 1 packet to be sent in both cases, but 1000 context switches, plus the instructions necessary to add each byte to the TCP output queue, will certainly be more costly in respect to CPU instructions.

With the introduction of different offloading schemes in the Network interface cards (NICs), like segmentation offloading and checksum offloading, the number of required CPU instructions are reduced significantly. In the Linux kernel, a NIC driver can signal that it supports e.g. TCP Segmentation Offload (TSO) by setting the netif_f_tso bit in the netdev_features_t struct in the socket struct. This requires that it supports TCP checksum offloading as well as Scatter-Gather (SG), which is the ability of the NIC to produce TCP packets from a list of segmented memory locations. Instead of the kernel segmenting the data from user space into MSS sized segments placed in separate SKBs, this is left for the NIC to do. The TCP output engine will instead add the data to the memory pages of the newest SKB in the TCP output queue, as long as the data in this SKB has not been sent. The total amount of data in these fragmented memory areas pointed to by one SKB may far exceed the MSS.

The SKBs have two variables, nr_frags and frags, where nr_frags is the number of fragments referred to in the frags array. Each element in this array, is a skb_frag_struct containing a reference to a memory page fragment struct and its length. When the NIC receives the SKB from the kernel, it will read the data from the memory pages, split the data into MSS sized segments, and create
Figure 5.1: Packet processing time in a network router (Ramaswamy, Weng, and Wolf [2004, a38])

TCP packets. The reason the NIC must also support TCP checksum offloading in hardware is that if the TCP checksum is to be computed by the CPU, it is most efficient to combine the checksum operation with copying the data into the packet. However, segmentation offloading has been shown to be useful only for transfers of large amounts of data Foong et al. [2003, a41].

Figure 5.1 shows the measurements of processing time in a router for IP packet forwarding and Internet Protocol Security (IPSec) packets passing through an IPSec tunnel. The forwarded IP packets require no processing of the payload, while the data in the IPSec packets need cryptographic processing. From the plot it is clear that the weights for $\alpha_a$ and $\beta_a$ differ greatly between the two applications, where the per-byte processing cost ($\beta_a$) is more dominating for the IPSec packets than for the IP packets. One might expect the $\beta_a$ for the IP packets to be zero, but even though the packet payload is not processed, it must be copied from the input queue to the output queue, and the cost of copying these bytes depends on the number of bytes.

5.1.3.2 Overhead of RDB

While RDB requires extra procedures to copy the redundant data into the packet to be sent, the CPU workload of transferring the data from user space into the TCP output queue is the same. What may require extra CPU instructions is the RDBv2-CC that is required to traverse the TCP output queue regularly to perform
loss detection, as well as the send rate calculation explained in section 4.4.1.3. The send rate calculation requires the continuous recalculation of the loss event rate, followed by the throughput calculation.

Even if RDB is a sender side modification, it may affect the receiver side’s performance. With a regular TCP network stream, the receiver side will usually receive each packet in the correct order. Therefore, the receiver can expect that the sequence number of the next packet, to be equal to the sequence number of the last byte of the last data segment received, plus one. This will be the case unless packets are lost, or reordering occurs.

With RDB, this changes. As the first part of the data in RDB packets is old, the sequence number of the data will not be the expected sequence number, unless packets are lost, or reordering occurs. The procedure that handles incoming packets in the TCP engine (tcp_data_queue), fails on the first three if-tests, before passing on the last if-test (see code listing 2.3 line 24). If the DSACK options is enabled, the receiver will also add the sequence range of the redundant data to the SACK option (see code listing 2.3 line 28). This causes extra packet processing and function calls compared to packets from a regular TCP stream.

Measuring the approximate overhead of using RDB could be done on a theoretical level, e.g. by using the amount of extra data RDB produces as input to equation 5.1. However, as the experiments show, it is often difficult to separate the effect of the bundling itself, and effects caused by how the behavior of the CC changes as a result of the RDB data. If RDB, together with the RDBv2-CC, cause more packets to be sent in total, the calculation must be adjusted to take the extra number of packets into account.

As we will see in some of the experiments, the RDB streams cause more loss for other thin streams in some scenarios. Some of the results show that the TCP thin streams end up sending fewer packets in total, with more payload per packet. This will actually cause less workload for the network nodes, as fewer packets have to transferred through the network.

Is is made even more complex due to the fact that the bundling rate depends on many different criteria, such as the PIF limitation, the maximum limit of the sysctl variables described in section 4.3.2, the size of the data segments the application produces, and the interval between each send call. In addition comes the network characteristics, which will affect how easily the stream becomes network limited.

5.1.3.3 RDBv2 resource usage

To ensure minimum resource overhead for regular TCP streams that use a CC that does not implement the pkt_send entry point, the if-tests that test for pkt_send uses the keywords unlikely and likely. Normally, the compiler uses branch-prediction to optimize the machine code for the most likely branch path (if-else). These keywords are instructions that tell the compiler to optimize for a certain branch, causing the machine code to be optimized for branches specified using likely, and not for those specified with unlikely. By using unlikely when testing if pkt_send is defined, we ensure that the most probable branch (where it is not defined), is the most efficient. However, when pkt_send is defined, the call to pkt_send will probably succeed, hence, we surround the call with likely.
5.2. Test environment

Simulation is a common way to perform experiments in network research. The ns–2 simulator created in the late 1990s, was for many years the de-facto standard tool used within academic network research (Riley and Henderson [2010, a42]). Today, there are multiple alternatives, among them ns–3, which set out to replace ns–2. ns–2, however, still has by far the richest TCP functionality (Rosbach [2012, a43]).

For the purpose of the experiments presented in this thesis, simulation is not a practical choice, as that would require implementing the modifications in the language of a simulator such as ns–2 in addition to the implementation in the Linux kernel.

To run tests that produce statistically significant results we need the tests to generate a certain amount of data. We may either have the tests last long enough to produce enough data, or make the tests produce data faster. By running multiple similar network streams, and aggregating the results, we are able to produce data faster.

5.2.1 Testbed setup

We have performed all the experiments in a lab testbed using Linux Debian 7. The testbed is set up with three sender hosts and one receiver host, with two bridges between them, as depicted in figure 5.2.

The two hosts used to send thin streams, rdbsender and wsender, are running Linux kernel version 3.16, and the host that sends greedy streams, zsender, is running Linux kernel version 3.12. The bridges, bridge1 and bridge2, are set up with rate control and network delay, respectively. Rate control is set up as depicted in code listing 5.1, and network delay is set up with netem as in code listing 5.2.

According to Bufferbloat-project [2014, b11], it is best to configure netem on a separate host, which is why we have used separate hosts for netem and rate control. We have also followed the guidelines of disabling any offloading mechanisms on the NICs.

<table>
<thead>
<tr>
<th>Testbed configuration code</th>
</tr>
</thead>
<tbody>
<tr>
<td>tc qdisc del dev eth1 root</td>
</tr>
<tr>
<td>tc qdisc add dev eth1 root handle 1: htb default 10</td>
</tr>
<tr>
<td>tc class add dev eth1 parent 1: classid 1:10 htb rate 5000kbit</td>
</tr>
<tr>
<td>tc qdisc add dev eth1 parent 1:10 pfifo limit %BDP/s</td>
</tr>
</tbody>
</table>

Commands used to set up rate control on bridge1. %BDP/s is replaced with the calculated BDP.

Code Listing 5.1: Setup of rate control with htb qdisc
Figure 5.2: Testbed network setup

Testbed configuration code

```
tc qdisc add dev eth1 handle 1:0 root netem %(delay)s
.tc qdisc add dev eth2 handle 1:1 root netem %(delay)s %(loss)s
```

Commands used to set up network delay bridge2. %(delay)s and %(loss)s is replaced with the values for the specific test being run.

Code Listing 5.2: Setup of network delay with netem

5.2.2 Challenges and pitfalls in testbed experiments

Choosing network parameters, such as rate limit on the bottleneck and the RTT for the test machines, is a much more complex topic than one might expect. There are a multitude of variables that can affect the test results in unexpected ways, such as the configuration of the end hosts, the hardware of the NICs used (or the driver version), the router/switch hardware, and configuration of the bottleneck node.

There are many pitfalls when configuring the network nodes, such as the effect the clock interrupt rate of a hosts kernel may have on the behavior of buffer queues, and how configuring network emulation together with rate control may give unreliable results ([Bufferbloat-project](https://github.com/bufferbloat-project/bufferbloat-project) [2014, b11]).

*Bufferbloat-project* [2014, b11] advises to remove as much (potential) buffer bloat as possible, such as caused by offloading mechanisms in the kernel. We performed tests with different offloading mechanisms enabled and found there were...
differences, but we had a hard time figuring out a system behind the different results. We ended up following the advice of disabling the offloading mechanisms on all hosts in the tested. These are a) Generic Segmentation Offload (GSO) b) TCP Segmentation Offload (TSO) c) Generic Receive Offload (GRO), and d) Large Receive Offload (LRO)

5.2.3 Rate control

We have experimented with different parameters before finding a setup that works for all the different test setups we have used. The final setup is the result of a wide range of tests where we, at the end, believed we could trust the results to be correct.

5.2.3.1 Bufferbloat

The rate limit set for the bottleneck node will control the amount of data being sent through the node within a certain time. This however, brings up the question of buffer queue lengths and how that affects the network streams. The problem with traditional router queues is that different kinds of network streams share the same buffer. While queues are important to handle packet bursts, they also cause increased delays for delay-sensitive traffic. If the queue is too short, the node is unable to properly deal with fluctuations in the arrival rate caused by bursty traffic, but if they are too big they cause the buffers to stay consistently full, a problem known as bufferbloat (Nichols and Jacobson [2012, a44]). Even though the problem was identified decades ago, this issue is still relevant, and getting even more relevant due to today’s cheaper memory leading to more available memory for queues in the routers, the increase in applications that need low latency, and the increase in bandwidth-hungry applications such as video streaming.

While there is no “correct” queue length, as it all depends on the network properties, and what you want to achieve with the queue, a standard approach is to base the queue length on the BDP (Vu-Brugier et al. [2007, a45]), which is what we have based our tests on.

5.2.3.2 Finding a rate limit setup

By basing the queue length on the bottleneck on the BDP as calculated by equation 5.3, we found it necessary to set the rate limit to a value that supports the number of simultaneous streams we want to test. In the latency tests (experiments 2 and 3) we restricted the tests to run up to $16 \times 2 + 5$ simultaneous streams, and in the fairness tests (experiment 4), up to $13 \times 3$ streams.

When first testing with a rate limit of 1000kbit/s, where the BDP suggests a queue size of 13 packets, we experienced unreliable results in some of our tests. When comparing the results of a test with 5 greedy streams and 10 thin streams versus a test with 5 greedy streams and 20 thin streams, the difference in loss rates for the thin streams could be significant. We had expected the results to be fairly similar as we presumed the greedy streams would simply get a smaller share of the total bandwidth, however, that was not the case. With multiple greedy streams constantly trying to send at maximum rate, they continually have packets
in the queue on the bottleneck. In our test setup, there is a direct link between the sender hosts and the bottleneck (bridge), which means that the ability to absorb the packets depends solely on that one host. The thin streams are dependent on not arriving when the queue is full - the same queue the greedy streams are doing their best to fill up. With the queue being heavily used by greedy streams, the capacity for thin streams is not excessive, which explains why a doubling of the thin streams count would lead to unreliable results. The problem is not first and foremost the bandwidth limit being too small, but the small queue length being unable to handle that many network streams.

\[
BDP = \frac{kbit\_lim * 1000}{8} \times (RTT \times 10^{-3})
\]

\[
QLEN = \frac{BDP}{MTU}
\]

(5.3)

5.2.4 Network properties and stream characteristics

Testing thin streams with different characteristics on different types of network environments is important to find where RDB performs best, and finding the corner cases where it does not perform as well. The stream characteristics we have chosen are based on the typical streams observed by applications that produce thin streams (see table 2.1).

Different network environments will affect how RDB behaves. The two most important properties are the ITT in relation to the RTT, and the loss rates. For RDB, the ITT in relation to the RTT controls how much can be bundled. Therefore running tests where only the ITT is varied would in theory produce the results we are after. However, there is a problem with that theory, and that is the RTO timer, and how its minimum value is controlled by the static value TCP_RTO_MIN. With the default setting in the Linux kernel of 200, the RTO timer will never be less than 200 ms, independent of the RTT. On a connection with a low RTT, such as 10 ms, the RTO timer will therefore be very large relative to the RTT, compared to a connection with an RTT of 150 ms.

We have found it necessary to reduce the number of variables as much as possible to make the amount of test results manageable. We have therefore used an RTT of 150 ms for all the experiments, a value which gives room for a wide range of ITTs, and is is shown to be a good approximation of common RTTs in the Internet based on website RTT measurements (Markussen [2014, a34]).

To be able to run tests with a wide range of simultaneous streams, we decided on a rate limit of 5000 kbit/s, which gives a pfifo queue length of 63 packets. However, due to rounding errors in our script, the value used for the tests in experiment 1 is 60.
5.3. Tools

We have used a wide range of tools to perform the experiments and analyse the results. The tools we have developed in-house are explained briefly.

5.3.1 sshscheduler

sshscheduler is a python script we developed to run the experiments in the testbed. It is designed for easily generating a large number of jobs with different test properties. Implementing strict return value verification, and full logs for every command, makes it easy to identify errors and misconfigurations in the tests.

5.3.2 graph_r

graph_r is a collection of python scripts we have developed to analyse data and generate plots based on the results of the testbed experiments. By providing a directory containing pcap files from a test run, it will analyse the file names and extract the test parameters. Depending on which types of results is requested, it will run a set of commands to produce different types of results, like ACK latency, loss statistics, throughput, goodput, and CWND. Further, the results are plotted with R using the python wrapper rpy2.

5.3.3 streamzero

We have used Streamzero to generate network streams, a program developed specifically for the task of testing thin streams. It can set up multiple network streams that send data simultaneously, using either TCP or UDP. It is similar to iperf, but has a few extra features implemented for our use cases. Important features that we have used when testing:

- **Send streams with defined packet size/ITT or bandwidth**
  When sending thin streams it does send calls to the kernel with small chunks of data to try to make the kernel send small packets, but it cannot guarantee small packets being sent, as the kernel may merge the data into bigger chunks depending on many factors and settings.

- **Specify randomly varying ITT and packet size with standard deviation.**
  This is important to avoid synchronous transmits when testing with multiple
streams per host. By specifying an ITT using the notation $X : Y$, the time between each data segment is sent to the kernel is randomly chosen from a normal distribution of mean $X$ and standard deviation $Y$. The random variation is computed using a Box–Muller transform of the pseudo-random normally distributed numbers produced by `drand48`, as shown in code listing 5.3.

- **Specifying socket options**
  - Enable `tcp_thin_linear_timeouts` and `tcp_thin_dupack`
  - Disable Nagle’s Algorithm
  - Enable RDB

Command example 5.1: Running streamzero

```
root@rdbsender:~$ ./streamzero_client -s 10.0.0.22 -p 6000 -v2 -A4 -W -j 3 -P 22000 -d 5m -i 10:1 -S 300 -c 3 -G rdb -r -B1500

Registered intervals:

Active streams: 1, completed 0, goodput: 0.00 bps (0.00 Bytes/s), time left: 00:04:59:999
Number of streams started: 3
Waiting for streams to finish.
Connection 22002 ->10.0.0.22:6000 starting up
Connection 22003 ->10.0.0.22:6000 starting up
Connection 22004 ->10.0.0.22:6000 starting up
Active streams: 3, completed 0, goodput: 277.20 Kbps (34.65 KBytes/s), time left: 00:04:47:987
Caught signal SIGINT
Waiting for threads to finish...
Waiting for 3 threads.
Done waiting for threads!

3 threads out of 3 targeted arrived: 3 completed successfully, 0 errors and 0 aborted.
Number of streams that timed out: 0
Done waiting for threads!

STREAM SESSION ENDED

Running time 00:00:15:102
Connections established: 3
Total bytes transferred: 483300 (483.30 KB)
Total packets sent: 1611 (Per stream: 537)
Average session goodput: 256.00 Kbps (32.00 KBytes/s)
Streams not finished cleanly: 0

root@rdbsender:~$ 
```
5.3.3.1 Why variate the ITT?

We noticed some strange results in our tests, regarding the loss rates of the individual streams from the hosts sending thin streams. For some tests the loss rates were varying to a great degree, where we expected them to be fairly equal. We tested with up to 40 simultaneous thin streams, and the more streams, the greater the variation of loss. After investigating the issue with many different tests, we found that the issue was caused by the thin streams sending data at regular intervals. When the total number of streams exceed a threshold, that depends on the bottleneck queue size, a certain number of streams will more often than others encounter a full queue, and hence lose the packet. When the ITT is not adjusted dynamically, the same streams will end up continually sending at a bad spot right after other thin streams have sent their packets.

To illustrate this, the results in subfigure 3.2.(e) (from section 3.1) have been plotted together with the results from rerunning the test using an ITT of 100 ms, instead of 100:10 ms as in the original tests. In these plots (figure 5.3) the latency is plotted for each stream, and not aggregated. Comparing the loss rates in the two the tables we see that the loss values of subfigure 5.3.(c) has a greater variation than subfigure 5.3.(d).

To avoid any synchronization issues with the thin streams we therefore decided to start the streams with a variable delay as well as using a random variation for the ITT of each thin stream.

---

**Source Code**

```c
int get_negexp_val(int mean, int stdev) {
    double rand = sqrt(-2 * log(drand48())) * cos(2 * M_PI * drand48());
    return rand * stdev + mean;
}
```

Code Listing 5.3: Function in `streamzero` that generates pseudo-random numbers from a given mean and standard deviation.
Figure 5.3: Results showing the difference and variation in ACK latency and loss between static (100) and dynamic (100:15) ITT
5.3.3.2 Contributions

As part of this thesis, Streamzero has been in large parts rewritten and extended with a set of features:

- **Stream intervals** which is a way to specify stream characteristics that last for a certain period. With $-I^d:10,i:300,b:2-I^d:2,i:50,S:300:50$, two intervals are specified where the first interval has a 10 seconds duration, ITT of 300 ms, and bandwidth of 2kbps, which means it will adjust the size of the payload argument to send to match this throughput. The second interval lasts for 2 seconds, has an ITT of 50 ms and an average payload size of 300 bytes with a standard deviation of 50. The specified intervals will be run in a round-robin fashion until the global duration expires.

- **Specifying a Congestion Control Algorithm** to use for the streams. This was a strict requirement after implementing RDBv2. The alternative is to set RDB as the default CC, but that would lead to all the TCP connections using RDB which easily causes trouble when there are bugs in the code.

- **Support for UDP traffic** instead of TCP.

- **Bug fixes and performance improvements** allowing one sender host to run thousands of simultaneous threads.

- **Verify the data integrity with SHA-1**

Implementing SHA-1 data verification between the server and client helped to identify problems during the development of RDBv2.

5.3.4 analyseTCP

Analysetcp is a tool developed by the authors of mFR, LT and RDBv1 for analyzing tcpdump trace (pcap) files. The most important feature of Analysetcp, that other tools do not provide, is analyzing both sender and receiver trace to calculate exact loss. Analysetcp can also calculate the variation of one-way latency for the packets by adjusting for clock drift on the two traces. It is also the only tool that can handle RDB packets, and differentiate between retransmitted and RDB data. The following are the most important features that we have used when analyzing the results:

- **Calculating ACK latency based on ACKs**

  The ACK latency, described in 5.1.1.1, is the main metric we have used to measure the latency differences for different TCP variations.

- **Loss estimation based on retransmissions**

  This is simply the ratio of the number of retransmissions compared to the number of packets sent. This gives a rough approximation of lost packets, which is fairly correct in many scenarios. In some scenarios, however, such as when using RDB, the approximation is not useful as RDB reduces the number of retransmissions drastically.
• Calculating exact packet and byte loss
  This has been one of the most important features we have used for this thesis. By comparing the data from the sender side pcap file to the data in the receiver file, the exact amount of lost packets and bytes is calculated. This has been very useful to get a precise overview of the network environment the network streams are tested in.

5.3.4.1 Contributions

As part of the work in this thesis, large parts of Analysetcp have been rewritten. A range of bugs have been implemented and fixed in the process of improving and stabilizing the code. At a certain point most traces were handled properly, and most new issues we encountered were caused by corner cases in the traces. After comparing the results from Analysetcp with tcptrace and wireshark/tshark, we are confident that it produces reliable results.

• Re-implementation of how the information for the data segments are stored has enabled a more accurate analysis. Every data segment is structured in a struct ByteRange that contains all the relevant information. By storing the relative sequence numbers (first packet starts with sequence number 0) we can more easily compare the output with wireshark when debugging. Following is a list of the variables from struct ByteRange and what feature they provide:

  • `packet_sent_count`, `packet_retrans_count` and `packet_received_count` giving a complete picture of the number of packets that were used to transfer this data.

  • `data_sent_count`, `data_retrans_count`, `data_received_count`, `rdbuf_count` providing the context for each transmit as well as how many times the data arrived on the receiver side.

  • `pcap timestamps for all the packets that sent this data segment` which makes it possible to use the correct timestamp when calculating the ACK latency. One might think that the correct timestamp is always from the last packet that sent the data, but the sender side may retransmit data even when it is not necessary, in which case using the timestamp from the last packet yields incorrect latencies.

  • `tcp timestamps for all the sent packets and the packet that was first received`. This makes it possible to identify exactly which of the sent packets were received.

  • `acked_sent`, `ack_count`, `dupack_count` to count the number of pure-ACKs sent for this sequence number, the number of times this sequence number was ACKed as well as the number of dupACKs.

• Calculating RDB “hits” and “misses”
  The ByteRange also has the variables `rdb_byte_hits` and `rdb_byte_miss` that are used to count the `hits` and `misses`. When RDB bundles data, `hits` is the measure of how many of the RDB packets that filled a hole on the data
sequence on the receiver side. *Misses* are the packets that contained bundled data which was already successfully delivered.

- **Ability to produce data from only parts of the trace**

- **Performance improvements**

  based on profiling with valgrind’s cache_grind tool as well as memory optimizations. The result is that Analysetcp, compared to earlier versions, performs much better both in terms of CPU time and memory usage, while producing more detailed data and statistics.

**Command example**

```
$ analysetcp -s 10.0.0.12 -r 10.0.0.22 -p 5000 -f /root/bendiko/pcap/
  rdbsender_wsender_zsender_to_zreceiver/
  RDBSENDER_WSENDER_ZSENDER_RDB_vs_GREEDY_TFRC5/all_results/r5-rdb-itt10:1-
p120-ccrdb-dab-110-er3-dp1f20-tfrc1_vs_w5-itt0:1-p120-ccubic_vs_z5
  ..kbit5000_min5_itt150_loss_pifo_qlen60_delayfixed_num0_rdbsender.pcap -g /
  root/bendiko/pcap/rdbsender_wsender_zsender_to_zreceiver/
  RDBSENDER_WSENDER_ZSENDER_RDB_vs_GREEDY_TFRC5/all_results/r5-rdb-itt10:1-
p120-ccrdb-dab-110-er3-dp1f20-tfrc1_vs_w5-itt0:1-p120-ccubic_vs_z5
  ..kbit5000_min5_itt150_loss_pifo_qlen60_delayfixed_num0_zreceiver.pcap -A

Processing sent packets...
Using filter: 'tcp && src host 10.0.0.12 && dst host 10.0.0.22 && dst port 5000'
Finished processing sent packets...
Processing receiver dump...
Using filter: 'tcp && src host 10.0.0.12 && dst host 10.0.0.22 && dst port 5000'
Finished processing receiver dump...

10.0.0.12-22000-10.0.0.22-5000 : Failed to find timestamp for 1 out of 13869 packets. These packets were at the end of the stream, so presumably they were just not caught by tcpdump.

10.0.0.12-22000-10.0.0.22-5000 : Failed to find timestamp for 1 out of 14090 packets. These packets were at the end of the stream, so presumably they were just not caught by tcpdump.

Aggregated Statistics for 5 connections:
Duration: 297 seconds (0.082508 hours)
  Total packets sent : 68152
  Total data packets sent : 68132
  Total pure acks (no payload) : 10
  SYN/FIN/RST packets sent : 7/5/0
  Number of retransmissions : 507
  Number of packets with bundled segments : 20538
  Number of packets with redundant data : 21043
  Number of received acks : 67143
  Total bytes sent (payload) : 36743144
  Number of unique bytes : 18538760
  Number of retransmitted bytes : 129984
  Redundant bytes (bytes already sent) : 18212384 (49.57 %)
  Estimated loss rate based on retransmissions : 0.74 %

Receiver side loss stats:
  Number of packets received : 67319
  Packets lost : 833
  Packet loss % : 1.22 %
  Bytes Lost (actual loss on receiver side) : 546960
  Bytes Loss : 1.49 %
  Ranges Lost (actual loss on receiver side) : 2049
  Ranges Loss : 1.53 %

Payload size stats:
  Average of all packets in all connections : 539
```
Chapter 5. Evaluation of RDB prototype v2

---

Average of the average for each connection: 550
Minimum payload (min, avg, max): 88, 100, 104 bytes
Average payload (min, avg, max): 442, 551, 653 bytes
Maximum payload (min, avg, max): 1448, 1448, 1448 bytes
Average for all packets in all conns: 0 ms

RDB stats:
- RDB packets: 20538 (30.14% of data packets sent)
- RDB bytes bundled: 3453280 (9.40% of total bytes sent)
- RDB packet hits: 329 (1.60% of RDB packets sent)
- RDB packet misses: 20209 (98.40% of RDB packets sent)
- RDB byte hits: 95232 (2.76% of RDB bytes, 0.26% of total bytes)
- RDB byte misses: 3358048 (97.24% of RDB bytes, 9.14% of total bytes)

Latency stats (Average for all the connections):
- Minimum latencies (min, avg, max): 150, 150, 150 ms
- Average latencies (min, avg, max): 702, 807, 907 ms
- Maximum latencies (min, avg, max): 1214, 2272, 4957 ms
- Average for all packets in all conns: 789 ms

ITT stats (Average for all the connections):
- Minimum ITT (min, avg, max): 0, 0, 0 ms
- Average ITT (min, avg, max): 17, 22, 29 ms
- Maximum ITT (min, avg, max): 843, 1341, 2703 ms
- Average for all packets in all conns: 21 ms

General info for entire dump:
10.0.0.12: -> 10.0.0.22:5000
Filename: /root/bendiko/pcap/rdbsender_wsender_zsender_to_zreceiver/RDBSENDER_WSENDER_ZSENDER_RDB_vs_GREEDY_TFRC5/all_results/r5-rdb-itt10:1-ps120-crdbr-dad-lt8-er3-dpif20-tfrc1_vs_w5-tcp-itt10:1-ps120-cubic_vs_z5...
kbit5000_min5_rtt15b_loss_pif0_qlen50_delayfixed_num0_rdbsender.pcap
Sent Packet Count: 68152
Received Packet Count: 67319
ACK Count: 67143
Sent Bytes Count: 36743144
Max payload size: 1448
Received Bytes: 36196184
Packets Lost: 833
Packet Loss: 1.22227 %
Ranges Count: 67637
Ranges Sent: 133960
Ranges Lost: 2049

Command example 5.2: Running analyseTCP
5.3. Tools

5.3.5 tcpproberdb

tcpproberdb is a kernel module we have written which is based on the tcpprobe module available in the Linux kernel. This tool uses kprobes to initiate a callback for each received ACK to print different properties of the socket, like CWND and PIFs.

kprobes enable code in the kernel to dynamically attach a callback routine to be called before entering a kernel function. This way of gathering the information should be more efficient than calling getsockopt with TCP_INFO argument from user space continually, and will also provide the information more consistently. The kprobe trap is guaranteed to be called for every call to the kernel function, while the getsockopt call will only return the information available at the time the user mode process is scheduled. Code listing 5.4 shows the script used before each test to load the module and capture the data. Example output from tcpproberdb can be seen in commands listing 5.3.

Source Code

<table>
<thead>
<tr>
<th>Source Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  #/bin/bash</td>
</tr>
<tr>
<td>2  PORT=${PORT:-0} # Default 0 == All</td>
</tr>
<tr>
<td>3  FULL=${FULL:-1} # On every ACK</td>
</tr>
<tr>
<td>4  FLUSH=${FLUSH:-1} # Flush on every packet</td>
</tr>
<tr>
<td>5  cd /root/tcp_probe_rdb</td>
</tr>
<tr>
<td>6  make reload V=1 PORT=${PORT} FLUSH=${FLUSH} FULL=${FULL}</td>
</tr>
<tr>
<td>7  cat /proc/net/tcpproberdb &gt; /root/tcpproberdb.out</td>
</tr>
</tbody>
</table>

The bash script used to fetch information of the TCP socket

Code Listing 5.4: Bash script used to run tcpproberdb

Command example

<table>
<thead>
<tr>
<th>Command example</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.874426735 10.0.0.12:22018 10.0.0.22:5000 32 3444766184 3444765944 10</td>
</tr>
<tr>
<td>2147483647 5792 302 29312 604 506 2 0 0</td>
</tr>
<tr>
<td>28.174357481 10.0.0.12:22018 10.0.0.22:5000 32 3444766304 3444766064 10</td>
</tr>
<tr>
<td>2147483647 5888 283 29312 604 887 2 0 0</td>
</tr>
<tr>
<td>28.185576427 10.0.0.12:22019 10.0.0.22:5000 32 1150720319 1150720079 10</td>
</tr>
<tr>
<td>2147483647 5792 302 29312 604 506 2 0 0</td>
</tr>
<tr>
<td>28.2758249416 10.0.0.12:22018 10.0.0.22:5000 32 3444766424 3444766184 10</td>
</tr>
<tr>
<td>2147483647 5888 266 29312 604 870 2 0 0</td>
</tr>
<tr>
<td>28.285576217 10.0.0.12:22019 10.0.0.22:5000 32 1150720439 1150720199 10</td>
</tr>
<tr>
<td>2147483647 5888 283 29312 604 887 2 0 0</td>
</tr>
<tr>
<td>28.297799684 10.0.0.12:22020 10.0.0.22:5000 32 922659059 922658819 10</td>
</tr>
<tr>
<td>2147483647 5792 302 29312 604 506 2 0 0</td>
</tr>
<tr>
<td>28.393789342 10.0.0.12:22018 10.0.0.22:5000 32 3444765544 3444766304 10</td>
</tr>
<tr>
<td>2147483647 5888 252 29312 604 852 2 0 0</td>
</tr>
<tr>
<td>28.408576188 10.0.0.12:22019 10.0.0.22:5000 32 1150720559 1150720319 10</td>
</tr>
<tr>
<td>2147483647 5888 266 29312 604 870 2 0 0</td>
</tr>
</tbody>
</table>

Command example 5.3: Example output from tcpproberdb
5.3.6 Modifications to netem

During the development of RDBv2, we modified netem by adding a new loss mode. By default, netem supports specifying random packet loss. During the development of RDBv2, to reliably reproduce a simple loss pattern, we extended netem with a fixed loss mode. The patch for the Linux kernel and iproute2 can be studied at appendix C.2.
5.4 Experiments

We have performed three experiments with different configurations, where we test the latency by calculating the ACK latency, and the TCP-fairness by comparing the throughput and goodput.

- **Experiment 1** is designed to test the effect RDB has on the packet latency, and especially how it alleviates the HOL blocking issue. To isolate the effect RDB has on the latency as much as possible, the tests in this experiment are run in a more controlled fashion compared to the other experiments. To avoid the unpredictability of a shared bottleneck, we use only one host to generate traffic, and induce different uniform loss rates using `netem`.

- **Experiment 2**, which we call the latency experiment, is designed to test the effect of RDB in a more realistic scenario. In addition to the thin streams that we want to measure, a set of Greedy streams are run simultaneously through the same network path, to cause congestion.

- **Experiment 3** uses the same configuration as experiment 2, but adds a limitation on the amount of data the RDB streams can bundle with each packet.

- **Experiment 4** is set up to test the TCP-friendliness of RDBv2. This is done by using a transmission pattern that tries to increase the throughput by bundling redundant data on streams that are not thin.

From the results in appendix A.2 we have picked out some key results that show interesting effects or patterns that we think is the most important when evaluating RDB.

5.4.1 Reading the plot results

We analyse the goodput, throughput and ACK latency and plot the aggregated values of all the network streams sent from each host along with some key parameters in tables below each plot.

The results are presented in three different plot types, goodput, throughput, and ACK latency. All the plots contain information about the test parameters at the top, which identifies which test they describe. Each plot type is accompanied by a table containing a range of relevant data. We will go through the three different plot types, and show an example of each type.

The plot examples that follow are from a test with three sender hosts, where the two hosts `wsender` and `rdbsender` start 5 thin streams each, which competes against 5 greedy streams produced by `zsender`. The ITT for the thin streams is 10:1, the RTT between all the sender hosts and the receiver host is 150 ms. The rate limit is set at 5000kbit, with `pfifo` queue of length 63 packets. The thin streams write 120 bytes each ITT.

5.4.1.1 Goodput

The goodput is measured on the receiver side `tcpdump` trace, and is aggregated over all the streams of each type. The goodput is presented in a boxplot, where
the goodput rate is aggregated for all the streams. The columns in the table below each goodput plot is as follows:

- **Name**: is the TCP/RDB variation used for these streams.
- **Host**: is the name of the host generating these streams.
- **Streams**: is the number of streams this host is sending.
- **ITT (avg)**: is the minimum, average and maximum values of the average ITT calculated for each stream sent by this host. In figure 5.4, the ITT values of 24/28/32 is the result of calculating the average ITT over all the packets of each stream separately. The stream with the lowest average ITT value had 24 ms, and the highest average was 32 ms. 28 ms is the value calculated by taking the average of the average for each stream.
- **Payload (avg)**: is the minimum, average and maximum values of the average payload calculated for each stream on this host. These values are calculated like the ITT values.
- **CWND**: is the CWND values (Mean, Minimum, Q1, Median, Q3, Maximum) calculated from the values of all the streams sent from this host.

### 5.4.1.2 Throughput

As with goodput, the throughput is calculated from the receiver side tcpdump trace, and is aggregated over all the streams of each stream type. The throughput includes
5.4. Experiments

Figure 5.5: Throughput plot example

the IP and TCP header size in the calculation. The table columns are as follows:

- **Sent p:** is the number of packets sent for the streams.
- **Lost p:** is the number of packets lost for the streams.
- **Retr p:** is the number of packets retransmitted by the streams.
- **RDB p:** is the number of packets containing RDB data sent by the streams.
- **Payl B:** is the total number of payload bytes sent by the streams.
- **Unique B:** is the total amount of unique bytes, i.e., payload bytes subtracted all redundant bytes.
- **Rdn B:** redundant bytes is the payload that was retransmitted after the first transmission, i.e., the payload bytes subtracted the unique bytes. This includes RDB data and data from retransmissions.
- **Retr B:** is the total amount of payload in retransmissions.
- **Loss % (B/p):** is the average byte loss rate, and the average packet loss rate of the streams on this host.
- **DupACKs:** is the total number of dupACKs received by the sender side of all the streams.
5.4.1.3 Latency

This is the aggregated ACK latency for all the streams calculated from the sender side tcpdump trace. In the table in figure 5.6 we can see the average and maximum ACK latency, along with a range of percentiles chosen to give an overview of the most important parts of the distribution. While the maximum ACK latency value can be interesting, it cannot be used to evaluate which group performs the best.
5.4.2 Experiment 1 - Latency tests with uniform loss

The tests we have performed in this experiment are set up with a uniform loss, enforced by netem on the host bridge2. The test setup for this experiment differ significantly to the setup of the experiments in the following sections. The tests in this experiment have only one sender machine, that sends one type of stream. This is to rule out unpredictable effects and variations, such as for the loss rate, caused by competition with greedy streams at the bottleneck.

Therefore, each plot presented in this section shows the result of three different test runs, where each test run generated streams with one type of the TCP variations in the parameters list below.

In contrast to the experiments in the following sections, which use the DPIFL (presented in section 4.1), the bundling rate of the RDB streams in these tests are restricted by a SPIFL. This is to isolate the effect the different PIF limits have on the bundle frequency. The SPIFL is enforced by the function tcp_stream_is_thin_spif (code listing B.8) described in section 4.3.2, the the customized version of tcp_stream_is_thin in the current kernel.

By using a uniform loss distribution, instead of enforcing congestion with greedy cross traffic, we are able to avoid the natural variations to the latencies caused by queuing delay. This allows us to fulfill the main goal of this experiment, which is to illustrate the effect of HOL blocking described in section 3.1.1, and how RDB alleviates this issue.

5.4.2.1 Test parameters

The test parameters for this experiment are as follows:

**Network**

- **RTT**: 150 ms
- **Uniform loss**: 0.5%, 2%, 5%, and 10%

**rdbsender**

- **Stream type**: Thin
- **Stream count**: 20
- **Payload**: 120 bytes
- **ITT**: 30:3, 50:5, and 100:10 ms
- **TCP variation**:
  - **TCP New Reno** Linux kernel version 3.16 with ER+TLP
  - **RDB** with SPIFL4
  - **RDB** with SPIFL8
In appendix A.1 the complete test setup can be seen in table A.1, along with the full set of plots for the experiment.

5.4.2.2 Key results

Figure 5.7 shows the results from running thin streams with ITT of 30:3 and 2% loss. We see how the ACK latency for the RDB streams with SPIFL of 8 (RDB8), is exceptional compared to the other streams, with a 97th percentile of 150 ms. The RDB streams with SPIFL of 4 (RDB4) have the same latencies as the TCP streams, simply because the ITT of 30:3 results in a PIF value above the threshold of 3.

In figure 5.8, where the streams send with an ITT of 50:5, we can see that the ACK latencies for RDB4 and RDB8 are similar. In these tests, the higher ITT enables the RDB4 streams to bundle redundant data on all the packets. In these plots we can observe the same stair pattern as found in the preliminary experiments presented in section 3.1.

From figure 5.8 we can see that the 97th percentile for both the RDB4 and RDB8 streams, is 150 ms. This means that the ACK latencies for 97% of the packets sent were not affected at all by HOL blocking. The small step for the RDB streams between 150 – 200 on the x-axis, is due to the packets that were lost, but who were recovered by the redundant data bundled with the next data packet. The ACK latency for these packets is therefore increased by 50 ms, the same as the ITT.
5.4. Experiments

In figure 5.9 we see a mix, where the RDB₈ can bundle on all packets, and RDB₄ can bundle on some, but not all. In this plot, the stair pattern is even easier to spot. Looking at the ACK latency values for RDB₈, we can see that 90% of the packets were unaffected by the packet loss. The 97th percentile of 83 ms, shows that most packets that got lost were recovered by the redundant data in the following packet.

The RDB₄ streams were only allowed to bundle on some of the packets, which give them lower ACK latencies than the TCP streams, however, the difference to RDB₈ is striking.

5.4.2.3 Summary

The results of the tests in this experiment show us that RDB causes improvements on the ACK latency. When the SPIFL is high enough, allowing RDB to bundle for most packets, the ACK latency for most packets is unaffected by the HOL blocking phenomenon.

We see that HOL blocking is reduced greatly on the streams that are allowed to bundle on all the sent packets. All losses of one packet on the RDB streams cause only a HOL blocking of ITT * 1 as can be seen in (section 5.4.2.1), where the 95th percentile for the RDB₈ streams are 150 ms (RTT = 150), and 99th percentile is 180 ms (RTT + ITT * 1).

The very small second step we can see in figure 5.9, between 180 and 210 on the x-axis, is caused by the loss of two packets in a row, where the third RDB packet recovers the lost data, resulting in an increase of ITT * 2 for the ACK latency. This is confirmed by the 99th of 209 ms.
All in all, we see that the latency numbers for the streams that are allowed to bundle are substantially lower, where the most important improvement is that the potential HOL blocking on the receiver side’s application layer is reduced to a minimum.

**Explaining the stair pattern**

The stair pattern can be explained by a few different factors, mainly the lack of properties that normally cause a high degree of variation to the ACK latencies. The most important factor is that these tests have virtually no queuing delay, causing most events to be triggered on very strict points in time. In addition, disabling Nagle’s Algorithm gives the sender less room to send data packets at it’s own pace. The result is that the ACK latencies for the RDB streams is $RTT + ITT \times L$ where $L$ is the number of times the packet is lost.

When a packet is lost for the TCP streams, the packets that follow are held back on the receiver side due to HOL blocking. These packets have exactly $ITT \times i$ difference in ACK latency where $i$ is their place in the queue. Their ACK latency is dependent on when the missing data is retransmitted, but this is also the result of a “deterministic” chain of events. It is either caused by a) an RTO, which is strictly dependent on the RTT measurements, which does not vary much for these tests, or b) a fast retransmit that is triggered by the reception of ACKs, which are also coming at a “deterministic” time of one RTT after a packet is sent.

---

1Deterministic in the context of a best effort packet switched network.
5.4.3 Experiment 2 - Latency tests with greedy cross traffic

We have performed a set of tests to measure the latency of different thin streams in a network with competing greedy streams. This is to create a network environment that is more realistic than the uniform loss induced by netem.

These tests are set up with three senders, where the hosts rdbsender and wsender send thin streams, and host zsender sends greedy traffic to cause congestion. The zsender is set up to run a constant number of greedy streams of 5 for all tests. This is to ensure a certain amount of traffic through the bottleneck, leading to a more realistic scenario for the thin streams.

5.4.3.1 Test parameters

The test parameters for this experiment are as follows:

Network

- **RTT**: 150 ms
- **Rate limit**: 5000kbit
- **Queue**: pfifo with 63 packet limit

rdbsender

- **Stream type**: Thin
- **Stream count**: 5, 10, and 16
- **Payload**: 120 bytes
- **ITT**: 10:1, 30:3, 50:5, and 75:7 ms
- **TCP variation**:
  - TCP New Reno from Linux kernel version 3.15 with ER+TLP
  - RDB with DPIFL10
  - RDB with DPIFL10 + TFRC
  - RDB with DPIFL20
  - RDB with DPIFL20 + TFRC

wsender

- **Stream type**: Thin
- **Stream count**: 5, 10, and 16
- **Payload**: 120 bytes
- **ITT**: 10:1, 30:3, 50:5, and 75:7 ms
- **TCP variation**: TCP New Reno from Linux kernel version 3.15 with ER+TLP
Chapter 5. Evaluation of RDB prototype v2

**zsender**

- **Stream type:** Greedy
- **Stream count:** 5
- **TCP variation:** system default TCP Cubic with ER+TLP.

In appendix A.2 the complete test setup can be seen in table A.2, followed by the full set of plots for the experiment.

### 5.4.3.2 Key results

The amount of congestion varies to a great deal in the different tests in this experiment. While the tests cover a range of different environments, some are more relevant than others. As the number of thin streams increase in the different tests, we see the effect more network congestion has on the streams.

The tests with 32 \((16 + 16)\) thin streams competing with 5 greedy stream on a 5Mbit bottleneck can be considered an edge case, at least when the ITT is as low as 10:1, where we are at the limit of whether they can be considered as *thin*. We start out with pointing out some key points which is relevant when analyzing any of the results in the experiments.

**Reading the latency values**

As explained in section 5.1.1.2, the ACK latency differs from the application layer latency, and does not provide the complete picture in many cases. Consider the plots in figure 5.10, which show the results from tests with \(5 + 5\) thin streams, performing send system calls with an ITT of 10:1, against 5 greedy stream. We refer to these tests as E2-5-10:1, where the test with only TCP streams is named E2-5-10:1-TCP, and the test with RDB streams is named E2-5-10:1-RDB.

Looking at the latency plots for E2-5-10:1-TCP (subfigure 5.10.(a)) and E2-5-10:1-RDB (subfigure 5.10.(b)), we clearly see a difference in how the ACK latency for the RDB streams (blue line) sticks out. At a quick glance, it looks like the RDB mechanism provides a much lower latency. There are, however, a few important things that can be easy to miss, which may put the results in a different light.

By looking at the ITT values in the table below the goodput plots, we see the difference in queuing delay on the sender side that is not reflected in the ACK latency numbers. For the E2-5-10:1-TCP goodput plot (subfigure 5.11.(a)), the ITTs average at 19 and 20 ms, while in the RDB plot (subfigure 5.11.(b)), they average at 28 ms for the TCP streams, and 33 ms for the RDB streams. This matches well with the payload sizes in each of the tables, showing a big difference between the two tests. In E2-5-10:1-RDB, the streams from both the thin-stream hosts send fewer packets (higher ITT) with larger payload, compared to E2-5-10:1-TCP. Naturally, the packets sent by the RDB streams are much larger due to bundling, however, the amount of new data in each packet is also greater.
If we look closer at the latency values, and compare the results for the two tests, we can see an important difference. While the RDB streams in E2–5:10:1–RDB have lower latencies than the TCP streams, the latencies for all the streams are also increased compared to E2–5:10:1–TCP. The average for any of the streams in E2–5:10:1–RDB is higher than for E2–5:10:1–TCP, and so are most of the percentiles. The exceptions are the 90th, 93rd and 95th percentiles for the RDB streams, which are the lowest in both tests.

This is also reflected in the steepness of how the ACK latency distribution increases in the latency plots, where a steeper increase reflects less variation in the ACK latencies. More congestion on the bottleneck node causes greater variation in queuing delay, and hence, greater variations of the ACK latencies. The more
congested scenario in E2-5-10:1-TCP shows a less steep increase, meaning that more of the values fall within a greater range up to around 0.8. This stands in contrast to the latency plots presented in experiment 1 (section 5.4.2.2), where the queuing delay is non-existent, and the steepness is a straight vertical line.

These values can be explained by comparing the loss rates in the throughput plots of E2-5-10:1-TCP (subfigure 5.12.(a)) and E2-5-10:1-RDB (subfigure 5.12.(b)). Due to the increased payload of the RDB streams caused by redundant bundling, the total throughput is greater, as seen in the throughput plot of E2-5-10:1-TCP. This leads to the rate limit discarding more packets, causing increased loss rates for all the streams. While the number of lost packets for the TCP streams only increase from 1117 in E2-5-10:1-TCP to 1340 in E2-5-10:1-RDB (20% increase), the relative packet loss rate increases from 1.43% to 2.55% (78% increase).

As the relative packet loss rate is the main factor controlling the CWND, this explains the lower send rate in terms of packets sent. This is confirmed by the values in the CWND column, where we see how the thin streams in E2-5-10:1-TCP have slightly higher values, with a median around 13, compared to a median of 10 in E2-5-10:1-TCP.

Effects of TFRC

The plots of the test with TFRC enabled, referred to as E2-5-10:1-RDB-TFRC, show some interesting effects which confirms the reasoning in section 5.4.3.2. From the goodput plot of E2-5-10:1-RDB-TFRC (subfigure 5.13.(a)), we can see that the CWND values average at 12, which is the same as the greedy streams, and higher than for the RDB streams in E2-5-10:1-RDB (which is 10).

The increase in the CWND allows the RDB streams to maintain more PIFs, leading to sending a total of 59.5K packets, compared to 45.2 in E2-5-10:1-RDB. We also see this reflected in the ITT values, which average at 25 ms in E2-5-10:1-
5.4. Experiments

RDB-TFRC compared to 33 ms in E2–5–10:1–RDB. This leads to a lower relative packet loss rate for the RDB streams in E2–5–10:1–RDB–TFRC, even though they send more redundant data in total (increase from 28.2 M in E2–5–10:1–RDB to 38.4 M in E2–5–10:1–RDB–TFRC).

From the latency plot E2–5–10:1–RDB–TFRC (subfigure 5.13.(c)) we see that the average ACK latencies are slightly lower compared to the values for E2–5–10:1–RDB.

When comparing the results from the tests with and without TFRC, we see the following effects:

- The RDB streams in E2–5–10:1–RDB–TFRC get a higher average CWND, leading to more packet sent in total, and a lower average ITT, compared to E2–5–10:1–RDB. This gives less send buffering delay.

- The ACK latency measurements in E2–5–10:1–RDB and E2–5–10:1–RDB–TFRC are approximately the same, with a slight improvement for both thin streams in the test with TFRC enabled.

- The higher CWND in E2–5–10:1–RDB–TFRC compared to E2–5–10:1–RDB does not affect the TCP streams negatively. However, from the results of E2–5–10:1–TCP we can see that the TCP streams perform worse when competing with the RDB streams. The average ITT increases, the number of packets sent decreases, and the ACK latencies are higher.
Chapter 5. Evaluation of RDB prototype v2

Figure 5.13

Less congested scenario with ITT of 30:3

In the tests E2–5–30:3, where the ITT has been raised to 30:3, we see how the streams perform in a less congested scenario. Comparing the goodput plots in figure 5.14 of E2–5–30:3–TCP and E2–5–30:3–RDB, we see that the ITT and CWND values are almost identical, indicating that the negative effect the RDB streams have on the competing TCP streams, is not significant.

From the throughput plots (figure 5.15) we can see that both the thin-stream groups send approximately the same number of packet in both tests, however, looking at the greedy streams, the number of packets sent decreased from 113K in E2–5–30:3–TCP, to 87.8K in E2–5–30:3–RDB. The most revealing information, that shows why the RDB streams have lower latencies than the competing TCP streams, can be found in the column “Retr p” in the throughput plots. The number of packet retransmissions is reduced from 534 in E2–5–30:3–TCP, to 2 in E2–5–30:3–RDB.

From the latency plots in figure 5.16 we see that the RDB streams have considerably lower ACK latencies than the competing TCP streams, where the 99th percentile of the RDB streams is 278 ms, compared to 551 ms for the TCP streams. When comparing the ACK latencies for the TCP streams in E2–5–30:3–TCP, with the TCP streams in E2–5–30:3–RDB, we see the unfortunate effect the RDB
streams have on the competing TCP streams. The ACK latency values for the TCP streams are generally higher in E2–5–30: 3–TCP compared E2–5–30: 3–RDB. This may be explained by the increase in retransmissions, from 542 to 613, due to the higher loss rates caused by the RDB streams.
Chapter 5. Evaluation of RDB prototype v2

5.4.3.3 Summary

In the tests performed in this experiment, we see that the streams utilizing RDB consistently achieve lower latencies than the competing thin stream using TCP. At the same time, we see that the competing streams are affected negatively, to a varying degree depending on the test scenario. In some cases, such as the test E2-5-10:1, both the competing TCP thin streams, and the greedy cross traffic experience a performance reduction when competing with RDB, where the throughput of the greedy streams is reduced significantly, and the latency of the TCP thin streams is heavily increased. In other cases, such as E2-5-30:3, the increased aggressiveness of the RDB streams mostly affect the greedy cross traffic, who’s throughput is greatly decreased, while the competing TCP thin stream experience no increase in the average ITT, and less severe increases to the latencies compared to the E2-5-10:1 tests.

An important key finding in the E2-5-10:1-RDB tests, is that while the RDB streams achieve lower latencies than the competing TCP thin stream, the increased aggressiveness produces more loss and congestion. The result is that, compared to E2-5-10:1-TCP, the latencies for all the streams increase, including the RDB streams, as shown by an increased mean ACK latency.

In these tests we do not see any trace of the stair pattern we found in the tests with uniform loss. This is caused by the continuous variations in the queueing delays on the bottleneck, due to the greedy cross traffic.
5.4.4 Experiment 3 - Latency tests with bundling limitation and greedy cross traffic

This experiment uses the same configuration as experiment 2 (section 5.4.3), except for one property. The bundling rate of the RDB streams in experiment 2 were limited by the DPIFL test, however, when they were allowed to bundle, they would include as much redundant data as possible. In this experiment, they are limited to bundle one previous segment.

5.4.4.1 Test parameters

**rdsender**

- **Bundling limitation on rdsender**: 1 previous SKB.

**rdsender and wsender**

- **Stream count**: 5, 10, 13, and 16
- **ITT**: 10:1, 30:3, 50:5, 75:7, and 100:10 ms

In appendix A.3 the complete test setup can be seen in table A.3, followed by the full set of plots for the experiment.

5.4.4.2 Key results

The goal of this experiment is to measure the effect of applying a bundling limit to the same scenarios tested in experiment 2. We will therefore present some key results from the tests in this experiment, and compare them to the tests in experiment 2, to see how the bundling limitation affects the performance of the RDB streams and the competing streams.

**Low ITT scenario (10:1)**

We will first look at the goodput plots of the low ITT scenario E3-5-10:1 shown in figure 5.17. We see that the ITT values increase slightly from E3-5-10:1-TCP to E3-5-10:1-RDB. The TCP thin stream ITTs in E3-5-10:1-TCP average at 19 ms and 20 ms, whereas in E3-5-10:1-RDB, the TCP thin streams have 21 ms and the RDB streams have 23 ms. While this shows that the RDB streams affect the competing thin streams, the effect is not as severe as observed in the E2-5-10:1 tests from experiment 2.

From the throughput plots in figure 5.18 we see that the thin streams send fewer packets in the RDB test, where the TCP thin streams in E3-5-10:1-TCP send 79.9K and 74K packets, which, in E3-5-10:1-RDB, is reduced to 73.3K for the TCP streams, and 64.3K for the RDB streams.
This shows that the competing TCP thin streams are still affected when competing with RDB, however, the difference compared to the tests in E2-5-10:1 is evident. In E3-5-10:1-RDB, the total packets sent for the thin TCP and RDB streams, is 52.6K and 45.2K, respectively. Comparing the loss rates, we also see that the loss rates for the thin stream of 2.55% and 2.79% in E2-5-10:1-RDB, have been reduced to 1.6%, and 1.74% in E3-5-10:1-RDB.

From the latency plot of E3-5-10:1-RDB (subfigure 5.19,(b)) we also see that, while the TCP thin streams have higher ACK latencies than the RDB streams, where the average is 246 ms for the TCP streams, and 229 ms for the RDB streams, the increase to the latency values is less severe than in E2-5-10:1-RDB, where the average for the TCP streams is 279 ms.
5.4. Experiments

Effects of TFRC

The goodput and throughput plots in figure 5.20 show some interesting results. Looking at the ITT values, we see that they have not changed for the TCP thin streams compared to E3–5–10:1–RDB, but the ITT values for the RDB streams are lower. Being 17/18/19 in E3–5–10:1–RDB–TFRC, they are even lower than the values for the thin streams in E3–5–10:1–TCP (18/20/22) where only TCP streams compete.

From, the throughput plot E3–5–10:1–RDB–TFRC (subfigure 5.20.(b)) we see that the RDB streams send considerably more packets compared to E3–5–10:1–RDB, 80.7K and 64.4K, respectively.

Looking at the latency plot in subfigure 5.20.(c), we see the same effect as observed in E2–5–10:1–RDB–TFRC from experiment 2, where both the TCP and RDB thin streams achieve a slightly lower average ACK latency when TFRC is enabled for the RDB streams.

Differences in comparison to experiment 2

In the E2–5–10:1 tests in experiment 2, we found that the average ACK latency increased from 236 ms in E2–5–10:1–TCP with only TCP streams, to 279 ms in E2–5–10:1–RDB, and 274 ms in E2–5–10:1–RDB–TFRC, when they compete with RDB streams. The difference to the results for E3–5–10:1 is significant. From the latency plot of E3–5–10:1–RDB–TFRC (subfigure 5.20.(c)), we see that the average ACK latency only increases from 236 ms (in E3–5–10:1–TCP) to 246 ms in E3–5–10:1–RDB, and 242 ms in E3–5–10:1–RDB–TFRC.

It is evident that the bundling limitation enforced on the RDB streams in these tests contribute greatly to reducing the negative effects inflicted on the competing TCP thin streams.
Goodput
Duration: 5 min  TFRC: Yes  Thin: 5  Greedy: 5
ITT: 10:1 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
<th>Sent p</th>
<th>Lost p</th>
<th>Retr p</th>
<th>RDB p</th>
<th>Payload B</th>
<th>Unique B</th>
<th>Retr B</th>
<th>Retr B Loss % (B/s)</th>
<th>DupACKs</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>5</td>
<td>19/21/22</td>
<td>247/269/279</td>
<td>12/8/15</td>
<td>71.1 K</td>
<td>1188</td>
<td>1190</td>
<td>0</td>
<td>19.1 M</td>
<td>18.8 M</td>
<td>262.3 K</td>
<td>262.3 K</td>
<td>1.37</td>
</tr>
<tr>
<td>RDB DPIF10 E+TLP</td>
<td>rdbsender</td>
<td>5</td>
<td>17/19/19</td>
<td>363/381/404</td>
<td>14/10/13/18/71</td>
<td>80.7 K</td>
<td>1328</td>
<td>535</td>
<td>15.7</td>
<td>30.7 M</td>
<td>18.5 M</td>
<td>12.2 M</td>
<td>122.2 K</td>
<td>1.46/1.65</td>
</tr>
<tr>
<td>Greedy</td>
<td>zaender</td>
<td>5</td>
<td>16/17/18</td>
<td>1446/1447/1447/1447</td>
<td>14/10/12/18/43</td>
<td>87.1 K</td>
<td>1644</td>
<td>1646</td>
<td>0</td>
<td>126.1 M</td>
<td>122.7 M</td>
<td>2.4 M</td>
<td>2.4 M</td>
<td>1.89/1.89</td>
</tr>
</tbody>
</table>

(a) E3−5:10:1−RDB−TFRC

Throughput
Duration: 5 min  TFRC: Yes  Thin: 5  Greedy: 5
ITT: 10:1 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

Latency
Duration: 5 min  TFRC: Yes  Thin: 5  Greedy: 5
ITT: 10:1 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

(c) E3−5:10:1−RDB−TFRC

Figure 5.20
5.4.4.3 Summary

The results from this experiment show that the ACK latency can be greatly reduced by only bundling one older data segment with each new packet. Enforcing this limitation on bundling will result in less robustness against events where two packets in a row are lost. However, bundling only one previous segment will still recover most occurrences of sporadic loss. The difference in how the RDB streams affect the competing streams is also very clear. In a congested scenario, the increased payload of the RDB packets will result in higher loss rates, at least if the rate limitation scheme we have used on the experiments is representable for real world scenarios. If, however, the rate limitation imposes an artificial limitation that will not be observable in “the wild”, then the harsh bundling restriction used in this experiment may not be necessary.
5.4.5  Experiment 4 - Fairness experiments

We have performed a set of fairness tests with three hosts, where one host (zsender) sends greedy traffic, and two hosts (wsender and rdbsender) send isochronous streams, which differ from the thin streams in the previous experiments only by the amount of data per send call, and the lower ITT rates.

To see the potential of abusing the redundant bundle technique, the rdbsender sends 400 byte chunks of data to the kernel at a fast rate. This leaves room to bundle two earlier segments if the stream is able to transmit continuously without send buffering delay on the sender side. If the rate at which the application writes data to the kernel is too high, the TCP engine will fill each of the SKBs, leaving no room for redundant data.

As a reference point, the wsender host is set to send the same traffic patterns as rdbsender, but with the TCP New Reno CC from Linux kernel version 3.15.

5.4.5.1  Test parameters

The test parameters for this experiment are as follows:

Network

- **RTT**: 150 ms
- **Rate limit**: 5000kbit
- **Queue**: pfifo with 63 packet limit

rdbsender

- **Stream type**: Isochronous
- **Stream count**: 3, 5, 7, 10, and 13
- **Payload**: 400 bytes
- **ITT**: 1, 5:1, 10:1, 15:1, 30:3, and 50:5 ms
- **TCP variation**:
  - TCP New Reno from Linux kernel version 3.15 with ER+TLP
  - RDB with DPIFL10
  - RDB with DPIFL10 + TFRC
  - RDB with DPIFL20
  - RDB with DPIFL20 + TFRC

wsender

- **Stream type**: Isochronous
- **Stream count**: 3, 5, 7, 10, and 13
- **Payload**: 400 bytes
- **ITT**: 1, 5:1, 10:1, 15:1, 30:3, and 50:5 ms
- **TCP variation**: TCP New Reno from Linux kernel version 3.15 with ER+TLP
Chapter 5.

Evaluation of RDB prototype v2

5.4.5.2 Key results

In the tests with the lowest ITT, such as E4–3–5:1–RDB (subfigure 5.21.(a)), we can see that the RDB streams are for the most part unable to bundle. The result is that the average payload size of the isochronous streams is close to the MSS of 1460 bytes.

In E4–3–5:1–RDB–TFRC, the same test with TFRC enabled (subfigure 5.21.(b)), the only difference is that the TFRC CC is stricter, giving the RDB streams a lower CWND.

In a higher ITT scenario, such as E4–5–30:3, we see from figure 5.22 how the average ITT for both the isochronous streams are slightly higher in the E4–5–30:3–RDB. From the throughput plots in figure 5.23, we see that, while being able to bundle on most packets, the RDB streams only achieve half the throughput rate of the greedy streams. The goodput is still not higher than the isochronous TCP streams.

Moving to the E4–10–30:3 tests, we see a congested scenario where the RDB streams are allowed to bundle on more than half of the packets sent. Figure 5.24

In appendix A.4 the complete test setup can be seen in table A.4, followed by the full set of plot results for the experiment.
shows the throughput plots of the two tests E4–10–30 : 3–TCP and E4–10–30 : 3–RDB–TFRC. In the non-TFRC test, the RDB streams achieve considerably less throughput than the greedy streams. In the TFRC test, they get closer, but are still not able to match the throughput of the greedy streams. The goodput plot in figure 5.25 shows that the goodput is at the same level as the isochronous TCP streams, leaving nothing to gain from using RDB.

5.4.5.3 Summary

The main goal of this experiment is to show that RDBv2 maintains TCP-friendliness, even in scenarios designed for optimizing the possibility of gaining an unfair advantage. As the results show, in none of the tests do the RDB streams get an advantage over the other streams throughput-wise.
Making the abuser application smarter

When the TCP output queue is increasing we know that the effect is that RDB is disabled. A smarter abuser could use this knowledge to try to improve the chance of having more data bundled in RDB packets. The amount the queued outgoing data on the socket can be fetched from the kernel (SIOCOUTQ on Linux), and based on this the sender application could adjust the send rate to avoid filling the queue. The ideal would be to adjust the pace of the send calls to the kernel, such that there is just enough room to bundle redundant data with each packet.
5.5 Summary

In this chapter we have presented our test setup, and described the key findings in each experiment. The experiments show that RDB gives thin stream over TCP significantly lower per-packet latency, mostly by reducing the negative effects of the HOL blocking phenomenon. By imposing limitations on how much redundant data can be bundled, the RDB mechanism can achieve great improvements to the latencies, without affecting competing thin stream significantly.

The range of environments tested in the experiments does not represent the environments where we believe RDB should be used. However, to find which scenarios RDB can be used, it is necessary to test different edge case scenarios. Whereas the lowest ITTs used in the experiments were as low as 10 ms, it does not mean we will argue that it is a reasonable limit for when to identify a stream as thin. However, as the experiments show, RDB streams with such low ITTs can be made to have little affect on competing streams.

Based on the results of the experiments, we argue that RDB definitely can be used by thin streams in many scenarios, without penalizing competing traffic in an unfair manner. However, further tests are required to figure out when and where to enable RDB.
6.1 Summary

One often hears about how fast things change, especially in the realm of computer science. While being first specified in the “ancient” times of 1970s (RFC675), TCP is still the first choice for most general applications on the Internet. Its ease of use and reliability guarantees makes it a solid choice of protocol for applications that do not transmit time-dependent data. While there are alternatives to TCP, it would be of great benefit to improve TCP’s performance for applications with strict latency requirements.

The RDB mechanism we have developed and tested, does not aim to replace the default CCs used today, but to be an easy alternative for applications that have strict latency requirements, and for various reasons need to use TCP.

In the later years, research has started focusing on the latency performance of TCP, with new mechanisms emerging, like LT, mFR, ER and TLP. What all these mechanisms have in common is that they try to minimize the time before lost packets are retransmitted by the TCP engine. The problem is that they wait until they believe it is likely that a loss has occurred. This means that the per-packet latencies for lost packets are increased by a minimum of one RTT, but often it can be more. This is where RDB takes a different path, and retransmits un-ACKed data segments simply because it may be(come) lost. By bundling old un-ACKed data that probably will be redundant, it will proactively ensure that the per-packet (or per-segment) latency is not drastically reduced in case of loss. We argue that this can be justified because RDB does not create extra packets on the wire by ensuring to create packets that are below the MSS limit.

The TFRC CC we have implemented in RDBv2 raises the CWND for thin streams compared to the latest version of the Linux kernel version 3.19, which causes more packets on the wire in some cases. This is not because it allows the stream to send more data, but because, the data is, to a greater degree, queued on the sender side, when the CWND is a limiting factor.

Through experiments we have confirmed the vast improvements in latencies reported by Petlund [2009, a4] when using RDB. We have also verified that the RDBv2 implementation is TCP fair, at least within the scope of our test environment.
6.2 Contributions

The work in this thesis has focused on how to improve the latency for thin streams using TCP. While some recent work has focused on mechanisms for reducing the retransmission delay for thin streams over TCP (section 2.6), the lost data segments still require retransmissions. The RDB mechanism modifies the packets sent by TCP by bundling already sent data with the packets containing new, un-transmitted data. By bundling redundant data with the data packets, an RDB-stream can recover from sporadic packet loss without having to wait for a retransmission from the sender host.

Based on earlier work on RDB (Evensen [2008, a5]; Petlund [2009, a4]), this thesis tries to tame the RDB mechanism, to ensure that the latency improvements for the streams utilizing RDB is balanced against the negative effects inflicted on competing traffic.

The main contributions of this thesis can be summarized as follows:

- We have evaluated the thin-stream classification mechanism in the Linux kernel. Further, we have developed an alternative method, which uses the RTT, together with the user space application’s transmission patterns, to identify if the stream is thin.
- We have identified an issue with recent changes to the Linux kernel’s TCP engine that severely affects the latency for thin streams negatively.
- Implementation of RDB as a Linux kernel module, that enables a sender host to transmit data to a TCP receiver using the RDB technique.
- To help thin streams maintain a more stable throughput rate than provided by standard TCP, we have implemented a CC in the kernel module which utilizes the equation based throughput calculation for TFRC-SP.
- Investigated and developed methods for detecting packet loss that RDB hides from the TCP engine.

The main contributions from the experiments performed on RDB, can be summarized as follows:

- Investigating how the redundant bundling mechanism performs under different test scenarios, and finding a tradeoff between improving the latency and increasing the payload size when bundling redundant data.
- The latency gains achieved with different bundling rates, i.e., limitations to how much data can be bundled with each packet.
- How RDB streams affect competing traffic under different test scenarios.
- How RDB can be prevented from being abused to obtain an advantage over competing traffic.
6.3 Future work

Thin streams over TCP have been overlooked for a long time, and require further attention to be able to compete with other transport protocols. Some candidates for future work is outlined next:

- **Investigate the recent changes to the Linux kernel**
  Recent changes to the Linux kernel’s TCP engine seem to affect the latency of thin streams (described in section 3.4). The negative effects these changes have, may force current applications sending thin streams over TCP over on other protocols. This issue should be investigated further, firstly by finding out if the behavior is intended or not. Hopefully it can be regarded as a bug, so that work on how to fix and improve the situation can be done. If it is intended, the standards need to be changed if thin streams should have a future together with TCP.

- **Improving the loss detection mechanisms for RDB.**
  Of the techniques we presented in section 4.2 for detecting loss, only the technique looking at multiple ACKed packets was implemented in RDBv2. As this technique is reliable as long as there is no packet reordering or loss of ACK packets, it may perform worse in the “real world”. However, the lack of mechanisms for detecting reordering can only lead to an over-estimation of the loss rate, which does not lead to unfairness in favor of the RDB streams. Further improvements to the loss detection can be made by implementing the technique detailed in section 4.2.0.4, where the DSACK information in the ACKs is utilized to identify which packet was lost.

- **Performance profiling**
  To measure the resource overhead caused by the different parts of the module it is necessary to perform profiling.

- **Measure the true Application layer latency**
  Due to the limitations of the ACK latency measurement explained in section 5.1.1.2, further work should be done on finding a solution on how to measure the application layer latency. This is important to get more precise latency measurements, allowing a fair comparison between the results of different mechanisms, by taking into account the sender side queuing delay.

  - One possible solution is to use kprobes to hook into the relevant functions in the kernel to get the timestamps of when the payload enters and leaves the kernel.
  - Another solution could be to implement this in the sender application such as streamzero, that saves the timestamps for the payload. By using the same computer for sending and receiving, the clock drift issue can be evaded.
Bibliography

References (a)


Online References (b)


URL: https://www.itu.int/rec/T-REC-G.114-200305-I (visited on 06/2014) (cit. on pp. 8, 9).

[b8] FreeBSD 9 source code.
VERSION: svn checkout -r252825. (sys/netinet/tcp_timer.h line 111).


[b10] Linux kernel IPv4 variables.

Internet Standards and Drafts (c)


Appendix A

Experiments results

A.1 Latency tests with uniform loss

<table>
<thead>
<tr>
<th>Traffic characteristics</th>
<th>Network properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streams</td>
<td>Payload</td>
</tr>
<tr>
<td>Figure A.1.1.(a)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.1.(a)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.1.(a)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.1.(b)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.1.(b)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.1.(b)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.1.(c)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.1.(c)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.1.(c)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.1.(d)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.1.(d)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.1.(d)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.2.(a)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.2.(a)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.2.(a)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.2.(b)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.2.(b)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.2.(b)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.2.(c)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.2.(c)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.2.(c)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.2.(d)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.2.(d)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.2.(d)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.3.(a)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.3.(a)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.3.(a)</td>
<td>20</td>
</tr>
<tr>
<td>Figure A.1.3.(b)</td>
<td>20</td>
</tr>
</tbody>
</table>
## Appendix A. Experiments results

<table>
<thead>
<tr>
<th>Figure</th>
<th>Setting</th>
<th>Protocol</th>
<th>RDB</th>
<th>QoS</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1.3. (b)</td>
<td>20 120</td>
<td>50:5 RDB</td>
<td>8</td>
<td>150</td>
<td>5</td>
</tr>
<tr>
<td>A.1.3. (b)</td>
<td>20 120</td>
<td>50:5 RDB</td>
<td>4</td>
<td>150</td>
<td>5</td>
</tr>
<tr>
<td>A.1.3. (c)</td>
<td>20 120</td>
<td>75:7 RDB</td>
<td>4</td>
<td>150</td>
<td>5</td>
</tr>
<tr>
<td>A.1.3. (c)</td>
<td>20 120</td>
<td>75:7 TCP</td>
<td>NA</td>
<td>150</td>
<td>5</td>
</tr>
<tr>
<td>A.1.3. (c)</td>
<td>20 120</td>
<td>75:7 RDB</td>
<td>8</td>
<td>150</td>
<td>5</td>
</tr>
<tr>
<td>A.1.3. (d)</td>
<td>20 120</td>
<td>100:10 RDB</td>
<td>8</td>
<td>150</td>
<td>5</td>
</tr>
<tr>
<td>A.1.3. (d)</td>
<td>20 120</td>
<td>100:10 TCP</td>
<td>NA</td>
<td>150</td>
<td>5</td>
</tr>
<tr>
<td>A.1.3. (d)</td>
<td>20 120</td>
<td>100:10 RDB</td>
<td>4</td>
<td>150</td>
<td>5</td>
</tr>
<tr>
<td>A.1.4. (a)</td>
<td>20 120</td>
<td>30:3 TCP</td>
<td>NA</td>
<td>150</td>
<td>10</td>
</tr>
<tr>
<td>A.1.4. (a)</td>
<td>20 120</td>
<td>30:3 RDB</td>
<td>4</td>
<td>150</td>
<td>10</td>
</tr>
<tr>
<td>A.1.4. (a)</td>
<td>20 120</td>
<td>30:3 RDB</td>
<td>8</td>
<td>150</td>
<td>10</td>
</tr>
<tr>
<td>A.1.4. (b)</td>
<td>20 120</td>
<td>50:5 TCP</td>
<td>NA</td>
<td>150</td>
<td>10</td>
</tr>
<tr>
<td>A.1.4. (b)</td>
<td>20 120</td>
<td>50:5 RDB</td>
<td>8</td>
<td>150</td>
<td>10</td>
</tr>
<tr>
<td>A.1.4. (b)</td>
<td>20 120</td>
<td>50:5 RDB</td>
<td>4</td>
<td>150</td>
<td>10</td>
</tr>
<tr>
<td>A.1.4. (c)</td>
<td>20 120</td>
<td>75:7 RDB</td>
<td>4</td>
<td>150</td>
<td>10</td>
</tr>
<tr>
<td>A.1.4. (c)</td>
<td>20 120</td>
<td>75:7 TCP</td>
<td>NA</td>
<td>150</td>
<td>10</td>
</tr>
<tr>
<td>A.1.4. (c)</td>
<td>20 120</td>
<td>75:7 RDB</td>
<td>8</td>
<td>150</td>
<td>10</td>
</tr>
<tr>
<td>A.1.4. (d)</td>
<td>20 120</td>
<td>100:10 RDB</td>
<td>8</td>
<td>150</td>
<td>10</td>
</tr>
<tr>
<td>A.1.4. (d)</td>
<td>20 120</td>
<td>100:10 TCP</td>
<td>NA</td>
<td>150</td>
<td>10</td>
</tr>
<tr>
<td>A.1.4. (d)</td>
<td>20 120</td>
<td>100:10 RDB</td>
<td>4</td>
<td>150</td>
<td>10</td>
</tr>
</tbody>
</table>

**Table A.1:** Test setup for experiment 1
A.1. Latency tests with uniform loss

Figure A.1.1
### Appendix A. Experiments results

Duration: 5 min  Streams: 20  RTT: 150 ms  QLEN: 60

<table>
<thead>
<tr>
<th>Name</th>
<th>Mean</th>
<th>40th</th>
<th>50th</th>
<th>60th</th>
<th>70th</th>
<th>80th</th>
<th>90th</th>
<th>95th</th>
<th>97th</th>
<th>99th</th>
<th>99.9th</th>
<th>99.99th</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>172</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>871</td>
<td>1433</td>
<td></td>
</tr>
<tr>
<td>RDB 4</td>
<td>170</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>630</td>
<td>1158</td>
<td></td>
</tr>
<tr>
<td>RDB 8</td>
<td>151</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>231</td>
<td>567</td>
<td></td>
</tr>
</tbody>
</table>

- **(a)**

Duration: 5 min  Streams: 20  RTT: 150 ms  QLEN: 60

<table>
<thead>
<tr>
<th>Name</th>
<th>Mean</th>
<th>40th</th>
<th>50th</th>
<th>60th</th>
<th>70th</th>
<th>80th</th>
<th>90th</th>
<th>95th</th>
<th>97th</th>
<th>99th</th>
<th>99.9th</th>
<th>99.99th</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>172</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>871</td>
<td>1433</td>
<td></td>
</tr>
<tr>
<td>RDB 4</td>
<td>152</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>213</td>
<td>1238</td>
<td></td>
</tr>
<tr>
<td>RDB 8</td>
<td>152</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>210</td>
<td>1504</td>
<td></td>
</tr>
</tbody>
</table>

- **(b)**

Duration: 5 min  Streams: 20  RTT: 150 ms  QLEN: 60

<table>
<thead>
<tr>
<th>Name</th>
<th>Mean</th>
<th>40th</th>
<th>50th</th>
<th>60th</th>
<th>70th</th>
<th>80th</th>
<th>90th</th>
<th>95th</th>
<th>97th</th>
<th>99th</th>
<th>99.9th</th>
<th>99.99th</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>168</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>1150</td>
<td>1802</td>
<td></td>
</tr>
<tr>
<td>RDB 4</td>
<td>152</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>304</td>
<td>589</td>
<td></td>
</tr>
<tr>
<td>RDB 8</td>
<td>153</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>3215</td>
<td>9667</td>
<td></td>
</tr>
</tbody>
</table>

- **(c)**

Duration: 5 min  Streams: 20  RTT: 150 ms  QLEN: 60

<table>
<thead>
<tr>
<th>Name</th>
<th>Mean</th>
<th>40th</th>
<th>50th</th>
<th>60th</th>
<th>70th</th>
<th>80th</th>
<th>90th</th>
<th>95th</th>
<th>97th</th>
<th>99th</th>
<th>99.9th</th>
<th>99.99th</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>160</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>592</td>
<td>675</td>
<td></td>
</tr>
<tr>
<td>RDB 4</td>
<td>153</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>272</td>
<td>438</td>
<td></td>
</tr>
<tr>
<td>RDB 8</td>
<td>153</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>270</td>
<td>435</td>
<td></td>
</tr>
</tbody>
</table>

- **(d)**

---

Figure A.1.2
A.1. Latency tests with uniform loss

**Figure A.1.3**
### Duration: 5 min  Streams: 20  RTT: 150 ms  QLEN: 60

<table>
<thead>
<tr>
<th>Name</th>
<th>Mean</th>
<th>40th</th>
<th>50th</th>
<th>60th</th>
<th>70th</th>
<th>80th</th>
<th>90th</th>
<th>95th</th>
<th>97th</th>
<th>99th</th>
<th>99.9th</th>
<th>99.99th</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>270</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>2852</td>
<td>4095</td>
</tr>
<tr>
<td>RDB 4</td>
<td>227</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>5495</td>
<td>9650</td>
</tr>
<tr>
<td>RDB 8</td>
<td>154</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>813</td>
<td>3154</td>
</tr>
</tbody>
</table>

### Duration: 5 min  Streams: 20  RTT: 150 ms  QLEN: 60

<table>
<thead>
<tr>
<th>Name</th>
<th>Mean</th>
<th>40th</th>
<th>50th</th>
<th>60th</th>
<th>70th</th>
<th>80th</th>
<th>90th</th>
<th>95th</th>
<th>97th</th>
<th>99th</th>
<th>99.9th</th>
<th>99.99th</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>273</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>2175</td>
<td>3230</td>
</tr>
<tr>
<td>RDB 4</td>
<td>158</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>751</td>
<td>2266</td>
</tr>
<tr>
<td>RDB 8</td>
<td>156</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>296</td>
<td>979</td>
</tr>
</tbody>
</table>

### Duration: 5 min  Streams: 20  RTT: 150 ms  QLEN: 60

<table>
<thead>
<tr>
<th>Name</th>
<th>Mean</th>
<th>40th</th>
<th>50th</th>
<th>60th</th>
<th>70th</th>
<th>80th</th>
<th>90th</th>
<th>95th</th>
<th>97th</th>
<th>99th</th>
<th>99.9th</th>
<th>99.99th</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>223</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>1896</td>
<td>2541</td>
</tr>
<tr>
<td>RDB 4</td>
<td>163</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>2198</td>
<td>6105</td>
</tr>
<tr>
<td>RDB 8</td>
<td>162</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>639</td>
<td>1380</td>
</tr>
</tbody>
</table>

---

**Figure A.1.4**
## A.2 Latency tests with greedy cross traffic

### Table A.2

<table>
<thead>
<tr>
<th>Figure</th>
<th>Type</th>
<th>Streams</th>
<th>Cong</th>
<th>TFRC</th>
<th>RTT</th>
<th>ITT</th>
<th>Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.2.1</td>
<td>TCP ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>10:1</td>
<td>120</td>
</tr>
<tr>
<td>A.2.2</td>
<td>RDB DPIF10 ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>10:1</td>
<td>120</td>
</tr>
<tr>
<td>A.2.3</td>
<td>RDB DPIF10 ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>10:1</td>
<td>120</td>
</tr>
<tr>
<td>A.2.4</td>
<td>RDB DPIF20 ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>10:1</td>
<td>120</td>
</tr>
<tr>
<td>A.2.5</td>
<td>RDB DPIF20 ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>10:1</td>
<td>120</td>
</tr>
<tr>
<td>A.2.6</td>
<td>TCP ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>30:3</td>
<td>120</td>
</tr>
<tr>
<td>A.2.7</td>
<td>RDB DPIF10 ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>30:3</td>
<td>120</td>
</tr>
<tr>
<td>A.2.8</td>
<td>RDB DPIF10 ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>30:3</td>
<td>120</td>
</tr>
<tr>
<td>A.2.9</td>
<td>RDB DPIF20 ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>30:3</td>
<td>120</td>
</tr>
<tr>
<td>A.2.10</td>
<td>RDB DPIF20 ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>30:3</td>
<td>120</td>
</tr>
<tr>
<td>A.2.11</td>
<td>TCP ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>50:5</td>
<td>120</td>
</tr>
<tr>
<td>A.2.12</td>
<td>RDB DPIF10 ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>50:5</td>
<td>120</td>
</tr>
<tr>
<td>A.2.13</td>
<td>RDB DPIF10 ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>50:5</td>
<td>120</td>
</tr>
<tr>
<td>A.2.14</td>
<td>RDB DPIF20 ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>50:5</td>
<td>120</td>
</tr>
<tr>
<td>A.2.15</td>
<td>RDB DPIF20 ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>50:5</td>
<td>120</td>
</tr>
<tr>
<td>A.2.16</td>
<td>TCP ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>75:7</td>
<td>120</td>
</tr>
<tr>
<td>A.2.17</td>
<td>RDB DPIF10 ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>75:7</td>
<td>120</td>
</tr>
<tr>
<td>A.2.18</td>
<td>RDB DPIF10 ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>75:7</td>
<td>120</td>
</tr>
<tr>
<td>A.2.19</td>
<td>RDB DPIF20 ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>75:7</td>
<td>120</td>
</tr>
<tr>
<td>A.2.20</td>
<td>RDB DPIF20 ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>75:7</td>
<td>120</td>
</tr>
<tr>
<td>A.2.21</td>
<td>TCP ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>10:1</td>
<td>120</td>
</tr>
<tr>
<td>A.2.22</td>
<td>RDB DPIF10 ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>10:1</td>
<td>120</td>
</tr>
<tr>
<td>A.2.23</td>
<td>RDB DPIF10 ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>10:1</td>
<td>120</td>
</tr>
<tr>
<td>A.2.24</td>
<td>RDB DPIF20 ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>10:1</td>
<td>120</td>
</tr>
<tr>
<td>A.2.25</td>
<td>RDB DPIF20 ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>10:1</td>
<td>120</td>
</tr>
<tr>
<td>A.2.26</td>
<td>TCP ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>30:3</td>
<td>120</td>
</tr>
<tr>
<td>A.2.27</td>
<td>RDB DPIF10 ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>30:3</td>
<td>120</td>
</tr>
<tr>
<td>A.2.28</td>
<td>RDB DPIF10 ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>30:3</td>
<td>120</td>
</tr>
<tr>
<td>A.2.29</td>
<td>RDB DPIF20 ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>30:3</td>
<td>120</td>
</tr>
<tr>
<td>A.2.30</td>
<td>RDB DPIF20 ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>30:3</td>
<td>120</td>
</tr>
<tr>
<td>A.2.31</td>
<td>TCP ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>50:5</td>
<td>120</td>
</tr>
<tr>
<td>A.2.32</td>
<td>RDB DPIF10 ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>50:5</td>
<td>120</td>
</tr>
<tr>
<td>A.2.33</td>
<td>RDB DPIF10 ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>50:5</td>
<td>120</td>
</tr>
<tr>
<td>A.2.34</td>
<td>RDB DPIF20 ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>50:5</td>
<td>120</td>
</tr>
<tr>
<td>A.2.35</td>
<td>RDB DPIF20 ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>50:5</td>
<td>120</td>
</tr>
<tr>
<td>A.2.36</td>
<td>TCP ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>75:7</td>
<td>120</td>
</tr>
<tr>
<td>A.2.37</td>
<td>RDB DPIF10 ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>75:7</td>
<td>120</td>
</tr>
<tr>
<td>A.2.38</td>
<td>RDB DPIF10 ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>75:7</td>
<td>120</td>
</tr>
<tr>
<td>A.2.39</td>
<td>RDB DPIF20 ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>75:7</td>
<td>120</td>
</tr>
<tr>
<td>A.2.40</td>
<td>RDB DPIF20 ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>75:7</td>
<td>120</td>
</tr>
<tr>
<td>A.2.41</td>
<td>TCP ER+TLP</td>
<td>16</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>10:1</td>
<td>120</td>
</tr>
<tr>
<td>A.2.42</td>
<td>RDB DPIF10 ER+TLP</td>
<td>16</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>10:1</td>
<td>120</td>
</tr>
<tr>
<td>A.2.43</td>
<td>RDB DPIF10 ER+TLP</td>
<td>16</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>10:1</td>
<td>120</td>
</tr>
<tr>
<td>A.2.44</td>
<td>RDB DPIF20 ER+TLP</td>
<td>16</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>10:1</td>
<td>120</td>
</tr>
<tr>
<td>A.2.45</td>
<td>RDB DPIF20 ER+TLP</td>
<td>16</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>10:1</td>
<td>120</td>
</tr>
<tr>
<td>A.2.46</td>
<td>TCP ER+TLP</td>
<td>16</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>30:3</td>
<td>120</td>
</tr>
</tbody>
</table>
Table A.2: Test setup for experiment 2

| Figure A.2.47 | RDB DPIF10 ER+TLP | 16 | Rdb | No | 150 | 30:3 | 120 |
| Figure A.2.48 | RDB DPIF10 ER+TLP | 16 | Rdb | Yes | 150 | 30:3 | 120 |
| Figure A.2.49 | RDB DPIF20 ER+TLP | 16 | Rdb | No | 150 | 30:3 | 120 |
| Figure A.2.50 | RDB DPIF20 ER+TLP | 16 | Rdb | Yes | 150 | 30:3 | 120 |
| Figure A.2.51 | TCP ER+TLP | 16 | Rdb | No | 150 | 50:5 | 120 |
| Figure A.2.52 | RDB DPIF10 ER+TLP | 16 | Rdb | No | 150 | 50:5 | 120 |
| Figure A.2.53 | RDB DPIF10 ER+TLP | 16 | Rdb | Yes | 150 | 50:5 | 120 |
| Figure A.2.54 | RDB DPIF20 ER+TLP | 16 | Rdb | No | 150 | 50:5 | 120 |
| Figure A.2.55 | RDB DPIF20 ER+TLP | 16 | Rdb | Yes | 150 | 50:5 | 120 |
| Figure A.2.56 | TCP ER+TLP | 16 | Rdb | No | 150 | 75:7 | 120 |
| Figure A.2.57 | RDB DPIF10 ER+TLP | 16 | Rdb | No | 150 | 75:7 | 120 |
| Figure A.2.58 | RDB DPIF10 ER+TLP | 16 | Rdb | Yes | 150 | 75:7 | 120 |
| Figure A.2.59 | RDB DPIF20 ER+TLP | 16 | Rdb | No | 150 | 75:7 | 120 |
| Figure A.2.60 | RDB DPIF20 ER+TLP | 16 | Rdb | Yes | 150 | 75:7 | 120 |
Figure A.2.1

Figure A.2.2
**Figure A.2.5**

**Goodput**
- Duration: 5 min
- TFRC: Yes
- Thin: 5
- Greedy: 5
- ITT: 10:1 ms
- RTT: 150 ms
- QLEN: 63p
- Payload: 120 B

**TCP variation used for competing streams**

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>5</td>
<td>26 / 28 / 31</td>
<td>325399591258</td>
<td>101/71/10328</td>
</tr>
<tr>
<td>RDB DPIF10 ER+TLP</td>
<td>zsender</td>
<td>5</td>
<td>24 / 25 / 26</td>
<td>8869591023</td>
<td>121/8/111486</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>19 / 21 / 23</td>
<td>144614471448</td>
<td>121/8/111514/42</td>
</tr>
</tbody>
</table>

**Throughput**
- Duration: 5 min
- TFRC: Yes
- Thin: 5
- Greedy: 5
- ITT: 10:1 ms
- RTT: 150 ms
- QLEN: 63p
- Payload: 120 B

**TCP variation used for competing streams**

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>5</td>
<td>19 / 19 / 20</td>
<td>24472020253</td>
<td>131/9/121629</td>
</tr>
<tr>
<td>RDB DPIF10 ER+TLP</td>
<td>zsender</td>
<td>5</td>
<td>18 / 18 / 18</td>
<td>274276287</td>
<td>131/10/131628</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>15 / 16 / 17</td>
<td>144614471447</td>
<td>144/10/131638</td>
</tr>
</tbody>
</table>

**Latency**
- Duration: 5 min
- TFRC: Yes
- Thin: 5
- Greedy: 5
- ITT: 10:1 ms
- RTT: 150 ms
- QLEN: 63p
- Payload: 120 B

**TCP variation used for competing streams**

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>5</td>
<td>26 / 28 / 31</td>
<td>325399591258</td>
<td>101/71/10328</td>
</tr>
<tr>
<td>RDB DPIF10 ER+TLP</td>
<td>zsender</td>
<td>5</td>
<td>24 / 25 / 26</td>
<td>8869591023</td>
<td>121/8/111486</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>19 / 21 / 23</td>
<td>144614471448</td>
<td>121/8/111514/42</td>
</tr>
</tbody>
</table>

**Figure A.2.4**

**Goodput**
- Duration: 5 min
- TFRC: Yes
- Thin: 5
- Greedy: 5
- ITT: 10:1 ms
- RTT: 150 ms
- QLEN: 63p
- Payload: 120 B

**Throughput**
- Duration: 5 min
- TFRC: Yes
- Thin: 5
- Greedy: 5
- ITT: 10:1 ms
- RTT: 150 ms
- QLEN: 63p
- Payload: 120 B

**Latency**
- Duration: 5 min
- TFRC: Yes
- Thin: 5
- Greedy: 5
- ITT: 10:1 ms
- RTT: 150 ms
- QLEN: 63p
- Payload: 120 B
Figure A.2.5

(a) Goodput
Duration: 5 min  TFRC: Yes  Thin: 5  Greedy: 5
ITT: 10.1 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP ER+TLP  RDB DPIF20 ER+TLP  Greedy

(b) Throughput
Duration: 5 min  TFRC: Yes  Thin: 5  Greedy: 5
ITT: 10.1 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP ER+TLP  RDB DPIF20 ER+TLP  Greedy

(c) Latency
Duration: 5 min  TFRC: Yes  Thin: 5  Greedy: 5
ITT: 10.1 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP variation used for competing streams

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>rdsender</td>
<td>5</td>
<td>19 / 21 / 24</td>
<td>247 / 272 / 312</td>
<td>12 / 15 / 15</td>
</tr>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>5</td>
<td>21 / 22 / 23</td>
<td>373 / 393 / 418</td>
<td>14 / 16 / 16</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>16 / 16 / 17</td>
<td>1447 / 1447 / 1447</td>
<td>14 / 10 / 13</td>
</tr>
</tbody>
</table>

Sent B  Lost B  Retr B  Unique B  Rdn B  Retr B  Loss %  (B/s)  DupACKs
70.5 K  1191  1193  0  19.1 M  18.8 M  240 K  249 K  1.30 / 1.66  10415
90.4 K  1686  1686  0  131 M  128 M  2.4 M  2.4 M  1.86 / 1.86  12882

Mean 50th  70th  90th  95th  97th  98th  99th  99.9th  99.99th  Max
244 K  296 K  357 K  401 K  455 K  499 K  543 K  549 K  570 K  754 K  1254 K  1429 K
241 K  293 K  350 K  402 K  449 K  474 K  520 K  524 K  570 K  754 K  1254 K  1429 K
250 K  298 K  354 K  407 K  450 K  477 K  523 K  527 K  577 K  754 K  1254 K  1429 K

Figure A.2.6

(a) Goodput
Duration: 5 min  TFRC: No  Thin: 5  Greedy: 5
ITT: 10.1 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP variation used for competing streams

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>rdsender</td>
<td>5</td>
<td>31 / 31 / 32</td>
<td>126 / 127 / 130</td>
<td>11 / 13 / 13</td>
</tr>
<tr>
<td>TCP ER+TLP</td>
<td>rdsender</td>
<td>5</td>
<td>31 / 31 / 32</td>
<td>126 / 127 / 130</td>
<td>11 / 13 / 13</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>12 / 13 / 14</td>
<td>1446 / 1447 / 1448</td>
<td>18 / 16 / 17</td>
</tr>
</tbody>
</table>

Sent B  Lost B  Retr B  Unique B  Rdn B  Retr B  Loss %  (B/s)  DupACKs
48.4 K  540  542  0  6.2 M  6.1 M  65.5 K  65.5 K  1.08 / 1.12  4025
47.8 K  532  534  0  6.1 M  6.0 M  64.3 K  64.3 K  1.06 / 1.11  4062
113 K  1175  1177  0  164 M  162 M  1.7 M  1.7 M  1.04 / 1.04  13889

Mean 50th  70th  90th  95th  97th  98th  99th  99.9th  99.99th  Max
246 K  296 K  357 K  401 K  455 K  499 K  543 K  549 K  570 K  754 K  1254 K  1429 K
245 K  293 K  350 K  402 K  449 K  474 K  520 K  524 K  577 K  754 K  1254 K  1429 K
253 K  298 K  354 K  407 K  450 K  477 K  523 K  527 K  577 K  754 K  1254 K  1429 K

(b) Throughput
Duration: 5 min  TFRC: No  Thin: 5  Greedy: 5
ITT: 30.3 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP variation used for competing streams

(c) Latency
Duration: 5 min  TFRC: No  Thin: 5  Greedy: 5
ITT: 30.3 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP variation used for competing streams

ACK Latency in milliseconds

(a) TCP variation used for competing streams

(b) TCP variation used for competing streams

(c) TCP variation used for competing streams
Figure A.2.9

Figure A.2.10
Figure A.2.11

Figure A.2.12
**A.2. Latency tests with greedy cross traffic**

---

**Figure A.2.13**

**Throughput**

Duration: 5 min  
TFRC: Yes  
Thin: 5  
Greedy: 5  
ITT: 50.5 ms  
RTT: 150 ms  
QLEN: 63p  
Payload: 120 B  

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>5</td>
<td>50 / 50 / 50</td>
<td>122/122/122</td>
<td>9/1/8/10/10/11</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>13 / 14 / 14 1447/1447/1447</td>
<td>18/7/15/18/21/41</td>
<td></td>
</tr>
</tbody>
</table>

---

**Latency**

Duration: 5 min  
TFRC: Yes  
Thin: 5  
Greedy: 5  
ITT: 50.5 ms  
RTT: 150 ms  
QLEN: 63p  
Payload: 120 B  

---

**Figure A.2.14**

**Throughput**

Duration: 5 min  
TFRC: No  
Thin: 5  
Greedy: 5  
ITT: 50.5 ms  
RTT: 150 ms  
QLEN: 63p  
Payload: 120 B  

---

**Latency**

Duration: 5 min  
TFRC: No  
Thin: 5  
Greedy: 5  
ITT: 50.5 ms  
RTT: 150 ms  
QLEN: 63p  
Payload: 120 B  

---

**Note:** The diagrams illustrate the throughput and latency performance of various protocols and configurations over time, with specifics such as ITT, RTT, QLEN, and payload size, as well as packet loss rates and other metrics.
A.2. Latency tests with greedy cross traffic
Figure A.2.19

**Goodput**
Duration: 5 min  TFRC: No  Thin: 5  Greedy: 5  
ITT: 75.7 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP variation used for competing streams

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>usender</td>
<td>5</td>
<td>74 / 74 / 74</td>
<td>120/120/121</td>
<td>71/8/8/10</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>12 / 13 / 14</td>
<td>1446/1447/1447</td>
<td>21/7/16/20/24/51</td>
</tr>
</tbody>
</table>

**Throughput**
Duration: 5 min  TFRC: No  Thin: 5  Greedy: 5  
ITT: 75.7 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP variation used for competing streams

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>usender</td>
<td>5</td>
<td>75 / 75 / 75 474/474/474</td>
<td>7/9/8/8/10</td>
<td></td>
</tr>
</tbody>
</table>

**Latency**
Duration: 5 min  TFRC: No  Thin: 5  Greedy: 5  
ITT: 75.7 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP variation used for competing streams

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>usender</td>
<td>5</td>
<td>75 / 75 / 75 474/474/474</td>
<td>15/5/9/11/16/85</td>
<td></td>
</tr>
</tbody>
</table>

Figure A.2.20

**Goodput**
Duration: 5 min  TFRC: Yes  Thin: 5  Greedy: 5  
ITT: 75.7 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP variation used for competing streams

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>usender</td>
<td>5</td>
<td>74 / 74 / 74</td>
<td>120/120/121</td>
<td>71/8/8/10</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>12 / 13 / 14</td>
<td>1446/1447/1447</td>
<td>21/7/16/20/24/47</td>
</tr>
</tbody>
</table>

**Throughput**
Duration: 5 min  TFRC: Yes  Thin: 5  Greedy: 5  
ITT: 75.7 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP variation used for competing streams

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>usender</td>
<td>5</td>
<td>75 / 75 / 75 474/474/474</td>
<td>15/5/9/11/16/85</td>
<td></td>
</tr>
</tbody>
</table>

**Latency**
Duration: 5 min  TFRC: Yes  Thin: 5  Greedy: 5  
ITT: 75.7 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP variation used for competing streams

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>usender</td>
<td>5</td>
<td>75 / 75 / 75 474/474/474</td>
<td>15/5/9/11/16/85</td>
<td></td>
</tr>
</tbody>
</table>
Figure A.2.21

Goodput

- Name: wsender
  - Streams: 10
  - ITT: 10 ms
  - RTT: 150 ms
  - QLEN: 63
  - Payload: 120 B

- Name: zsender
  - Streams: 5
  - ITT: 10 ms
  - RTT: 150 ms
  - QLEN: 63
  - Payload: 120 B

Throughput

- Name: wsender
  - Streams: 10
  - ITT: 10 ms
  - RTT: 150 ms
  - QLEN: 63
  - Payload: 120 B

- Name: zsender
  - Streams: 5
  - ITT: 10 ms
  - RTT: 150 ms
  - QLEN: 63
  - Payload: 120 B

Latency

- Name: wsender
  - Streams: 10
  - ITT: 10 ms
  - RTT: 150 ms
  - QLEN: 63
  - Payload: 120 B

- Name: zsender
  - Streams: 5
  - ITT: 10 ms
  - RTT: 150 ms
  - QLEN: 63
  - Payload: 120 B

Figure A.2.22
Appendix A. Experiments results

Figure A.2.23

(a) Goodput

Duration: 5 min  TFRC: Yes  Thin: 10  Greedy: 5
ITT: 10:1 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP ER+TLP  RDB DPIF10 ER+TLP  Greedy

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>10</td>
<td>39 / 44 / 46</td>
<td>300571903</td>
<td>7157923</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>34 / 35 / 37</td>
<td>144614461446</td>
<td>8167932</td>
</tr>
</tbody>
</table>

(b) Throughput

Duration: 5 min  TFRC: Yes  Thin: 10  Greedy: 5
ITT: 10:1 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP ER+TLP  RDB DPIF10 ER+TLP  Greedy

(c) Latency

Duration: 5 min  TFRC: Yes  Thin: 10  Greedy: 5
ITT: 10:1 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

Figure A.2.24

(a) Goodput

Duration: 5 min  TFRC: No  Thin: 10  Greedy: 5
ITT: 10:1 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP ER+TLP  RDB DPIF20 ER+TLP  Greedy

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>10</td>
<td>34 / 35 / 36</td>
<td>4384446883</td>
<td>8157927</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>28 / 29 / 32</td>
<td>144614461446</td>
<td>8167939</td>
</tr>
</tbody>
</table>

(b) Throughput

Duration: 5 min  TFRC: No  Thin: 10  Greedy: 5
ITT: 10:1 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP ER+TLP  RDB DPIF20 ER+TLP  Greedy

(c) Latency

Duration: 5 min  TFRC: No  Thin: 10  Greedy: 5
ITT: 10:1 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

(Continued...)


166
A.2. Latency tests with greedy TCP variation used for competing streams

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>10</td>
<td>28/33/33</td>
<td>367/396/435</td>
<td>9/1/6/9/15/25</td>
</tr>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>10</td>
<td>25/28/31</td>
<td>367/396/435</td>
<td>9/1/6/9/15/25</td>
</tr>
<tr>
<td>TCP ER+TLP</td>
<td>rtsender</td>
<td>10</td>
<td>25/31/34</td>
<td>528/589/637</td>
<td>10/9/6/13/25</td>
</tr>
<tr>
<td>TCP ER+TLP</td>
<td>zsender</td>
<td>5</td>
<td>25/28/31</td>
<td>367/396/435</td>
<td>9/1/6/9/15/25</td>
</tr>
</tbody>
</table>

**Figure A.2.26**

**Figure A.2.25**
Figure A.2.27

Figure A.2.28

Appendix A. Experiments results
TCP variation used for competing streams

Name    Host   Streams  ITT (avg)  Payload (avg)  CWND
TCP ER+TLP  wsender  10  31 / 32 / 33  130/150/155  9/11/15/11/13
RDB DPIF20 ER+TLP  rbdsender  10  31 / 32 / 33  561/581/527  9/11/15/11/13
Greedy  zsender  5  19 / 20 / 21  1447/1447/1444  10/14/11/13

(a)

TCP variation used for competing streams

Name    Host   Streams  ITT (avg)  Payload (avg)  CWND
TCP ER+TLP  wsender  10  35 / 36 / 37  143/148/153  8/11/16/11/13
RDB DPIF20 ER+TLP  rbdsender  10  33 / 34 / 35  943/454/454  9/11/15/11/13
Greedy  zsender  5  24 / 26 / 27  1446/1447/1447  9/27/8/10/43

(b)

TCP variation used for competing streams

Name    Host   Streams  ITT (avg)  Payload (avg)  CWND
TCP ER+TLP  wsender  10  35 / 36 / 37  143/148/153  8/11/16/11/13
RDB DPIF20 ER+TLP  rbdsender  10  33 / 34 / 35  943/454/454  9/11/15/11/13
Greedy  zsender  5  24 / 26 / 27  1446/1447/1447  9/27/8/10/43

(c)
Appendix A. Experiments results

**Figure A.2.31**

- **Goodput**
  - Duration: 5 min
  - TFRC: No
  - Thin: 10
  - Greedy: 5
  - ITT: 50.5 ms
  - RTT: 150 ms
  - QLEN: 63p
  - Payload: 120 B

- **Throughput**
  - Duration: 5 min
  - TFRC: No
  - Thin: 10
  - Greedy: 5
  - ITT: 50.5 ms
  - RTT: 150 ms
  - QLEN: 63p
  - Payload: 120 B

- **Latency**
  - Duration: 5 min
  - TFRC: No
  - Thin: 10
  - Greedy: 5
  - ITT: 50.5 ms
  - RTT: 150 ms
  - QLEN: 63p
  - Payload: 120 B

**Figure A.2.32**

- **Goodput**
  - Duration: 5 min
  - TFRC: No
  - Thin: 10
  - Greedy: 5
  - ITT: 50.5 ms
  - RTT: 150 ms
  - QLEN: 63p
  - Payload: 120 B

- **Throughput**
  - Duration: 5 min
  - TFRC: No
  - Thin: 10
  - Greedy: 5
  - ITT: 50.5 ms
  - RTT: 150 ms
  - QLEN: 63p
  - Payload: 120 B

- **Latency**
  - Duration: 5 min
  - TFRC: No
  - Thin: 10
  - Greedy: 5
  - ITT: 50.5 ms
  - RTT: 150 ms
  - QLEN: 63p
  - Payload: 120 B
Figure A.2.33

Figure A.2.34
Figure A.2.35

Goodput
Duration: 5 min  TFRC: Yes  Thin: 10  Greedy: 5
ITT: 75.7 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

Throughput
Duration: 5 min  TFRC: Yes  Thin: 10  Greedy: 5
ITT: 75.7 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

Latency
Duration: 5 min  TFRC: Yes  Thin: 10  Greedy: 5
ITT: 75.7 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

Figure A.2.36

Goodput
Duration: 5 min  TFRC: No  Thin: 10  Greedy: 5
ITT: 75.7 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

Throughput
Duration: 5 min  TFRC: No  Thin: 10  Greedy: 5
ITT: 75.7 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

Latency
Duration: 5 min  TFRC: No  Thin: 10  Greedy: 5
ITT: 75.7 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B
**Goodput**

Duration: 5 min  TFRC: No  Thin: 10  Greedy: 5
ITT: 75.7 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP ER+TLP  RDB DPIF20 ER+TLP  Greedy

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>10</td>
<td>74 / 74 / 74</td>
<td>120/120/121</td>
<td>7/1/7/8/10</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>13 / 14 / 15</td>
<td>1447/1447/1447</td>
<td>18/4/14/17/20/55</td>
</tr>
</tbody>
</table>

**Throughput**

Duration: 5 min  TFRC: No  Thin: 10  Greedy: 5
ITT: 75.7 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP ER+TLP  RDB DPIF20 ER+TLP  Greedy

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>10</td>
<td>74 / 74 / 74</td>
<td>120/120/121</td>
<td>7/1/7/8/10</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>13 / 14 / 15</td>
<td>1447/1447/1447</td>
<td>18/4/14/17/20/55</td>
</tr>
</tbody>
</table>

**Latency**

Duration: 5 min  TFRC: Y es  Thin: 10  Greedy: 5
ITT: 75.7 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP ER+TLP  RDB DPIF20 ER+TLP  Greedy

<table>
<thead>
<tr>
<th>Sent p</th>
<th>Lost p</th>
<th>Retr p</th>
<th>RDB p</th>
<th>Payl B</th>
<th>Unique B</th>
<th>Rdn B</th>
<th>Retr B</th>
<th>Loss % (B/p)</th>
<th>DupACKs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>40.6 K</td>
<td>498</td>
<td>502</td>
<td>0</td>
<td>4.9 M</td>
<td>4.8 M  60.2 K</td>
<td>60.2 K</td>
</tr>
<tr>
<td>ITT: 75:7 ms</td>
<td>RTT: 150 ms</td>
<td>QLEN: 63p</td>
<td>Payload: 120 B</td>
<td>1.22 / 1.22</td>
<td>2268</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39.6 K</td>
<td>478</td>
<td>11</td>
<td>39.5 K</td>
<td>17.9 M</td>
<td>4.8 M</td>
<td>13.2 M</td>
<td>240</td>
<td>1.26 / 1.21</td>
<td>2</td>
</tr>
<tr>
<td>107 K</td>
<td>1163</td>
<td>0</td>
<td>155 M</td>
<td>153 M</td>
<td>1.7 M</td>
<td>1.7 M</td>
<td>1.09 / 1.09</td>
<td>13619</td>
<td></td>
</tr>
</tbody>
</table>

**Figure A.2.39**

**Figure A.2.40**
Goodput (Kbit/second aggregated over 1 second)

RDB DPIF10 ER+TLP

1000

5000

2000

3000

4000

5000

TCP ER+TLP

0

Greedy

Name

TCP ER+TLP

wsender

16

40 / 43 / 46

519/569/608

6/1/4/5/23

Greedy

zsender

5

37 / 39 / 43

1146/1346/1446

6/1/4/5/23

Duration: 5 min   TFRC: No   Thin: 16   Greedy: 5

ITT: 10:1 ms   RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP variation used for competing streams

throughput (Kbit/second aggregated over 1 second)

1000

2000

3000

4000

5000

TCP ER+TLP

0

Greedy

Name

TCP ER+TLP

wsender

16

40 / 43 / 46

519/569/608

6/1/4/5/23

Greedy

zsender

5

37 / 39 / 43

1146/1346/1446

6/1/4/5/23

Duration: 5 min   TFRC: No   Thin: 16   Greedy: 5

ITT: 10:1 ms   RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP variation used for competing streams

Figure A.2.41

Throughput

RDB DPIF10 ER+TLP

1000

56 / 65 / 75

516 / 575 / 635

6/1/4/5/23

Greedy

zsender

5

44 / 49 / 54

1146 / 1346 / 1446

6/1/4/5/23

Duration: 5 min   TFRC: No   Thin: 16   Greedy: 5

ITT: 10:1 ms   RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP variation used for competing streams

Figure A.2.42

Latency

Mean 50th 60th 70th 80th 90th 95th 99th Max

TCP ER+TLP

0.0

768

674

475

586

563

647

670

747

922

1718

3405

2548

300

225

230

241

416

489

612

644

662

693

850

955

1524

2100

7554

TCP variation used for competing streams

ACK Latency in milliseconds

99.9th

2577

3414

4249

Max

99.99th

7668

8363

16540

TCP variation used for competing streams

Figure A.2.41

Throughput

RDB DPIF10 ER+TLP

1000

56 / 65 / 75

516 / 575 / 635

6/1/4/5/23

Greedy

zsender

5

44 / 49 / 54

1146 / 1346 / 1446

6/1/4/5/23

Duration: 5 min   TFRC: No   Thin: 16   Greedy: 5

ITT: 10:1 ms   RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP variation used for competing streams

Figure A.2.42

Latency

Mean 50th 60th 70th 80th 90th 95th 99th Max

TCP ER+TLP

0.0

768

674

475

586

563

647

670

747

922

1718

3405

2548

300

225

230

241

416

489

612

644

662

693

850

955

1524

2100

7554

TCP variation used for competing streams

ACK Latency in milliseconds

99.9th

2577

3414

4249

Max

99.99th

7668

8363

16540

TCP variation used for competing streams
Goodput (Kbit/second aggregated over 1 second)

TCP variation used for competing streams

Name | Host | Streams | ITT (avg) | Payload (avg) | CWND
--- | --- | --- | --- | --- | ---
TCP ER+TLP | wsender | 16 | 58 / 64 / 71 | 785/840/945 | 6/1/4/5/7/42
Greedy | zsender | 5 | 57 / 60 / 66 | 1444/1445/1447 | 5/1/4/5/6/21

Figure A.2.43

Throughput

Sent p | Lost p | Retr p | RDB p | PAYL B | Unique B | Rlb B | Retr B | Loss % (B/p) | DupACKs
--- | --- | --- | --- | --- | --- | --- | --- | --- | ---
TCP ER+TLP | 0 | 518 | 120 | 120 | 120 | 120 | 120 | 120 | 120
Greedy | 5 | 249 | 120 | 120 | 120 | 120 | 120 | 120 | 120

Figure A.2.44

Throughput

Sent p | Lost p | Retr p | RDB p | PAYL B | Unique B | Rlb B | Retr B | Loss % (B/p) | DupACKs
--- | --- | --- | --- | --- | --- | --- | --- | --- | ---
TCP ER+TLP | 0 | 19448 | 120 | 120 | 120 | 120 | 120 | 120 | 120
Greedy | 5 | 1904 | 120 | 120 | 120 | 120 | 120 | 120 | 120

Figure A.2.43

Latency

Mean | 50th | 60th | 70th | 80th | 90th | 95th | 99th | 99.9th | 99.99th | Max
--- | --- | --- | --- | --- | --- | --- | --- | --- | --- | ---
TCP ER+TLP | 387 | 261 | 267 | 250 | 271 | 278 | 285 | 293 | 297 | 308
Greedy | 361 | 260 | 264 | 269 | 292 | 305 | 310 | 314 | 317 | 322

Max
--- | --- | --- | --- | --- | --- | --- | --- | --- | --- | ---
TCP ER+TLP | 1227 | 704 | 733 | 751 | 760 | 763 | 765 | 767 | 769 | 771
Greedy | 1241 | 711 | 730 | 748 | 756 | 759 | 761 | 763 | 765 | 767

Max
Figure A.2.47

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
<th>Throughput</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>16</td>
<td>43 / 46 / 48</td>
<td>914 / 919 / 920</td>
<td>6 / 1 / 5</td>
<td>46.0 K</td>
<td>66.6 M</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>32 / 34 / 34</td>
<td>1446 / 1447 / 1447</td>
<td>6 / 1 / 5</td>
<td>46.0 K</td>
<td>66.6 M</td>
</tr>
</tbody>
</table>

Figure A.2.48

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
<th>Throughput</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>16</td>
<td>43 / 46 / 48</td>
<td>914 / 919 / 920</td>
<td>6 / 1 / 5</td>
<td>46.0 K</td>
<td>66.6 M</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>32 / 34 / 34</td>
<td>1446 / 1447 / 1447</td>
<td>6 / 1 / 5</td>
<td>46.0 K</td>
<td>66.6 M</td>
</tr>
</tbody>
</table>
Figure A.2.49

Figure A.2.50

A.2. Latency tests with greedy cross traffic
Figure A.2.51

Figure A.2.52
**Figure A.2.53**

**Goodput**

Duration: 5 min  
TFRC: Yes  
Thin: 16  
Greedy: 5  
ITT: 50.5 ms  
RTT: 150 ms  
QLEN: 63p  
Payload: 120 B

TCP ER+TLP  
RDB DPIF20 ER+TLP  
Greedy

**Throughput**

Duration: 5 min  
TFRC: Yes  
Thin: 16  
Greedy: 5  
ITT: 50.5 ms  
RTT: 150 ms  
QLEN: 63p  
Payload: 120 B

TCP ER+TLP  
RDB DPIF20 ER+TLP  
Greedy

**Latency**

Duration: 5 min  
TFRC: Yes  
Thin: 16  
Greedy: 5  
ITT: 50.5 ms  
RTT: 150 ms  
QLEN: 63p  
Payload: 120 B

TCP ER+TLP  
RDB DPIF20 ER+TLP  
Greedy

Name | Host | Streams | ITT (avg) | Payload (avg) | CWND | Rate (Kbit/second aggregated over 1 second)
--- | --- | --- | --- | --- | --- | ---
TCP ER+TLP | wsender | 16 | 51 / 52 / 53 | 123 / 126 / 129 | 5 / 5 / 5 | 0.0 / 0.0 / 0.0
RDB DPIF20 ER+TLP | zsender | 5 | 19 / 20 / 21 | 1447 / 1447 / 1447 | 11 / 39 / 11 | 0.0 / 0.0 / 0.0
    
**Figure A.2.54**

**Goodput**

Duration: 5 min  
TFRC: No  
Thin: 16  
Greedy: 5  
ITT: 50.5 ms  
RTT: 150 ms  
QLEN: 63p  
Payload: 120 B

TCP ER+TLP  
RDB DPIF20 ER+TLP  
Greedy

**Throughput**

Duration: 5 min  
TFRC: No  
Thin: 16  
Greedy: 5  
ITT: 50.5 ms  
RTT: 150 ms  
QLEN: 63p  
Payload: 120 B

TCP ER+TLP  
RDB DPIF20 ER+TLP  
Greedy

**Latency**

Duration: 5 min  
TFRC: No  
Thin: 16  
Greedy: 5  
ITT: 50.5 ms  
RTT: 150 ms  
QLEN: 63p  
Payload: 120 B

TCP ER+TLP  
RDB DPIF20 ER+TLP  
Greedy

Name | Host | Streams | ITT (avg) | Payload (avg) | CWND | Rate (Kbit/second aggregated over 1 second)
--- | --- | --- | --- | --- | --- | ---
TCP ER+TLP | wsender | 16 | 51 / 52 / 54 | 124 / 127 / 133 | 7 / 9 / 9 | 0.0 / 0.0 / 0.0
RDB DPIF20 ER+TLP | zsender | 5 | 18 / 19 / 20 | 1447 / 1447 / 1447 | 11 / 39 / 11 | 0.0 / 0.0 / 0.0
Figure A.2.57

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>16</td>
<td>74 / 74 / 74</td>
<td>120 / 120 / 121</td>
<td>7 / 1 / 8 / 10</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>15 / 15 / 16</td>
<td>1447 / 1447 / 1448</td>
<td>15 / 5 / 12 / 14 / 17 / 42</td>
</tr>
</tbody>
</table>

(a)

Figure A.2.58

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>16</td>
<td>74 / 74 / 74</td>
<td>120 / 120 / 121</td>
<td>7 / 1 / 8 / 10</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>15 / 15 / 16</td>
<td>1447 / 1447 / 1448</td>
<td>15 / 5 / 12 / 14 / 17 / 42</td>
</tr>
</tbody>
</table>

(a)
**Figure A.2.59**

**Figure A.2.60**
## A.3 Latency tests with greedy cross traffic bundle limitation

### Table A.3

<table>
<thead>
<tr>
<th>Figure A.3.1</th>
<th>Type Streams Cong TFRC RTT ITT Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure A.3.2</td>
<td>TCP ER+TLP 5 Rdb No 150 10:1 120</td>
</tr>
<tr>
<td>Figure A.3.3</td>
<td>RDB DPIF10 ER+TLP 5 Rdb No 150 10:1 120</td>
</tr>
<tr>
<td>Figure A.3.4</td>
<td>RDB DPIF20 ER+TLP 5 Rdb No 150 10:1 120</td>
</tr>
<tr>
<td>Figure A.3.5</td>
<td>RDB DPIF20 ER+TLP 5 Rdb Yes 150 10:1 120</td>
</tr>
<tr>
<td>Figure A.3.6</td>
<td>TCP ER+TLP 5 Rdb No 150 30:3 120</td>
</tr>
<tr>
<td>Figure A.3.7</td>
<td>RDB DPIF10 ER+TLP 5 Rdb No 150 30:3 120</td>
</tr>
<tr>
<td>Figure A.3.8</td>
<td>RDB DPIF10 ER+TLP 5 Rdb Yes 150 30:3 120</td>
</tr>
<tr>
<td>Figure A.3.9</td>
<td>RDB DPIF20 ER+TLP 5 Rdb No 150 30:3 120</td>
</tr>
<tr>
<td>Figure A.3.10</td>
<td>RDB DPIF20 ER+TLP 5 Rdb Yes 150 30:3 120</td>
</tr>
<tr>
<td>Figure A.3.11</td>
<td>TCP ER+TLP 5 Rdb No 150 50:5 120</td>
</tr>
<tr>
<td>Figure A.3.12</td>
<td>RDB DPIF10 ER+TLP 5 Rdb No 150 50:5 120</td>
</tr>
<tr>
<td>Figure A.3.13</td>
<td>RDB DPIF10 ER+TLP 5 Rdb Yes 150 50:5 120</td>
</tr>
<tr>
<td>Figure A.3.14</td>
<td>RDB DPIF20 ER+TLP 5 Rdb No 150 50:5 120</td>
</tr>
<tr>
<td>Figure A.3.15</td>
<td>RDB DPIF20 ER+TLP 5 Rdb Yes 150 50:5 120</td>
</tr>
<tr>
<td>Figure A.3.16</td>
<td>TCP ER+TLP 5 Rdb No 150 75:7 120</td>
</tr>
<tr>
<td>Figure A.3.17</td>
<td>RDB DPIF10 ER+TLP 5 Rdb No 150 75:7 120</td>
</tr>
<tr>
<td>Figure A.3.18</td>
<td>RDB DPIF10 ER+TLP 5 Rdb Yes 150 75:7 120</td>
</tr>
<tr>
<td>Figure A.3.19</td>
<td>RDB DPIF20 ER+TLP 5 Rdb No 150 75:7 120</td>
</tr>
<tr>
<td>Figure A.3.20</td>
<td>RDB DPIF20 ER+TLP 5 Rdb Yes 150 75:7 120</td>
</tr>
<tr>
<td>Figure A.3.21</td>
<td>TCP ER+TLP 5 Rdb No 150 100:10 120</td>
</tr>
<tr>
<td>Figure A.3.22</td>
<td>RDB DPIF10 ER+TLP 5 Rdb No 150 100:10 120</td>
</tr>
<tr>
<td>Figure A.3.23</td>
<td>RDB DPIF10 ER+TLP 5 Rdb Yes 150 100:10 120</td>
</tr>
<tr>
<td>Figure A.3.24</td>
<td>RDB DPIF20 ER+TLP 5 Rdb No 150 100:10 120</td>
</tr>
<tr>
<td>Figure A.3.25</td>
<td>RDB DPIF20 ER+TLP 5 Rdb Yes 150 100:10 120</td>
</tr>
<tr>
<td>Figure A.3.26</td>
<td>TCP ER+TLP 10 Rdb No 150 10:1 120</td>
</tr>
<tr>
<td>Figure A.3.27</td>
<td>RDB DPIF10 ER+TLP 10 Rdb No 150 10:1 120</td>
</tr>
<tr>
<td>Figure A.3.28</td>
<td>RDB DPIF10 ER+TLP 10 Rdb Yes 150 10:1 120</td>
</tr>
<tr>
<td>Figure A.3.29</td>
<td>RDB DPIF20 ER+TLP 10 Rdb No 150 10:1 120</td>
</tr>
<tr>
<td>Figure A.3.30</td>
<td>RDB DPIF20 ER+TLP 10 Rdb Yes 150 10:1 120</td>
</tr>
<tr>
<td>Figure A.3.31</td>
<td>TCP ER+TLP 10 Rdb No 150 30:3 120</td>
</tr>
<tr>
<td>Figure A.3.32</td>
<td>RDB DPIF10 ER+TLP 10 Rdb No 150 30:3 120</td>
</tr>
<tr>
<td>Figure A.3.33</td>
<td>RDB DPIF10 ER+TLP 10 Rdb Yes 150 30:3 120</td>
</tr>
<tr>
<td>Figure A.3.34</td>
<td>RDB DPIF20 ER+TLP 10 Rdb No 150 30:3 120</td>
</tr>
<tr>
<td>Figure A.3.35</td>
<td>RDB DPIF20 ER+TLP 10 Rdb Yes 150 30:3 120</td>
</tr>
<tr>
<td>Figure A.3.36</td>
<td>TCP ER+TLP 10 Rdb No 150 50:5 120</td>
</tr>
<tr>
<td>Figure A.3.37</td>
<td>RDB DPIF10 ER+TLP 10 Rdb No 150 50:5 120</td>
</tr>
<tr>
<td>Figure A.3.38</td>
<td>RDB DPIF10 ER+TLP 10 Rdb Yes 150 50:5 120</td>
</tr>
<tr>
<td>Figure A.3.39</td>
<td>RDB DPIF20 ER+TLP 10 Rdb No 150 50:5 120</td>
</tr>
<tr>
<td>Figure A.3.40</td>
<td>RDB DPIF20 ER+TLP 10 Rdb Yes 150 50:5 120</td>
</tr>
<tr>
<td>Figure A.3.41</td>
<td>TCP ER+TLP 10 Rdb No 150 75:7 120</td>
</tr>
<tr>
<td>Figure A.3.42</td>
<td>RDB DPIF10 ER+TLP 10 Rdb No 150 75:7 120</td>
</tr>
<tr>
<td>Figure A.3.43</td>
<td>RDB DPIF10 ER+TLP 10 Rdb Yes 150 75:7 120</td>
</tr>
<tr>
<td>Figure A.3.44</td>
<td>RDB DPIF20 ER+TLP 10 Rdb No 150 75:7 120</td>
</tr>
<tr>
<td>Figure</td>
<td>Details</td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
</tr>
<tr>
<td>A.3.45</td>
<td>RDB DPIF20 ER+TLP 10  Rdb Yes 150 75:7 120</td>
</tr>
<tr>
<td>A.3.46</td>
<td>TCP ER+TLP 10  Rdb No 150 100:10 120</td>
</tr>
<tr>
<td>A.3.47</td>
<td>RDB DPIF10 ER+TLP 10  Rdb No 150 100:10 120</td>
</tr>
<tr>
<td>A.3.48</td>
<td>RDB DPIF10 ER+TLP 10  Rdb Yes 150 100:10 120</td>
</tr>
<tr>
<td>A.3.49</td>
<td>RDB DPIF20 ER+TLP 10  Rdb No 150 100:10 120</td>
</tr>
<tr>
<td>A.3.50</td>
<td>TCP ER+TLP 13  Rdb No 150 10:1 120</td>
</tr>
<tr>
<td>A.3.51</td>
<td>RDB DPIF10 ER+TLP 13  Rdb Yes 150 10:1 120</td>
</tr>
<tr>
<td>A.3.52</td>
<td>RDB DPIF20 ER+TLP 13  Rdb No 150 10:1 120</td>
</tr>
<tr>
<td>A.3.53</td>
<td>RDB DPIF20 ER+TLP 13  Rdb Yes 150 10:1 120</td>
</tr>
<tr>
<td>A.3.54</td>
<td>RDB DPIF10 ER+TLP 13  Rdb No 150 10:1 120</td>
</tr>
<tr>
<td>A.3.55</td>
<td>TCP ER+TLP 13  Rdb No 150 10:1 120</td>
</tr>
<tr>
<td>A.3.56</td>
<td>RDB DPIF10 ER+TLP 13  Rdb No 150 30:3 120</td>
</tr>
<tr>
<td>A.3.57</td>
<td>RDB DPIF10 ER+TLP 13  Rdb No 150 30:3 120</td>
</tr>
<tr>
<td>A.3.58</td>
<td>RDB DPIF10 ER+TLP 13  Rdb Yes 150 30:3 120</td>
</tr>
<tr>
<td>A.3.59</td>
<td>RDB DPIF20 ER+TLP 13  Rdb No 150 30:3 120</td>
</tr>
<tr>
<td>A.3.60</td>
<td>RDB DPIF20 ER+TLP 13  Rdb Yes 150 30:3 120</td>
</tr>
<tr>
<td>A.3.61</td>
<td>TCP ER+TLP 13  Rdb No 150 50:5 120</td>
</tr>
<tr>
<td>A.3.62</td>
<td>RDB DPIF10 ER+TLP 13  Rdb No 150 50:5 120</td>
</tr>
<tr>
<td>A.3.63</td>
<td>RDB DPIF10 ER+TLP 13  Rdb Yes 150 50:5 120</td>
</tr>
<tr>
<td>A.3.64</td>
<td>RDB DPIF20 ER+TLP 13  Rdb No 150 50:5 120</td>
</tr>
<tr>
<td>A.3.65</td>
<td>RDB DPIF20 ER+TLP 13  Rdb Yes 150 50:5 120</td>
</tr>
<tr>
<td>A.3.66</td>
<td>TCP ER+TLP 13  Rdb No 150 75:7 120</td>
</tr>
<tr>
<td>A.3.67</td>
<td>RDB DPIF10 ER+TLP 13  Rdb No 150 75:7 120</td>
</tr>
<tr>
<td>A.3.68</td>
<td>RDB DPIF10 ER+TLP 13  Rdb Yes 150 75:7 120</td>
</tr>
<tr>
<td>A.3.69</td>
<td>RDB DPIF20 ER+TLP 13  Rdb No 150 75:7 120</td>
</tr>
<tr>
<td>A.3.70</td>
<td>TCP ER+TLP 13  Rdb No 150 75:7 120</td>
</tr>
<tr>
<td>A.3.71</td>
<td>RDB DPIF10 ER+TLP 13  Rdb No 150 100:10 120</td>
</tr>
<tr>
<td>A.3.72</td>
<td>RDB DPIF10 ER+TLP 13  Rdb No 150 100:10 120</td>
</tr>
<tr>
<td>A.3.73</td>
<td>RDB DPIF10 ER+TLP 13  Rdb Yes 150 100:10 120</td>
</tr>
<tr>
<td>A.3.74</td>
<td>RDB DPIF20 ER+TLP 13  Rdb No 150 100:10 120</td>
</tr>
<tr>
<td>A.3.75</td>
<td>RDB DPIF20 ER+TLP 13  Rdb Yes 150 100:10 120</td>
</tr>
<tr>
<td>A.3.76</td>
<td>TCP ER+TLP 16  Rdb No 150 10:1 120</td>
</tr>
<tr>
<td>A.3.77</td>
<td>RDB DPIF10 ER+TLP 16  Rdb No 150 10:1 120</td>
</tr>
<tr>
<td>A.3.78</td>
<td>RDB DPIF10 ER+TLP 16  Rdb Yes 150 10:1 120</td>
</tr>
<tr>
<td>A.3.79</td>
<td>RDB DPIF20 ER+TLP 16  Rdb No 150 10:1 120</td>
</tr>
<tr>
<td>A.3.80</td>
<td>TCP ER+TLP 16  Rdb No 150 10:1 120</td>
</tr>
<tr>
<td>A.3.81</td>
<td>RDB DPIF20 ER+TLP 16  Rdb Yes 150 10:1 120</td>
</tr>
<tr>
<td>A.3.82</td>
<td>TCP ER+TLP 16  Rdb No 150 30:3 120</td>
</tr>
<tr>
<td>A.3.83</td>
<td>RDB DPIF10 ER+TLP 16  Rdb Yes 150 30:3 120</td>
</tr>
<tr>
<td>A.3.84</td>
<td>RDB DPIF20 ER+TLP 16  Rdb No 150 30:3 120</td>
</tr>
<tr>
<td>A.3.85</td>
<td>RDB DPIF20 ER+TLP 16  Rdb Yes 150 30:3 120</td>
</tr>
<tr>
<td>A.3.86</td>
<td>TCP ER+TLP 16  Rdb No 150 50:5 120</td>
</tr>
<tr>
<td>A.3.87</td>
<td>RDB DPIF10 ER+TLP 16  Rdb No 150 50:5 120</td>
</tr>
<tr>
<td>A.3.88</td>
<td>RDB DPIF10 ER+TLP 16  Rdb Yes 150 50:5 120</td>
</tr>
<tr>
<td>A.3.89</td>
<td>RDB DPIF20 ER+TLP 16  Rdb No 150 50:5 120</td>
</tr>
<tr>
<td>A.3.90</td>
<td>TCP ER+TLP 16  Rdb No 150 50:5 120</td>
</tr>
<tr>
<td>A.3.91</td>
<td>RDB DPIF20 ER+TLP 16  Rdb Yes 150 50:5 120</td>
</tr>
<tr>
<td>A.3.92</td>
<td>TCP ER+TLP 16  Rdb No 150 75:7 120</td>
</tr>
<tr>
<td>A.3.93</td>
<td>RDB DPIF10 ER+TLP 16  Rdb No 150 75:7 120</td>
</tr>
<tr>
<td>A.3.94</td>
<td>TCP ER+TLP 16  Rdb No 150 75:7 120</td>
</tr>
<tr>
<td>A.3.95</td>
<td>RDB DPIF20 ER+TLP 16  Rdb Yes 150 75:7 120</td>
</tr>
<tr>
<td>A.3.96</td>
<td>TCP ER+TLP 16  Rdb No 150 100:10 120</td>
</tr>
</tbody>
</table>
A.3. Latency tests with greedy cross traffic bundle limitation

<table>
<thead>
<tr>
<th>Figure</th>
<th>Configuration</th>
<th>Rdb</th>
<th>Bundle Limitation</th>
<th>Throughput</th>
<th>Bandwidth</th>
<th>Latency (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.3.97</td>
<td>RDB DPIF10 ER+TLP</td>
<td>16</td>
<td>No</td>
<td>150</td>
<td>100:10</td>
<td>120</td>
</tr>
<tr>
<td>A.3.98</td>
<td>RDB DPIF10 ER+TLP</td>
<td>16</td>
<td>Yes</td>
<td>150</td>
<td>100:10</td>
<td>120</td>
</tr>
<tr>
<td>A.3.99</td>
<td>RDB DPIF20 ER+TLP</td>
<td>16</td>
<td>No</td>
<td>150</td>
<td>100:10</td>
<td>120</td>
</tr>
<tr>
<td>A.3.100</td>
<td>RDB DPIF20 ER+TLP</td>
<td>16</td>
<td>Yes</td>
<td>150</td>
<td>100:10</td>
<td>120</td>
</tr>
</tbody>
</table>

Table A.3: Test setup for experiment 3
A.3. Latency tests with greedy cross traffic bundle limitation
Figure A.3.5

Throughput

Duration: 5 min
TFRC: Yes
Thin: 5
Greedy: 5
ITT: 10.1 ms
RTT: 150 ms
QLEN: 63p
Payload: 120 B

TCP ER+TLP
RDB DPIF20 ER+TLP
Greedy

(a)

TCP variation used for competing streams

Name | Host | Streams | ITT (avg) | Payload (avg) | CWND
--- | --- | --- | --- | --- | ---
TCP ER+TLP | wsender | 5 | 18 / 20 / 21 | 238 / 253 / 267 | 13 / 15 / 16 / 28
TCP ER+TLP | zsender | 5 | 19 / 20 / 21 | 317 / 333 / 352 | 14 / 15 / 16 / 28
Greedy | zsender | 5 | 16 / 16 / 17 / 1446 / 1447 / 1447 | 13 / 15 / 16 / 28 / 15 / 31

(b)

Latency

Duration: 5 min
TFRC: Yes
Thin: 5
Greedy: 5
ITT: 10.1 ms
RTT: 150 ms
QLEN: 63p
Payload: 120 B

TCP ER+TLP
RDB DPIF20 ER+TLP
Greedy

(c)
Figure A.3.8
A.3. Latency tests with greedy cross traffic bundle limitation
Figure A.3.11

Table A.3.12

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP wsender</td>
<td>5</td>
<td>50/50/50</td>
<td>121/121/122</td>
<td>9/9/9/10/10</td>
<td></td>
</tr>
<tr>
<td>TCP ER+TLP rbsender</td>
<td>5</td>
<td>50/50/50</td>
<td>121/121/122</td>
<td>9/9/9/10/10</td>
<td></td>
</tr>
<tr>
<td>Greedy zsender</td>
<td>5</td>
<td>11/12/13</td>
<td>1444/1446/1447</td>
<td>20/7/16/19/23/50</td>
<td></td>
</tr>
</tbody>
</table>

Appendix A. Experiments results

194
Appendix A. Experiments results

(a) Goodput
(b) Throughput
(c) Latency

Figure A.3.15

(a) Goodput
(b) Throughput
(c) Latency

Figure A.3.16
A.3. Latency tests with greedy cross traffic bundle limitation
### Goodput
Duration: 5 min  TFRC: No  Thin: 5  Greedy: 5

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>5</td>
<td>74 / 74 / 75</td>
<td>120 / 120 / 120</td>
<td>8 / 8 / 8 / 10</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>12 / 12 / 14</td>
<td>1446 / 1447 / 1447</td>
<td>21 / 17 / 20 / 24 / 47</td>
</tr>
</tbody>
</table>

### Throughput
Duration: 5 min  TFRC: No  Thin: 5  Greedy: 5

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>5</td>
<td>74 / 74 / 75</td>
<td>120 / 120 / 120</td>
<td>8 / 8 / 8 / 10</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>11 / 12 / 13</td>
<td>1446 / 1447 / 1447</td>
<td>21 / 17 / 20 / 24 / 49</td>
</tr>
</tbody>
</table>

### Latency
Duration: 5 min  TFRC: No  Thin: 5  Greedy: 5

- **Mean**: 265 / 256 / 259 / 262 / 265 / 268 / 270 / 277 / 355 / 468 / 565 / 650 / 768 / 956 / 1152
- **Max**: 253 / 255 / 259 / 261 / 264 / 265 / 267 / 268 / 269 / 270 / 271 / 275 / 477 / 766 / 1151

---

**Figure A.3.19**

**Figure A.3.20**
Figure A.3.21
Throughput
Duration: 5 min  TFRC: No  Thin: 5  Greedy: 5
ITT: 100:10 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP variation used for competing streams

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdbsender</td>
<td>99 / 99 / 99</td>
<td>120 / 120 / 120</td>
<td>6/16/7/10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Greedy</td>
<td>11 / 12 / 13</td>
<td>1446/1447/1447</td>
<td>22/17/21/25/51</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) Goodput
Duration: 5 min  TFRC: No  Thin: 5  Greedy: 5
ITT: 100:10 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP variation used for competing streams

(b) Throughput
Duration: 5 min  TFRC: No  Thin: 5  Greedy: 5
ITT: 100:10 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP variation used for competing streams

(c) Latency
Duration: 5 min  TFRC: No  Thin: 5  Greedy: 5
ITT: 100:10 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

Figure A.3.22
Throughput
Duration: 5 min  TFRC: No  Thin: 5  Greedy: 5
ITT: 100:10 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP variation used for competing streams

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>98 / 98 / 98</td>
<td>120 / 120 / 120</td>
<td>7/17/7/10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Greedy</td>
<td>12 / 12 / 13</td>
<td>1446/1447/1447</td>
<td>22/17/21/25/51</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) Goodput
Duration: 5 min  TFRC: No  Thin: 5  Greedy: 5
ITT: 100:10 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP variation used for competing streams

(b) Throughput
Duration: 5 min  TFRC: No  Thin: 5  Greedy: 5
ITT: 100:10 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP variation used for competing streams

(c) Latency
Duration: 5 min  TFRC: No  Thin: 5  Greedy: 5
ITT: 100:10 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

Mean 50th 60th 70th 80th 90th 95th 99th 99.9th 99.99th Max:
274 263 266 269 272 276 286 417 499 557 624 711 829 1095 1151
274 262 266 269 271 274 276 333 425 517 612 707 822 1043 1151
287 265 269 271 279 381 461 503 519 532 544 568 749 835 1151
Figure A.3.23

Goodput
Duration: 5 min  TFRC: Yes  Thin: 5  Greedy: 5  ITT: 100:10 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B
TCP ER+TLP  RDB DPIF10 ER+TLP  Greedy

Throughput
Duration: 5 min  TFRC: Yes  Thin: 5  Greedy: 5  ITT: 100:10 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B
TCP ER+TLP  RDB DPIF10 ER+TLP  Greedy

Latency
Duration: 5 min  TFRC: Yes  Thin: 5  Greedy: 5  ITT: 100:10 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B
TCP ER+TLP  RDB DPIF10 ER+TLP  Greedy

Figure A.3.24

Goodput
Duration: 5 min  TFRC: No  Thin: 5  Greedy: 5  ITT: 100:10 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B
TCP ER+TLP  RDB DPIF20 ER+TLP  Greedy

Throughput
Duration: 5 min  TFRC: No  Thin: 5  Greedy: 5  ITT: 100:10 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B
TCP ER+TLP  RDB DPIF20 ER+TLP  Greedy

Latency
Duration: 5 min  TFRC: No  Thin: 5  Greedy: 5  ITT: 100:10 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B
TCP ER+TLP  RDB DPIF20 ER+TLP  Greedy
Figure A.3.25

A.3. Latency tests with greedy cross traffic bundle limitation

(a) Goodput
Duration: 5 min  TFRC: Yes  Thin: 5  Greedy: 5
ITT: 100:1 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP ER+TLP  RDB DPIF20 ER+TLP  Greedy

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP wsender</td>
<td>5</td>
<td>99 / 100 / 100</td>
<td>121/121/122</td>
<td>7/17/7/17/10</td>
<td></td>
</tr>
<tr>
<td>Greedy zsender</td>
<td>5</td>
<td>12 / 12 / 13</td>
<td>1447/1447/1447</td>
<td>22/17/21/25/44</td>
<td></td>
</tr>
</tbody>
</table>
Appendix A. Experiments Results

Figure A.3.27

(a) Goodput
Duration: 5 min  TFRC: No  Thin: 10  Greedy: 5
ITT: 10.1 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

(b) Throughput
Duration: 5 min  TFRC: No  Thin: 10  Greedy: 5
ITT: 10.1 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

(c) Latency
Duration: 5 min  TFRC: No  Thin: 10  Greedy: 5
ITT: 10.1 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

Figure A.3.28

(a) Goodput
Duration: 5 min  TFRC: Yes  Thin: 10  Greedy: 5
ITT: 10.1 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

(b) Throughput
Duration: 5 min  TFRC: Yes  Thin: 10  Greedy: 5
ITT: 10.1 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

(c) Latency
Duration: 5 min  TFRC: Yes  Thin: 10  Greedy: 5
ITT: 10.1 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B
Figure A.3.29

A.3. Latency tests with greedy cross traffic bundle limitation

<table>
<thead>
<tr>
<th>Goodput</th>
<th>Throughput</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration: 5 min</td>
<td>TFRC: No</td>
<td>Thin: 10</td>
</tr>
<tr>
<td>ITT: 10.1 ms</td>
<td>RTT: 150 ms</td>
<td>QLEN: 63</td>
</tr>
</tbody>
</table>

TCP variation used for competing streams

TCP variation used for competing streams

TCP variation used for competing streams

Lost p Lost p Lost p Retr p Retr p Retr p Loss % (Bps) DupACKs
97.8 K 2585 2586 0 38.7 M 37.6 M 1.0 M 1.0 M 2.70 / 3.05 17395
90.2 K 3179 2114 36.5 K 49.0 M 37.1 M 12.4 M 1.1 M 3.55 / 3.32 13187
61.9 K 1871 1873 0 89.6 M 86.9 M 2.7 M 2.7 M 3.02 / 3.02 10936

ACK Latency in milliseconds

Mean 50th 60th 70th 80th 90th 95th 99th 99.9th 99.99th Max
261 219 223 225 251 428 452 476 501 560 610 659 1132 1766 2558
252 218 222 227 236 403 435 456 486 492 570 647 1117 1618 2644
264 223 228 236 278 432 454 476 494 520 567 628 859 1151 1483

TCP variation used for competing streams

TCP variation used for competing streams

TCP variation used for competing streams
The figure shows the results of experiments conducted with TCP ER+TLP and Greedy. The graphs illustrate the goodput, throughput, and latency for these two protocols.

- **Goodput**: The goodput is measured over 5 minutes. The ITT (Inter-Transmit-Time) is 30:3 ms, RTT (Round-Trip-Time) is 150 ms, and the QLEN (Queue Length) is 63 bytes. The payload size is 120 bytes. The protocol used is TCP ER+TLP.

- **Throughput**: The throughputs are measured in Kbit/second aggregated over 1 second. The mean values are presented for the following protocols:
  - TCP ER+TLP: 136.0 Mbit/s
  - Greedy: 24.0 Mbit/s

- **Latency**: The ACK latency is measured in milliseconds. The mean values are presented for the following protocols:
  - TCP ER+TLP: 134.0 Mbit/s
  - Greedy: 12.0 Mbit/s

**Table**:

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>10</td>
<td>31 / 31</td>
<td>128 / 128</td>
<td>125 / 125 / 125</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>15 / 16 / 17</td>
<td>144 / 144 / 144</td>
<td>14 / 14 / 14</td>
</tr>
</tbody>
</table>

**Figure A.3.31**

(a) Goodput graph for TCP ER+TLP and Greedy.

(b) Throughput graph for TCP ER+TLP and Greedy.

(c) Latency graph for TCP ER+TLP and Greedy.

**Figure A.3.32**

(a) Goodput graph for TCP ER+TLP and RDB DPIF10 ER+TLP and Greedy.

(b) Throughput graph for TCP ER+TLP and RDB DPIF10 ER+TLP and Greedy.

(c) Latency graph for TCP ER+TLP and RDB DPIF10 ER+TLP and Greedy.
Figure A.3.35

(a) Goodput
Duration: 5 min  TFRC: Yes  Thin: 10  Greedy: 5
ITT: 30.3 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B
TCP ER+TLP  RDB DPIF20 ER+TLP  Greedy

(b) Throughput
Duration: 5 min  TFRC: Yes  Thin: 10  Greedy: 5
ITT: 30.3 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B
TCP ER+TLP  RDB DPIF20 ER+TLP  Greedy

(c) Latency
Duration: 5 min  TFRC: Yes  Thin: 10  Greedy: 5
ITT: 30.3 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B
TCP ER+TLP  RDB DPIF20 ER+TLP  Greedy

Figure A.3.36

(a) Goodput
Duration: 5 min  TFRC: No  Thin: 10  Greedy: 5
ITT: 50.5 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B
TCP ER+TLP  TCP ER+TLP  Greedy

(b) Throughput
Duration: 5 min  TFRC: No  Thin: 10  Greedy: 5
ITT: 50.5 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B
TCP ER+TLP  TCP ER+TLP  Greedy

(c) Latency
Duration: 5 min  TFRC: No  Thin: 10  Greedy: 5
ITT: 50.5 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B
TCP ER+TLP  TCP ER+TLP  Greedy

Appendix A. Experiments results
A.3. Latency tests with greedy cross traffic bundle limitation

### Table A.3.37

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>10</td>
<td>50 / 50 / 50</td>
<td>122/122/123</td>
<td>8/1/8/9/10</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>13 / 14 / 15</td>
<td>1447/1447/1447</td>
<td>16/6/13/15/18/51</td>
</tr>
</tbody>
</table>

### Figure A.3.37

- **(a)** Goodput
- **(b)** Throughput
- **(c)** Latency

### Table A.3.38

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>10</td>
<td>50 / 50 / 50</td>
<td>122/122/123</td>
<td>8/1/8/9/10</td>
</tr>
<tr>
<td>RDB DPIF10 ER+TLP</td>
<td>rdbsender</td>
<td>10</td>
<td>51 / 51 / 52</td>
<td>242/242/242</td>
<td>8/1/8/9/10</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>13 / 14 / 15</td>
<td>1447/1447/1447</td>
<td>16/6/13/15/18/51</td>
</tr>
</tbody>
</table>

### Figure A.3.38

- **(a)** Goodput
- **(b)** Throughput
- **(c)** Latency
Appendix A. Experiments results
A.3. Latency tests with greedy cross traffic bundle limitation

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>sender</td>
<td>10</td>
<td>75/76/77</td>
<td>120/121/124</td>
<td>7/1</td>
</tr>
<tr>
<td>Greedy</td>
<td>sender</td>
<td>5</td>
<td>12/13/14</td>
<td>1447/1447/1447</td>
<td>1/15</td>
</tr>
</tbody>
</table>

**Figure A.3.41**

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>sender</td>
<td>10</td>
<td>75/76/77</td>
<td>120/121/124</td>
<td>7/1</td>
</tr>
<tr>
<td>Greedy</td>
<td>sender</td>
<td>5</td>
<td>12/13/14</td>
<td>1447/1447/1447</td>
<td>1/15</td>
</tr>
</tbody>
</table>

**Figure A.3.42**
Figure A.3.43

Figure A.3.44
A.3. Latency tests with greedy cross traffic bundle limitation

![Graphs and diagrams showing latency tests results with greedy cross traffic bundle limitation.](image-url)

### Table A.3.45

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>10</td>
<td>74 / 74 / 74</td>
<td>120 / 120 / 120</td>
<td>7 / 1 / 6 / 10</td>
</tr>
<tr>
<td>TCP ER+TLP</td>
<td>zsender</td>
<td>5</td>
<td>13 / 13 / 13</td>
<td>1446 / 1447 / 1447</td>
<td>18 / 14 / 17 / 12</td>
</tr>
<tr>
<td>Greedy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Graphs and diagrams showing latency tests results with greedy cross traffic bundle limitation.](image-url)

Figure A.3.45

### Table A.3.46

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>10</td>
<td>75 / 75 / 75</td>
<td>240 / 240 / 240</td>
<td>12 / 8 / 9 / 10</td>
</tr>
<tr>
<td>TCP ER+TLP</td>
<td>zsender</td>
<td>5</td>
<td>12 / 13 / 13</td>
<td>1446 / 1447 / 1447</td>
<td>20 / 16 / 19 / 23 / 24</td>
</tr>
<tr>
<td>Greedy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure A.3.47

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>10</td>
<td>98 / 99 / 99</td>
<td>120 / 120 / 120</td>
<td>6 / 1 / 7 / 11 / 10</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>11 / 13 / 13</td>
<td>144 / 144 / 144</td>
<td>20 / 15 / 19 / 23 / 4</td>
</tr>
</tbody>
</table>

Figure A.3.48

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>10</td>
<td>98 / 99 / 99</td>
<td>120 / 120 / 120</td>
<td>6 / 1 / 7 / 11 / 10</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>11 / 13 / 13</td>
<td>144 / 144 / 144</td>
<td>20 / 15 / 19 / 23 / 4</td>
</tr>
</tbody>
</table>
A.3. Latency tests with greedy cross traffic bundle limitation.
**Figure A.3.53**

**Goodput**
- Duration: 5 min
- TFRC: Yes
- Thin: 13
- Greedy: 5
- ITT: 10:1 ms
- RTT: 150 ms
- QLEN: 63p
- Payload: 120 B

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>Cwnd</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>13</td>
<td>36 / 39 / 42</td>
<td>604 / 676 / 735</td>
<td>8 / 1 / 57 / 78</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>33 / 35 / 38</td>
<td>1446 / 1447 / 1447</td>
<td>7 / 1 / 57 / 923</td>
</tr>
</tbody>
</table>

**Throughput**
- Duration: 5 min
- TFRC: Yes
- Thin: 13
- Greedy: 5
- ITT: 10:1 ms
- RTT: 150 ms
- QLEN: 63p
- Payload: 120 B

**Latency**
- Duration: 5 min
- TFRC: Yes
- Thin: 13
- Greedy: 5
- ITT: 10:1 ms
- RTT: 150 ms
- QLEN: 63p
- Payload: 120 B

**Figure A.3.54**

**Goodput**
- Duration: 5 min
- TFRC: No
- Thin: 13
- Greedy: 5
- ITT: 10:1 ms
- RTT: 150 ms
- QLEN: 63p
- Payload: 120 B

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>Cwnd</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>13</td>
<td>34 / 37 / 44</td>
<td>438 / 480 / 515</td>
<td>7 / 1 / 57 / 22</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>31 / 32 / 33</td>
<td>1446 / 1447 / 1447</td>
<td>8 / 1 / 57 / 927</td>
</tr>
</tbody>
</table>

**Throughput**
- Duration: 5 min
- TFRC: No
- Thin: 13
- Greedy: 5
- ITT: 10:1 ms
- RTT: 150 ms
- QLEN: 63p
- Payload: 120 B

**Latency**
- Duration: 5 min
- TFRC: No
- Thin: 13
- Greedy: 5
- ITT: 10:1 ms
- RTT: 150 ms
- QLEN: 63p
- Payload: 120 B

---

A.3. Latency tests with greedy cross traffic bundle limitation
A.3. Latency tests with greedy cross traffic bundle limitation
Figure A.3.63

(a) Goodput
Duration: 5 min  
ITF: Yes  Th: 13  Greedy: 5
TCP variation used for competing streams

(b) Throughput
Duration: 5 min  
ITF: Yes  Th: 13  Greedy: 5
TCP variation used for competing streams

(c) Latency
Duration: 5 min  
ITF: Yes  Th: 13  Greedy: 5
TCP variation used for competing streams

Figure A.3.64

(a) Goodput
Duration: 5 min  
ITF: No  Th: 13  Greedy: 5
TCP variation used for competing streams

(b) Throughput
Duration: 5 min  
ITF: No  Th: 13  Greedy: 5
TCP variation used for competing streams

(c) Latency
Duration: 5 min  
ITF: No  Th: 13  Greedy: 5
TCP variation used for competing streams
A.3. Latency tests with greedy cross traffic bundle limitation

![Figure A.3.65](image1)

![Figure A.3.66](image2)
Figure A.3.67

(a) Goodput
Duration: 5 min  TFRC: No  Thin: 13  Greedy: 5
ITT: 75.7 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

(b) Throughput
Duration: 5 min  TFRC: No  Thin: 13  Greedy: 5
ITT: 75.7 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

(c) Latency
Duration: 5 min  TFRC: No  Thin: 13  Greedy: 5
ITT: 75.7 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

Figure A.3.68

(a) Goodput
Duration: 5 min  TFRC: Yes  Thin: 13  Greedy: 5
ITT: 75.7 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

(b) Throughput
Duration: 5 min  TFRC: Yes  Thin: 13  Greedy: 5
ITT: 75.7 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

(c) Latency
Duration: 5 min  TFRC: Yes  Thin: 13  Greedy: 5
ITT: 75.7 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B
### Figure A.3.69 Latency tests with greedy cross traffic bundle limitation

#### Goodput

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>sender</td>
<td>13</td>
<td>74/74/74</td>
<td>120/120/120</td>
<td>7/1/7/8/8/10</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>13/14/14</td>
<td>1446/1447/1448</td>
<td>17/5/13/16/20/42</td>
</tr>
</tbody>
</table>

#### Throughput

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>sender</td>
<td>13</td>
<td>74/74/74</td>
<td>120/120/120</td>
<td>7/1/7/8/8/10</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>13/14/14</td>
<td>1446/1447/1448</td>
<td>17/5/13/16/20/42</td>
</tr>
</tbody>
</table>

#### Latency

- **(a)**
- **(b)**
- **(c)**

---

### Figure A.3.70

#### Goodput

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>sender</td>
<td>13</td>
<td>74/74/74</td>
<td>120/120/120</td>
<td>7/1/7/8/8/10</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>13/14/14</td>
<td>1446/1447/1448</td>
<td>17/5/13/16/20/42</td>
</tr>
</tbody>
</table>

#### Throughput

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>sender</td>
<td>13</td>
<td>74/74/74</td>
<td>120/120/120</td>
<td>7/1/7/8/8/10</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>13/14/14</td>
<td>1446/1447/1448</td>
<td>17/5/13/16/20/42</td>
</tr>
</tbody>
</table>

#### Latency

- **(a)**
- **(b)**
- **(c)**

---

**Note:** The figures and tables illustrate latency tests with greedy cross traffic bundle limitation, showing Goodput (Kbit/second aggregated over 1 second), Throughput (Kbit/second aggregated over 1 second), and ACK Latency in milliseconds. The tables provide detailed data for sender and receiver stream metrics, including ITT, RTT, QLEN, Payload, and other performance indicators.
Appendix A. Experiments results

Figure A.3.71

Figure A.3.72
A.3. Latency tests with greedy cross traffic bundle limitation

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>13</td>
<td>98 / 99 / 99</td>
<td>120/120/120</td>
<td>7/1/7/7/7/10</td>
</tr>
<tr>
<td>RDB DPIF10 ER+TLP</td>
<td></td>
<td>13</td>
<td>99 / 100 / 100</td>
<td>240/240/240</td>
<td>10/10/9/9/1/2/7</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>13 / 13 / 14</td>
<td>1444/1446/1447</td>
<td>18/7/15/18/21/41</td>
</tr>
</tbody>
</table>

**Figure A.3.73**

**Goodput**
Duration: 5 min  
TFRC: Yes  
Thin: 13  
Greedy: 5  
ITT: 100:10 ms  
RTT: 150 ms  
QLEN: 63p  
Payload: 120 B

**Throughput**
Duration: 5 min  
TFRC: Yes  
Thin: 13  
Greedy: 5  
ITT: 100:10 ms  
RTT: 150 ms  
QLEN: 63p  
Payload: 120 B

**Latency**
Duration: 5 min  
TFRC: Yes  
Thin: 13  
Greedy: 5  
ITT: 100:10 ms  
RTT: 150 ms  
QLEN: 63p  
Payload: 120 B

---

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>13</td>
<td>98 / 99 / 99</td>
<td>120/120/120</td>
<td>7/1/7/7/7/10</td>
</tr>
<tr>
<td>RDB DPIF20 ER+TLP</td>
<td></td>
<td>13</td>
<td>100 / 101 / 103</td>
<td>240/240/243</td>
<td>7/1/7/7/7/10</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>12 / 13 / 14</td>
<td>1447/1447/1447</td>
<td>18/6/15/18/21/42</td>
</tr>
</tbody>
</table>

**Figure A.3.74**

**Goodput**
Duration: 5 min  
TFRC: No  
Thin: 13  
Greedy: 5  
ITT: 100:10 ms  
RTT: 150 ms  
QLEN: 63p  
Payload: 120 B

**Throughput**
Duration: 5 min  
TFRC: No  
Thin: 13  
Greedy: 5  
ITT: 100:10 ms  
RTT: 150 ms  
QLEN: 63p  
Payload: 120 B

**Latency**
Duration: 5 min  
TFRC: No  
Thin: 13  
Greedy: 5  
ITT: 100:10 ms  
RTT: 150 ms  
QLEN: 63p  
Payload: 120 B

---
Goodput
Duration: 5 min  TFRC: Yes  Thin: 13  Greedy: 5
ITT: 100/10 ms  RTT: 150 ms  QLEN: 63 p  Payload: 120 B
TCP ER+TLP  RDB DPIF20 ER+TLP  Greedy

Throughput
Duration: 5 min  TFRC: Yes  Thin: 13  Greedy: 5
ITT: 100/10 ms  RTT: 150 ms  QLEN: 63 p  Payload: 120 B
TCP ER+TLP  RDB DPIF20 ER+TLP  Greedy

Latency
Duration: 5 min  TFRC: Yes  Thin: 13  Greedy: 5
ITT: 100/10 ms  RTT: 150 ms  QLEN: 63 p  Payload: 120 B
TCP ER+TLP  RDB DPIF20 ER+TLP  Greedy

Table: Goodput
Name | Host | Streams | ITT (avg) | Payload (avg) | CWND
TCP ER+TLP  wsender  13  98 / 99 / 99  120 / 120 / 120  7 / 7 / 7 / 7 / 10
Greedy  zsender  5  12 / 13 / 15  1447 / 1447 / 1447  19 / 3 / 14 / 18 / 22 / 53

Figure A.3.75
(a) Goodput
(b) Throughput
(c) Latency

Figure A.3.76
(a) Goodput
(b) Throughput
(c) Latency
A.3. Latency tests with greedy cross traffic bundle limitation

**Figure A.3.77**

**Figure A.3.78**
Figure A.3.79

Throughput (Kbit/second aggregated over 1 second)

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>16</td>
<td>44 / 47 / 51</td>
<td>579/626/682</td>
<td>61/4/8/24</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>46 / 48 / 44</td>
<td>1446/1447/1447</td>
<td>61/4/5/7/29</td>
</tr>
</tbody>
</table>

Latency

<table>
<thead>
<tr>
<th>Sent p</th>
<th>Lost p</th>
<th>Retr p</th>
<th>RDB p</th>
<th>Payload</th>
<th>Loss %</th>
<th>DupACKs</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1 K</td>
<td>5478</td>
<td>5478</td>
<td>0</td>
<td>63.5 M</td>
<td>60.3 M</td>
<td>3.1 M</td>
</tr>
<tr>
<td>90.4 K</td>
<td>5376</td>
<td>3547</td>
<td>39.8 K</td>
<td>74.3 M</td>
<td>57.9 M</td>
<td>16.4 M</td>
</tr>
<tr>
<td>32.3 K</td>
<td>2059</td>
<td>2059</td>
<td>46.8 B</td>
<td>43.8 B</td>
<td>3.0 M</td>
<td>3.0 M</td>
</tr>
</tbody>
</table>

Figure A.3.80

Throughput (Kbit/second aggregated over 1 second)

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>16</td>
<td>44 / 47 / 51</td>
<td>579/626/682</td>
<td>61/4/8/24</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>46 / 48 / 44</td>
<td>1446/1447/1447</td>
<td>61/4/5/7/29</td>
</tr>
</tbody>
</table>

Latency

<table>
<thead>
<tr>
<th>Sent p</th>
<th>Lost p</th>
<th>Retr p</th>
<th>RDB p</th>
<th>Payload</th>
<th>Loss %</th>
<th>DupACKs</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1 K</td>
<td>5478</td>
<td>5478</td>
<td>0</td>
<td>63.5 M</td>
<td>60.3 M</td>
<td>3.1 M</td>
</tr>
<tr>
<td>90.4 K</td>
<td>5376</td>
<td>3547</td>
<td>39.8 K</td>
<td>74.3 M</td>
<td>57.9 M</td>
<td>16.4 M</td>
</tr>
<tr>
<td>32.3 K</td>
<td>2059</td>
<td>2059</td>
<td>46.8 B</td>
<td>43.8 B</td>
<td>3.0 M</td>
<td>3.0 M</td>
</tr>
</tbody>
</table>

Appendix A. Experiments results

228
A.3. Latency tests with greedy cross traffic bundle limitation
Goodput
Duration: 5 min  TFRC: Yes  Thin: 16  Greedy: 5
ITT: 30.3 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B
TCP ER+TLP  RDB DPIF20 ER+TLP  Greedy

Throughput
Duration: 5 min  TFRC: Yes  Thin: 16  Greedy: 5
ITT: 30.3 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B
TCP ER+TLP  RDB DPIF20 ER+TLP  Greedy

Latency
Duration: 5 min  TFRC: Yes  Thin: 16  Greedy: 5
ITT: 30.3 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B
TCP ER+TLP  RDB DPIF20 ER+TLP  Greedy

Appendix A. Experiments results
A.3. Latency tests with greedy cross traffic bundle limitation
Appendix A. Experiments results

Figure A.3.87

(a) Goodput
(b) Throughput
(c) Latency

Figure A.3.88

(a) Goodput
(b) Throughput
(c) Latency
Figure A.3.89

Figure A.3.90
Figure A.3.91

Figure A.3.92
Goodput Duration: 5 min  TFRC: Yes  Thin: 16  Greedy: 5  
ITT: 75.7 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP variation used for competing streams

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>16</td>
<td>75</td>
<td>75</td>
<td>120/120/121</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>13/14/15 1447/1447/1447</td>
<td>16/8/13/15/18/43</td>
<td></td>
</tr>
</tbody>
</table>

Throughput Duration: 5 min  TFRC: Yes  Thin: 16  Greedy: 5  
ITT: 75.7 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

TCP variation used for competing streams

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>16</td>
<td>75</td>
<td>77</td>
<td>120/120/121</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>5</td>
<td>13/14/15 1447/1447/1447</td>
<td>16/8/13/15/18/43</td>
<td></td>
</tr>
</tbody>
</table>

Latency Duration: 5 min  TFRC: Yes  Thin: 16  Greedy: 5  
ITT: 75.7 ms  RTT: 150 ms  QLEN: 63p  Payload: 120 B

ACK Latency in mSeconds

Max: 246 222 224 227 229 232 237 234 248 241 262 269 326 321 323 325 328 482 329 365 352 435 446 456 474 510 673 867 999
Figure A.3.95

(a) Goodput

(b) Throughput

(c) Latency

Figure A.3.96

(a) Goodput

(b) Throughput

(c) Latency
A.3. Latency tests with greedy cross traffic bundle limitation
A.4 Fairness experiments

Table A.4 gives the complete test setup for the fairness experiments.

<table>
<thead>
<tr>
<th>Figure</th>
<th>Type</th>
<th>Streams</th>
<th>Cong</th>
<th>TFRC</th>
<th>RTT</th>
<th>ITT</th>
<th>Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.4.1</td>
<td>TCP ER+TLP</td>
<td>3</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>5:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.2</td>
<td>RDB DPIF10 ER+TLP</td>
<td>3</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>5:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.3</td>
<td>RDB DPIF10 ER+TLP</td>
<td>3</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>5:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.4</td>
<td>TCP ER+TLP</td>
<td>3</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>15:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.5</td>
<td>RDB DPIF10 ER+TLP</td>
<td>3</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>15:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.6</td>
<td>RDB DPIF10 ER+TLP</td>
<td>3</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>15:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.7</td>
<td>TCP ER+TLP</td>
<td>3</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>30:3</td>
<td>400</td>
</tr>
<tr>
<td>A.4.8</td>
<td>RDB DPIF10 ER+TLP</td>
<td>3</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>30:3</td>
<td>400</td>
</tr>
<tr>
<td>A.4.9</td>
<td>RDB DPIF10 ER+TLP</td>
<td>3</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>30:3</td>
<td>400</td>
</tr>
<tr>
<td>A.4.10</td>
<td>TCP ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>5:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.11</td>
<td>RDB DPIF10 ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>5:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.12</td>
<td>RDB DPIF10 ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>5:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.13</td>
<td>TCP ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>15:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.14</td>
<td>RDB DPIF10 ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>15:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.15</td>
<td>RDB DPIF10 ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>15:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.16</td>
<td>TCP ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>30:3</td>
<td>400</td>
</tr>
<tr>
<td>A.4.17</td>
<td>RDB DPIF10 ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>30:3</td>
<td>400</td>
</tr>
<tr>
<td>A.4.18</td>
<td>RDB DPIF10 ER+TLP</td>
<td>5</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>30:3</td>
<td>400</td>
</tr>
<tr>
<td>A.4.19</td>
<td>TCP ER+TLP</td>
<td>7</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>5:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.20</td>
<td>RDB DPIF10 ER+TLP</td>
<td>7</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>5:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.21</td>
<td>RDB DPIF10 ER+TLP</td>
<td>7</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>5:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.22</td>
<td>TCP ER+TLP</td>
<td>7</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>15:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.23</td>
<td>RDB DPIF10 ER+TLP</td>
<td>7</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>15:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.24</td>
<td>RDB DPIF10 ER+TLP</td>
<td>7</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>15:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.25</td>
<td>TCP ER+TLP</td>
<td>7</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>30:3</td>
<td>400</td>
</tr>
<tr>
<td>A.4.26</td>
<td>RDB DPIF10 ER+TLP</td>
<td>7</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>30:3</td>
<td>400</td>
</tr>
<tr>
<td>A.4.27</td>
<td>RDB DPIF10 ER+TLP</td>
<td>7</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>30:3</td>
<td>400</td>
</tr>
<tr>
<td>A.4.28</td>
<td>TCP ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>5:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.29</td>
<td>RDB DPIF10 ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>5:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.30</td>
<td>RDB DPIF10 ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>5:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.31</td>
<td>TCP ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>15:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.32</td>
<td>RDB DPIF10 ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>15:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.33</td>
<td>RDB DPIF10 ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>15:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.34</td>
<td>TCP ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>30:3</td>
<td>400</td>
</tr>
<tr>
<td>A.4.35</td>
<td>RDB DPIF10 ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>30:3</td>
<td>400</td>
</tr>
<tr>
<td>A.4.36</td>
<td>RDB DPIF10 ER+TLP</td>
<td>10</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>30:3</td>
<td>400</td>
</tr>
<tr>
<td>A.4.37</td>
<td>TCP ER+TLP</td>
<td>13</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>5:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.38</td>
<td>RDB DPIF10 ER+TLP</td>
<td>13</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>5:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.39</td>
<td>RDB DPIF10 ER+TLP</td>
<td>13</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>5:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.40</td>
<td>TCP ER+TLP</td>
<td>13</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>15:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.41</td>
<td>RDB DPIF10 ER+TLP</td>
<td>13</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>15:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.42</td>
<td>RDB DPIF10 ER+TLP</td>
<td>13</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>15:1</td>
<td>400</td>
</tr>
<tr>
<td>A.4.43</td>
<td>TCP ER+TLP</td>
<td>13</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>30:3</td>
<td>400</td>
</tr>
<tr>
<td>A.4.44</td>
<td>RDB DPIF10 ER+TLP</td>
<td>13</td>
<td>Rdb</td>
<td>No</td>
<td>150</td>
<td>30:3</td>
<td>400</td>
</tr>
<tr>
<td>A.4.45</td>
<td>RDB DPIF10 ER+TLP</td>
<td>13</td>
<td>Rdb</td>
<td>Yes</td>
<td>150</td>
<td>30:3</td>
<td>400</td>
</tr>
</tbody>
</table>
Table A.4: Test setup for experiment 4
### A.4. Fairness experiments

#### Goodput
**Duration**: 5 min  **TFRC**: No  **Isoc**: 3  **Greedy**: 3  
**ITT**: 5:1 ms  **RTT**: 150 ms  **QLEN**: 63p  **Payload**: 400 B

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>3</td>
<td>12/18/24</td>
<td>1447/147/144</td>
<td>3.4</td>
</tr>
<tr>
<td>TCP ER+TLP</td>
<td>zsender</td>
<td>3</td>
<td>14/14/14</td>
<td>1447/147/144</td>
<td>3.3</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>3</td>
<td>14/14/14</td>
<td>1447/147/144</td>
<td>3.1</td>
</tr>
</tbody>
</table>

#### Throughput
**Duration**: 5 min  **TFRC**: No  **Isoc**: 3  **Greedy**: 3  
**ITT**: 5:1 ms  **RTT**: 150 ms  **QLEN**: 63p  **Payload**: 400 B

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>3</td>
<td>12/18/24</td>
<td>1447/147/144</td>
<td>3.4</td>
</tr>
<tr>
<td>TCP ER+TLP</td>
<td>zsender</td>
<td>3</td>
<td>14/14/14</td>
<td>1447/147/144</td>
<td>3.3</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>3</td>
<td>14/14/14</td>
<td>1447/147/144</td>
<td>3.1</td>
</tr>
</tbody>
</table>

#### Latency
**Duration**: 5 min  **TFRC**: No  **Isoc**: 3  **Greedy**: 3  
**ITT**: 5:1 ms  **RTT**: 150 ms  **QLEN**: 63p  **Payload**: 400 B

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>3</td>
<td>12/18/24</td>
<td>1447/147/144</td>
<td>3.4</td>
</tr>
<tr>
<td>TCP ER+TLP</td>
<td>zsender</td>
<td>3</td>
<td>14/14/14</td>
<td>1447/147/144</td>
<td>3.3</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>3</td>
<td>14/14/14</td>
<td>1447/147/144</td>
<td>3.1</td>
</tr>
</tbody>
</table>

**Figure A.4.1**

**Figure A.4.2**
Appendix A. Experiments results
Figure A.4.5

(a) Goodput
Duration: 5 min  TFRC: No  Isoc: 3  Greedy: 3
ITT: 15:1 ms  RTT: 150 ms  QLEN: 63p  Payload: 400 B

(b) Throughput
Duration: 5 min  TFRC: No  Isoc: 3  Greedy: 3
ITT: 15:1 ms  RTT: 150 ms  QLEN: 63p  Payload: 400 B

(c) Latency
Duration: 5 min  TFRC: No  Isoc: 3  Greedy: 3
ITT: 15:1 ms  RTT: 150 ms  QLEN: 63p  Payload: 400 B

Figure A.4.6
Figure A.4.10

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>3</td>
<td>30 / 30 / 30</td>
<td>45/48/48 / 15</td>
<td>12/11/14/14</td>
</tr>
<tr>
<td>TCP ER+TLP</td>
<td>rdbsender</td>
<td>3</td>
<td>30 / 30 / 30</td>
<td>159/179/190</td>
<td>16/14/16/22</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>3</td>
<td>14/10/10</td>
<td>1447/1447/1448</td>
<td>26/10/22/30/57</td>
</tr>
</tbody>
</table>

(ITT: ITT, RTT: RTT, QLEN: QLEN, Payload: Payload)

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>5</td>
<td>42 / 43 / 44</td>
<td>1440/1440/1445</td>
<td>7/15/19/28</td>
</tr>
<tr>
<td>TCP ER+TLP</td>
<td>rdbsender</td>
<td>5</td>
<td>39 / 41 / 42</td>
<td>1410/1410/1412</td>
<td>9/15/7/9/28</td>
</tr>
</tbody>
</table>

(ITT: ITT, RTT: RTT, QLEN: QLEN, Payload: Payload)

Figure A.4.9

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>5</td>
<td>42 / 43 / 44</td>
<td>1440/1440/1445</td>
<td>7/15/19/28</td>
</tr>
<tr>
<td>TCP ER+TLP</td>
<td>rdbsender</td>
<td>5</td>
<td>39 / 41 / 42</td>
<td>1410/1410/1412</td>
<td>9/15/7/9/28</td>
</tr>
</tbody>
</table>

(ITT: ITT, RTT: RTT, QLEN: QLEN, Payload: Payload)
Appendix A. Experiments results

Figure A.4.11

(a) Goodput
(b) Throughput
(c) Latency

Figure A.4.12

(a) Goodput
(b) Throughput
(c) Latency

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender 5</td>
<td>43 / 44 / 45 / 1446 / 1446</td>
<td>7.1 / 57 / 8 / 18</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>RDB DPIF10 ER+TLP</td>
<td>zsender 5</td>
<td>42 / 44 / 45 / 1446 / 1446 / 1446</td>
<td>7.1 / 57 / 8 / 18</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>Greedy</td>
<td>wsender 5</td>
<td>24 / 25 / 26 / 1446 / 1446 / 1447</td>
<td>12 / 59 / 11 / 13 / 32</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>TCP ER+TLP</td>
<td>wsender 5</td>
<td>43 / 44 / 45 / 1446 / 1446</td>
<td>7.1 / 57 / 8 / 18</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>RDB DPIF10 ER+TLP</td>
<td>zsender 5</td>
<td>42 / 44 / 45 / 1446 / 1446 / 1446</td>
<td>7.1 / 57 / 8 / 18</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>Greedy</td>
<td>wsender 5</td>
<td>24 / 25 / 26 / 1446 / 1446 / 1447</td>
<td>12 / 59 / 11 / 13 / 32</td>
<td>1000</td>
<td></td>
</tr>
</tbody>
</table>

Figure A.4.12

(a) Goodput
(b) Throughput
(c) Latency

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender 5</td>
<td>43 / 44 / 45 / 1446 / 1446</td>
<td>7.1 / 57 / 8 / 18</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>RDB DPIF10 ER+TLP</td>
<td>zsender 5</td>
<td>42 / 44 / 45 / 1446 / 1446 / 1446</td>
<td>7.1 / 57 / 8 / 18</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>Greedy</td>
<td>wsender 5</td>
<td>24 / 25 / 26 / 1446 / 1446 / 1447</td>
<td>12 / 59 / 11 / 13 / 32</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>TCP ER+TLP</td>
<td>wsender 5</td>
<td>43 / 44 / 45 / 1446 / 1446</td>
<td>7.1 / 57 / 8 / 18</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>RDB DPIF10 ER+TLP</td>
<td>zsender 5</td>
<td>42 / 44 / 45 / 1446 / 1446 / 1446</td>
<td>7.1 / 57 / 8 / 18</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>Greedy</td>
<td>wsender 5</td>
<td>24 / 25 / 26 / 1446 / 1446 / 1447</td>
<td>12 / 59 / 11 / 13 / 32</td>
<td>1000</td>
<td></td>
</tr>
</tbody>
</table>
A.4. Fairness experiments

Figure A.4.13

(a) Goodput
Duration: 5 min  TFRC: No  Isoc: 5  Greedy: 5
ITT: 15:1 ms  RTT: 150 ms  QLEN: 63p  Payload: 400 B

(b) Throughput
Duration: 5 min  TFRC: No  Isoc: 5  Greedy: 5
ITT: 15:1 ms  RTT: 150 ms  QLEN: 63p  Payload: 400 B

(c) Latency
Duration: 5 min  TFRC: No  Isoc: 5  Greedy: 5
ITT: 15:1 ms  RTT: 150 ms  QLEN: 63p  Payload: 400 B

Figure A.4.14

(a) Goodput
Duration: 5 min  TFRC: No  Isoc: 5  Greedy: 5
ITT: 15:1 ms  RTT: 150 ms  QLEN: 63p  Payload: 400 B

(b) Throughput
Duration: 5 min  TFRC: No  Isoc: 5  Greedy: 5
ITT: 15:1 ms  RTT: 150 ms  QLEN: 63p  Payload: 400 B

(c) Latency
Duration: 5 min  TFRC: No  Isoc: 5  Greedy: 5
ITT: 15:1 ms  RTT: 150 ms  QLEN: 63p  Payload: 400 B
**Appendix A. Experiments results**

**Figure A.4.15**

**Goodput**
- Duration: 5 min
- TFRC: Yes
- Isoc: 5
- Greedy: 5
- ITT: 30.3 ms
- RTT: 150 ms
- QLEN: 63p
- Payload: 400 B

**Throughput**
- Duration: 5 min
- TFRC: Yes
- Isoc: 5
- Greedy: 5
- ITT: 30.3 ms
- RTT: 150 ms
- QLEN: 63p
- Payload: 400 B

**Latency**
- Duration: 5 min
- TFRC: Yes
- Isoc: 5
- Greedy: 5
- ITT: 30.3 ms
- RTT: 150 ms
- QLEN: 63p
- Payload: 400 B

**Figure A.4.16**

**Goodput**
- Duration: 5 min
- TFRC: Yes
- Isoc: 5
- Greedy: 5
- ITT: 15.1 ms
- RTT: 150 ms
- QLEN: 63p
- Payload: 400 B

**Throughput**
- Duration: 5 min
- TFRC: Yes
- Isoc: 5
- Greedy: 5
- ITT: 15.1 ms
- RTT: 150 ms
- QLEN: 63p
- Payload: 400 B

**Latency**
- Duration: 5 min
- TFRC: Yes
- Isoc: 5
- Greedy: 5
- ITT: 15.1 ms
- RTT: 150 ms
- QLEN: 63p
- Payload: 400 B
### A.4. Fairness experiments

#### Figure A.4.17

**Goodput**
- Duration: 5 min
- TFRC: No
- Isoc: 5
- Greedy: 5
- ITT: 30.3 ms
- RTT: 150 ms
- QLEN: 63p
- Payload: 400 B

**Throughput**
- Duration: 5 min
- TFRC: No
- Isoc: 5
- Greedy: 5
- ITT: 30.3 ms
- RTT: 150 ms
- QLEN: 63p
- Payload: 400 B

**Latency**
- Duration: 5 min
- TFRC: No
- Isoc: 5
- Greedy: 5
- ITT: 30.3 ms
- RTT: 150 ms
- QLEN: 63p
- Payload: 400 B

---

#### Figure A.4.18

**Goodput**
- Duration: 5 min
- TFRC: Yes
- Isoc: 5
- Greedy: 5
- ITT: 30.3 ms
- RTT: 150 ms
- QLEN: 63p
- Payload: 400 B

**Throughput**
- Duration: 5 min
- TFRC: Yes
- Isoc: 5
- Greedy: 5
- ITT: 30.3 ms
- RTT: 150 ms
- QLEN: 63p
- Payload: 400 B

**Latency**
- Duration: 5 min
- TFRC: Yes
- Isoc: 5
- Greedy: 5
- ITT: 30.3 ms
- RTT: 150 ms
- QLEN: 63p
- Payload: 400 B
A.4. Fairness experiments
Figure A.4.23

Figure A.4.24
A.4. Fairness experiments
Appendix A. Experiments results
### A.4. Fairness experiments

#### Figure A.4.29

**Goodput**
- Duration: 5 min
- TFRC: No
- Isoc: 10
- Greedy: 10

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>10</td>
<td>78 / 81 / 84</td>
<td>1443/1443/1444</td>
<td>41/3/4/5/15</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>10</td>
<td>45 / 51 / 57</td>
<td>1441/1445/1447</td>
<td>6/2/5/6/7/9</td>
</tr>
</tbody>
</table>

**Throughput**
- Duration: 5 min
- TFRC: Yes
- Isoc: 10
- Greedy: 10

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>10</td>
<td>75 / 80 / 85</td>
<td>1442/1443/1444</td>
<td>41/3/4/5/15</td>
</tr>
<tr>
<td>RDB DPIF10 ER+TLP</td>
<td>zsender</td>
<td>10</td>
<td>81 / 89 / 96</td>
<td>1443/1444/1445</td>
<td>41/3/4/5/24</td>
</tr>
</tbody>
</table>

**Latency**
- Duration: 5 min
- TFRC: No
- Isoc: 10
- Greedy: 10

Mean 50th 60th 70th 80th 90th 95th 99th 99% 99.9% 99.99% Max

421 302 302 440 593 779 799 872 886 901 1085 1172 1475 2000 2312
429 302 302 465 598 787 847 884 896 1020 1111 1182 1332 1965 2296
391 302 302 327 588 607 624 790 849 874 888 898 9112 1787 2364

#### Figure A.4.30

**Goodput**
- Duration: 5 min
- TFRC: No
- Isoc: 10
- Greedy: 10

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>10</td>
<td>78 / 81 / 84</td>
<td>1443/1443/1444</td>
<td>41/3/4/5/15</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>10</td>
<td>45 / 51 / 57</td>
<td>1441/1445/1447</td>
<td>6/2/5/6/7/9</td>
</tr>
</tbody>
</table>

**Throughput**
- Duration: 5 min
- TFRC: Yes
- Isoc: 10
- Greedy: 10

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>wsender</td>
<td>10</td>
<td>75 / 80 / 85</td>
<td>1442/1443/1444</td>
<td>41/3/4/5/15</td>
</tr>
<tr>
<td>RDB DPIF10 ER+TLP</td>
<td>zsender</td>
<td>10</td>
<td>81 / 89 / 96</td>
<td>1443/1444/1445</td>
<td>41/3/4/5/24</td>
</tr>
</tbody>
</table>

**Latency**
- Duration: 5 min
- TFRC: Yes
- Isoc: 10
- Greedy: 10

Mean 50th 60th 70th 80th 90th 95th 99th 99% 99.9% 99.99% Max

419 302 302 404 591 779 801 876 891 931 1006 1050 1179 1494 2487 3273
449 302 302 431 598 792 887 893 908 1048 1160 1199 4627 22202 22230
386 302 302 317 586 605 615 748 782 825 864 884 896 1177 1465 1643
Figure A.4.31

(a) Goodput
Duration: 5 min  TFRC: No  Isoc: 10  Greedy: 10
ITT: 15:1 ms   RTT: 150 ms  QLEN: 63p  Payload: 400 B
TCP variation used for competing streams

(b) Throughput
Duration: 5 min  TFRC: No  Isoc: 10  Greedy: 10
ITT: 15:1 ms   RTT: 150 ms  QLEN: 63p  Payload: 400 B
TCP variation used for competing streams

(c) Latency
Duration: 5 min  TFRC: No  Isoc: 10  Greedy: 10
ITT: 15:1 ms   RTT: 150 ms  QLEN: 63p  Payload: 400 B
TCP variation used for competing streams

Figure A.4.32

(a) Goodput
Duration: 5 min  TFRC: No  Isoc: 10  Greedy: 10
ITT: 15:1 ms   RTT: 150 ms  QLEN: 63p  Payload: 400 B
TCP variation used for competing streams

(b) Throughput
Duration: 5 min  TFRC: No  Isoc: 10  Greedy: 10
ITT: 15:1 ms   RTT: 150 ms  QLEN: 63p  Payload: 400 B
TCP variation used for competing streams

(c) Latency
Duration: 5 min  TFRC: No  Isoc: 10  Greedy: 10
ITT: 15:1 ms   RTT: 150 ms  QLEN: 63p  Payload: 400 B
TCP variation used for competing streams
Figure A.4.35

Throughput
Duration: 5 min  TFRC: Yes  IsoC: 10  Greedy: 10
ITT: 303 ms  RTT: 150 ms  QLEN: 63p  Payload: 400 B
TCP ER+TLP  RDB DPIF10 ER+TLP  Greedy

(c)

Goodput
Duration: 5 min  TFRC: No  IsoC: 10  Greedy: 10
ITT: 303 ms  RTT: 150 ms  QLEN: 63p  Payload: 400 B
TCP ER+TLP  RDB DPIF10 ER+TLP  Greedy

(a)

Figure A.4.36

Throughput
Duration: 5 min  TFRC: Yes  IsoC: 10  Greedy: 10
ITT: 303 ms  RTT: 150 ms  QLEN: 63p  Payload: 400 B
TCP ER+TLP  RDB DPIF10 ER+TLP  Greedy

(b)

Goodput
Duration: 5 min  TFRC: No  IsoC: 10  Greedy: 10
ITT: 303 ms  RTT: 150 ms  QLEN: 63p  Payload: 400 B
TCP ER+TLP  RDB DPIF10 ER+TLP  Greedy

(a)
Figure A.4.37

Figure A.4.38

Figure A.4.39
Figure A.4.39

Figure A.4.40
### A.4. Fairness experiments

#### Figure A.4.41

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>zsender</td>
<td>13</td>
<td>90 / 97 / 119</td>
<td>1438 / 1438 / 1438</td>
<td>4 / 1 / 3 / 4 / 17</td>
</tr>
<tr>
<td>Greedy</td>
<td>zsender</td>
<td>13</td>
<td>64 / 70 / 76</td>
<td>1438 / 1448 / 1447</td>
<td>5 / 1 / 4 / 5 / 6 / 20</td>
</tr>
</tbody>
</table>

#### Figure A.4.42

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Streams</th>
<th>ITT (avg)</th>
<th>Payload (avg)</th>
<th>CWND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP ER+TLP</td>
<td>zsender</td>
<td>13</td>
<td>86 / 94 / 101</td>
<td>1388 / 1350 / 1368</td>
<td>5 / 1 / 3 / 4 / 5 / 6</td>
</tr>
<tr>
<td>RDB DPIF10 ER+TLP</td>
<td>zsender</td>
<td>13</td>
<td>60 / 70 / 79</td>
<td>1445 / 1446 / 1447</td>
<td>5 / 1 / 4 / 5 / 6 / 15</td>
</tr>
</tbody>
</table>
Figure A.4.43

Figure A.4.44

Appendix A. Experiments results
Figure A.4.45
Appendix B

RDBv2 implementation source code

In this chapter, some key parts of the RDBv2 implementation are listed. The complete source code of RDBv2 can be found at bitbucket.org/bendikro/net-next-rdb.

B.1 Congestion control changes in Linux

```c
/*
 * TCP Reno congestion control
 * This is special case used for fallback as well.
 */
/*
 * This is Jacobson's slow start and congestion avoidance.
 */
/* SIGCOMM '88, p. 328.
 */
void tcp_reno_cong_avoid(struct sock *sk, u32 ack, u32 acked)
{
    struct tcp_sock *tp = tcp_sk(sk);
    if (!tcp_is_c wnd_limited(sk))
        return;
    /* In "safe" area, increase. */
    if (tp->snd_cwnd <= tp->snd_ssthresh)
        tcp_slow_start(tp, acked);
    /* In dangerous area, increase slowly. */
    else
tcp_cong_avoid_ai(tp, tp->snd_cwnd);
}
```

This function is located in tcp_cong.c and tests if the stream is application limited.

Code Listing B.1: tcp_is_cwnd_limited from TCP New Reno that tests if the send rate is limited by the CWND
The implementation of `tcp_is_cwnd_limited` in TCP New Reno (`tcp_cong.c`) from kernel version 3.15

Code Listing B.2: `tcp_is_cwnd_limited` in Linux kernel version 3.15 that tests if the send rate is limited by the CWND
This function is located in `tcp_cong.c` and tests if the stream is application limited.

Code Listing B.3: `tcp_is_cwnd_limited` in Linux kernel version 3.16 that tests if the send rate is limited by the CWND

### B.2 RDBv2 bundling implementation

<table>
<thead>
<tr>
<th>Source Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>static bool tcp_write_xmit(struct sock *sk, unsigned int mss_now,</td>
</tr>
<tr>
<td>int nonagle, int push_one, gfp_t gfp)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>while ((skb = tcp_send_head(sk))) {</td>
</tr>
<tr>
<td>/* Does at least the first segment of SKB fit into the send</td>
</tr>
<tr>
<td>window? */</td>
</tr>
<tr>
<td>if (unlikely(!tcp_snd_wnd_test(tp, skb, mss_now)))</td>
</tr>
<tr>
<td>break;</td>
</tr>
<tr>
<td>/* .... */</td>
</tr>
<tr>
<td>TCP_SKB_CB(skb)-&gt;when = tcp_time_stamp;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>// Code added to support RDB</td>
</tr>
<tr>
<td>if (unlikely(inet_csk(skb)-&gt;icsk_ca_ops-&gt;pkt_send)) {</td>
</tr>
<tr>
<td>loop_counter++;</td>
</tr>
<tr>
<td>/* If 1, SKB has been sent by the function */</td>
</tr>
<tr>
<td>if (likely(inet_csk(skb)-&gt;icsk_ca_ops-&gt;</td>
</tr>
<tr>
<td>pkt_send(skb, mss_now, sk, tp,</td>
</tr>
<tr>
<td>gfp, loop_counter, false))) {</td>
</tr>
<tr>
<td>goto repair;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>// End: code added to support RDB</td>
</tr>
<tr>
<td>if (unlikely(tcp_transmit_skb(skb, skb, 1, gfp)))</td>
</tr>
<tr>
<td>break;</td>
</tr>
<tr>
<td>repair:</td>
</tr>
<tr>
<td>/* Advance the send_head. This one is sent out.</td>
</tr>
<tr>
<td>* This call will increment packets_out. */</td>
</tr>
<tr>
<td>tcp_event_new_data_sent(skb, skb);</td>
</tr>
<tr>
<td>/* .... */</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Code Listing B.4: Excerpt from the function `tcp_write_xmit` in `tcp_output.c`
The function `do_rdb` implemented in `rdb_cc.c`. The code has been stripped down to only the most important parts.
B.2. RDBv2 bundling implementation

Source Code

```c
inline bool run_rdb(struct RDBcong *cong, struct tcp_sock *tp) {
    return (tp->thin_rdb) &&
           (cong->dynamic_pif ? tcp_stream_is_thin_dpif(tp, cong->
                       minrtt_us) : tcp_stream_is_thin_spif(tp));
}
```

The function `run_rdb` implemented in `rdb_cc.c`.

Code Listing B.6

Source Code

```c
/**
 * tcp_stream_is_thin_dpif() - Tests if the stream is thin based in
dynamic PIF limit
 * min_rtt_us: The minimum RTT registered for the connection
 * Return: true if current PIF count is lower than the dynamic PIF
 * limit, else false
 */
inline bool tcp_stream_is_thin_dpif(struct tcp_sock *tp, u32
    min_rtt_us) {
  return tp->packets_out < ((min_rtt_us >> 3) / 1000 /
                           dynamic_pif);
}
```

The function `tcp_stream_is_thin_dpif` implemented in `rdb_cc.c`.

Code Listing B.7: The new function in RDBv2 for classifying thin streams based
on a DPIFL.

Source Code

```c
static unsigned int tcp_stream_is_thin_spif(struct tcp_sock *tp) {
    return tp->packets_out < sysctl_tcp_thin_packet_limit;
}
```

The function `tcp_stream_is_thin_spif` implemented in `rdb_cc.c`.

Code Listing B.8: A modification to `tcp_stream_is_thin` used in the experi-
ments with different SPIFL values.
Source Code

```c
1 /*
2  * redundant_data_bundling() - Tries to perform RDB
3  * xmit_skb: The SKB that should be sent
4  * retrans: If this is a retransmit
5  * Return: A new SKB containing redundant data, or NULL if no
6  * bundling could be performed
7 */
8 struct sk_buff* redundant_data_bundling(struct sk_buff *xmit_skb,
9    unsigned int mss_now, struct sock *sk, gfp_t gfp_mask, bool
10    retrans) {
11    u32 byte_count_in_rdb_skb;
12    u32 skb_data_offset;
13    u32 skb_in_bundle_count;
14    struct sk_buff* first_to_bundle, *rdb_skb;
15    ....
16    // First in write queue, means no previous SKB to merge with
17    if (!retrans && skb_queue_is_first(&sk->sk_write_queue,
18        xmit_skb)) {
19        return NULL;
20    }
21
22    // Previous has been SACKED
23    if (TCP_BYTECB(xmit_skb->prev)->sacked & TCPCB_SACKED_ACKED) {
24        return NULL;
25    }
26
27    // Find number of (previous) SKBs to get data from
28    first_to_bundle = rdb_can_bundle_check(sk, xmit_skb, mss_now,
29        &byte_count_in_rdb_skb, &skb_data_offset,
30        &skb_in_bundle_count);
31    if (!first_to_bundle) {
32        return NULL;
33    }
34
35    // Create an SKB that contains the data from 'first_to_bundle' SKBs.
36    rdb_skb = create_rdb_skb(first_to_bundle, xmit_skb, sk,
37        gfp_mask, byte_count_in_rdb_skb, 
38        skb_data_offset);
39    if (!rdb_skb) {
40        return NULL;
41    }
42    ....
43    return rdb_skb;
44}
```

Code from tcp_write_xmit in tcp_output.c. Removed code lines that are not directly relevant is indicated by “....”
B.2. RDBv2 bundling implementation

Source Code

```c
*/
* rdb_can_bundle_check() - check if redundant data can be bundled
* skb: The SKB with the newest data to be bundled
* byte_count_in_rdb_skb: Will contain the resulting number
* of bytes to bundle at exit.
* skb_data_offset: The data offset in the first SKB that will
* be in the bundle
* skbs_to_bundle_count: The total number of SKBs to be in the
* bundle
* Traverses the entire write queue and checks if there is
* data to be bundled.
* Return: The first SKB to be in the bundle, or NULL if
* no bundling.
*/

static struct sk_buff * rdb_can_bundle_check(struct sock *sk, struct sk_buff *xmit_skb, unsigned int mss_now, u32 *byte_count_in_rdb_skb, u32 *skb_data_offset, u32 *skbs_to_bundle_count) {
    struct sk_buff *first_to_bundle = NULL;
    struct sk_buff *tmp, *skb = xmit_skb->prev;
    u32 skbs_in_bundle_count = 1;
    u32 offset_in_skb = 0;
    u32 byte_count = xmit_skb->len;
    u32 tp_snd_una = tcp_sk(sk)->snd_una;

    if (!mss_now)
        mss_now = tcp_sk(sk)->mss_cache;
    if (skb_queue_is_first(&sk->sk_write_queue, xmit_skb))
        return NULL;

    // We start at the first skb, and go backwards in the list.
    tcp_for_write_queue_reverse_from_safe(skb, tmp, sk) {
        // This SKB does not contain unacked data, so break off here.
        if (TCP_SKB_CB(skb)->end_seq < tp_snd_una)
            break;

        // Not enough room to bundle data from this SKB
        if ((byte_count + skb->len) > mss_now) {
            break;
        }

        // Exceeds max_bundle_bytes
        if (sysctl_tcp_rdb_max_bundle_bytes && ((byte_count + skb->len) > sysctl_tcp_rdb_max_bundle_bytes)) {
            break;
        }

        // Exceeds max_bundle_skb
        if (sysctl_tcp_rdb_max_bundle_skbs && (skbs_in_bundle_count > sysctl_tcp_rdb_max_bundle_skbs)) {
            break;
        }

        byte_count += skb->len;
        // Calculate the offset in this SKB. If tp_snd_una is
        // bigger than seq, the first byte in the rdb-bundled SKB will be tp_snd_una
```
The function rdb_can_bundle_check in rdb_skb.c.

Code Listing B.10

B.3 TFRC simulations

TFRC-SP simulation: Throughput

```python
def calc_throughput(hist, tp_packet_size_func, hdr_accounting=False):
    """
    X is the transmit rate in bytes/second.
    s is the packet size in bytes.
    R is the round trip time in seconds.
    p is the loss event rate, between 0 and 1.0, of the
    number of loss events as a fraction of the number
    of packets transmitted.
    t_RTO is the TCP retransmission timeout value in seconds.
    b is the number of packets acknowledged by a single TCP
    acknowledgement.

    X = -----------------------------------
    R*sqrt(2*b*p/3) + (t_RTO * (3*sqrt(3*b*p/8) * p * (1+32*p^2)))

    """
    x = hist.rtt * sqrt(2 * hist.b * hist.p / 3.0) +
        (hist.t_rto * (3 * sqrt(3 * hist.b * hist.p / 8.0) * hist.p * (1 + 32 * pow(hist.p, 2))))
    x = tp_packet_size_func(hist) / x
    packets = x / hist.avg_size
    new_x = (x * hist.avg_size) / (hist.avg_size + 40)
    return new_x if hdr_accounting else x
```

Code Listing B.11: Function that calculates the send rate according to equation 2.2
B.4. TFRC Congestion Control implementation

TFRC-SP simulation: Loss Event Rate

```python
def loss_event_rate_sp(hist):
    I_tot0 = 0
    I_tot1 = 0
    W_tot = 0
    k_tot0 = 0
    k_tot1 = 0
    for i in range(len(hist.intervals)):
        I_i = hist.intervals[i].length
        if i > 0:
            interval_len = hist.intervals[i].t_start - hist.intervals[i-1].t_start
            rtt_tdelta = datetime.timedelta(microseconds=(self.rtt * 1000000))
        # This interval is short
        if interval_len < rtt_tdelta:
            I_i = hist.intervals[i].length / hist.intervals[i].lost
            I_tot0 += (I_i * tfrc_weights[i])
            W_tot += tfrc_weights[i]
            if i > 0:
                I_tot1 += (I_i * tfrc_weights[i-1])
    I_tot = max(I_tot0, I_tot1)
    try:
        I_mean = I_tot/float(W_tot)
        p = 1 / I_mean
        except ZeroDivisionError, e:
            raise
    return p
```

Code Listing B.12: Function that calculates the TFRC-SP loss event rate based on pseudo code 4.3.

B.4 TFRC Congestion Control implementation

Source Code

```c
struct tcp_congestion_ops {
    struct list_head list;
    unsigned long flags;

    /* initialize private data (optional) */
    void (*init)(struct sock *sk);
    /* cleanup private data (optional) */
    void (*release)(struct sock *sk);

    /* return slow start threshold (required) */
    u32 (*ssthresh)(struct sock *sk);
    /* lower bound for congestion window (optional) */
    u32 (*min_cwnd)(const struct sock *sk);
    /* do new cwnd calculation (required) */
    void (*cong_avoid)(struct sock *sk, u32 ack, u32 in_flight);
    /* call before changing ca_state (optional) */
    void (*set_state)(struct sock *sk, u8 new_state);
    /* new value of cwnd after loss (optional) */
    void (*cwnd_event)(struct sock *sk, enum tcp_ca_event ev);

    /* new value of cwnd after loss (optional) */
```
Code Listing B.13: Entry points defined in struct tcp_congestion_ops for the Linux CC framework in RDBv2

```c
u32 (*undo_cwnd)(struct sock *sk);
/* hook for packet ack accounting (optional) */
void (*pkts_acked)(struct sock *sk, u32 num_acked, s32 rtt_us);
/* call to send packet from tcp_write_xmit (optional) */
bool (*pkt_send)(struct sk_buff *xmit_skb, unsigned int mss_now,
                 struct sock *sk, struct tcp_sock *tp,
                 gfp_t gfp_mask, int loop_count,
                 bool retrans);
/* hook for ack received that removes packets from the write queue (optional) */
void (*tcp_ack)(struct sock *sk, struct sk_buff *skb, int flag,
                u32 prior_packets, u32 prior_snd_una, s32 sack_rtt);
/* get info for inet_diag (optional) */
void (*get_info)(struct sock *sk, u32 ext, struct sk_buff *skb);
char name[TCP_CA_NAME_MAX];
struct module *owner;
```

Defined in include/net/tcp.h

The function tcp_ack located in tcp_input.c handles incoming ACKs.

Code Listing B.14: tcp_ack function in the TCP engine handling incoming ACKs

```c
static int tcp_ack(struct sock *sk, const struct sk_buff *skb, int flag) {
    ...
    u32 prior_snd_una = tp->snd_una;
    u32 ack_seq = TCP_SKB_CB(skb)->seq;
    u32 ack = TCP_SKB_CB(skb)->ack_seq;
    int acked = 0; /* Number of packets newly acked */
    ...
    if (!flag & FLAG_SLOWPATH) && after(ack, prior_snd_una)) {
        ...
        tcp_ca_event(sk, CA_EVENT_FAST_ACK);
        ...
    } else {
        ...
        tcp_ca_event(sk, CA_EVENT_SLOW_ACK);
    }
    ...
    /* See if we can take anything off of the retransmit queue. */
    acked = tp->packets_out;
    flag |= tcp_clean_rtx_queue(sk, prior_packets, prior_snd_una,
                                sack_rtt_us);
    acked -= tp->packets_out;
    /* Advance cwnd if state allows */
    if (tcp_may_raise_cwnd(sk, flag))
        tcp_cong_avoid(sk, ack, acked);
    ...
}
```
B.4. TFRC Congestion Control implementation

Source Code

```c
/*
 * rdb_ack() - Perform loss detection
 */
void rdb_ack(struct sock *sk) {
    struct tcp_sock *tp = tcp_sk(sk);
    struct RDBcong *cong = inet_csk_ca(sk);
    struct sk_buff *acked = NULL;
    struct sk_buff *lost = NULL;
    int lost_count;

    lost_count = rdb_check_rtx_queue_acked(sk, tp->snd_una, &acked, &lost);
    if (lost_count) {
        cong->rdb_ack_loss_count += lost_count;
        cong->rdb_ack_loss_count_total += lost_count;
    }
    if (use_tfrc) {
        cong->tfrc.tx_stats->tx_rtt = tp->srtt_us >> 3;
        rdb_tfrc_update_loss_history(cong, sk, lost_count, lost);
    }
}
```

The function `rdb_ack` located in `rdb.cc` performs loss detection from incoming ACKs.

Code Listing B.15: Function `rdb_ack` in RDBv2

Source Code

```c
/**
 * rdb_check_rtx_queue_acked() - Perform loss detection
 * by analysing acks.
 * sk: the socket.
 * seq_acked: The sequence number that was acked.
 * acked: The sk_buff which will be set to the SKB
 * (with highest seqnum) that was acked.
 * lost: The sk_buff which will be set to the first SKB
 * (with lowest seqnum) that was lost.
 * Return: The number of packets that are presumed to be lost.
 */
static int rdb_check_rtx_queue_acked(struct sock *sk, u32 seq_acked, struct sk_buff **acked, struct sk_buff **lost) {
    struct tcp_sock *tp = tcp_sk(sk);
    struct sk_buff *skb, *tmp, *prev_skb = NULL;
    struct sk_buff *send_head = tcp_send_head(sk);
    struct tcp_skb_cb *scb;
    int fully_acked = true;
    int lost_count = 0;

    tcp_for_write_queue(skb, sk) {
        if (skb == send_head)
            break;
        scb = TCP_SKB_CB(skb);
    }
```
Appendix B. RDBv2 implementation source code

```c
/* Determine how many packets and what bytes were acked, no TSO support */
if (after(scb->end_seq, tp->snd_una)) {
    if (tcp_skb_pcount(skb) == 1 ||
        !after(tp->snd_una, scb->seq)) {
        break;
    }

    // We do not handle SKBs with gso_segs
    if (tcp_skb_pcount(skb)) {
        break;
    }
    fully_acked = false;
}

// Ackrs up to this packet
if (scb->end_seq == seq_acked) {
    *acked = skb;
    // This was sent with RDB data, and this skb acked data
    // on previous skb
    if (RDB_SKB_CB(skb)->rdb_start_seq != 0 && prev_skb) {
        tcp_for_write_queue(tmp, sk) {
            if (tmp == skb) // We have reached the acked SKB
                break;
            lost_count++;
            if (*lost == NULL)
                *lost = skb;
        }
    }
    break;
}
if (!fully_acked)
    break;
prev_skb = skb;
return lost_count;
```

The function `rdb_check_rtx_queue_acked` located in `rdb_cc.c` traverses the TCP output queue to detect lost packets.

Code Listing B.16: Function `rdb_check_rtx_queue_acked` in RDBv2

---

**Source Code**

```c
/*
** * rdb_tfrc_update_loss_history() - Update the loss history.
** * cong: The RDBCong struct for the connection.
** * lost_count: The number of packets believed to be lost.
** * lost: The first skb believed to be lost.
** */
void rdb_tfrc_update_loss_history(struct RDBcong *cong, struct sock *sk, int lost_count, struct sk_buff *lost) {
    struct rdb_tfrc_tx_stats* tx_stats = cong->tfrc.tx_stats;
    if (lost_count) {
        rdb_tfrc_handle_loss(cong->tfrc.li_hist, lost, lost_count,
                             tx_stats->tx_rtt, rdb_tfrc_first_li, sk);
    }
```
B.4. TFRC Congestion Control implementation

```c
// Increase interval length
rdb_tfrc_lh_update_i_mean(&cong->tfrc, 0);

if (cong->tfrc.li_hist->i_mean) {
    tx_stats->tx_p = scaled_div(1, cong->tfrc.li_hist->i_mean);
}

/* Update sending rate (step 4 of [RFC 3448, 4.3]) */
if (tx_stats->tx_p > 0) {
    // Calculate the send rate to use
    u32 tx_x_calc = tfrc_calc_x(tx_stats->tx_s, tx_stats->tx_rtt, tx_stats->tx_p);
    // We adjust the send rate by accounting for the network
    // and transport layer headers
    tx_x_calc = (tx_x_calc * tx_stats->tx_s) / (tx_stats->tx_s + 40);
    tx_stats->tx_x_calc = tx_x_calc;
}

// Recalculate allowed sending rate X
rdb_tx_update_x(cong, sk, NULL);
```

The function `rdb_tfrc_update_loss_history` located in `rdb_tfrc.c`.

Code Listing B.17: Function `rdb_tfrc_update_loss_history` in RDBv2

```
/*
 * rdb_tx_update_x() - Update allowed sending rate X
 * cong: The RDBCong struct for the connection
 * sk: the socket
 * stamp: most recent time if available (can be left NULL).
 * This function borrows from ccid3_hc_tx_update_x in dccp/ccids/ccid3.c (12/2014)
 */
void rdb_tx_update_x(struct RDBCong *cong, struct sock *sk, ktime_t *stamp) {

    struct rdb_tfrc_tx_stats* tx_stats = cong->tfrc.tx_stats;
    if (!tx_stats->tx_rtt)
        return;

    __u64 min_rate = tx_stats->tx_x >> 1; // Minimum rate is half the current rate
    const __u64 old_x = tx_stats->tx_x;
    ktime_t now = stamp ? *stamp : ktime_get_real();
    u32 delta = ktime_us_delta(now, tx_stats->tx_t_last_sent);
    u32 idle_rttts = delta / tx_stats->tx_rtt;
    /*
```
The function rdb_tx_update_x located in rdb_tfrc.c.

Code Listing B.18: Function rdb_tx_update_x in RDBv2

```c
24 * Handle IDLE periods: do not reduce below RFC3390 initial sending rate
25 * when idling [RFC 4342, 5.1]. Definition of idling is from rfc3448bis:
26 * a sender is idle if it has not sent anything over a 2-RTT-period.
27 * For consistency with X and X_recv, min_rate is also scaled by 2^6.
28 */
29 if (idle_rtt >= 2) {
30   min_rate = rfc3390_initial_rate(tx_stats);
31 }
32 if (tx_stats->tx_p > 0) {
33   tx_stats->tx_x = max(((u64)tx_stats->tx_x_calc) << 6,
34                         min_rate);
35 } else if (ktime_us_delta(now, tx_stats->tx_t_ld) - (s64)
36            tx_stats->tx_rtt >> 0) {
37   tx_stats->tx_x = max(tx_stats->tx_x,
38                        scaled_div(((u64)tx_stats->tx_s) << 6,
39                        tx_stats->tx_rtt));
40   tx_stats->tx_t_ld = now;
41 }
42 if (tx_stats->tx_x != old_x) {
43   u64 rtts_per_second = div64_u64(1000000, tx_stats->tx_rtt);
44   // Update the CWND value based on the calculated throughput
45   if (rtts_per_second) {
46     tx_stats->tx_x_rtt = div64_u64(tx_stats->tx_x >> 6,
47                                      rtts_per_second);
48     tx_stats->tx_x_cwnd = div64_u64((u64) tx_stats->
49                                      tx_x_rtt, tx_stats->tx_s);
50   }
51 }
52 }
```
if (k <= 0) {
    return;
}

skb_mstamp_get(&now);

// We start at the newest intervals (index 0)
for (i = 0; i <= k; i++) {
    tmp = tfrc_lh_get_interval(lh, i);
    i_i = tfrc_lh_get_interval_len(lh, i);
    if (i > 0) {
        // It's a short loss interval that spans less than two RTTs
        if (tmp->is_short) {
            i_i = div64_u64(tmp->li_length, tmp->li_lost);
        }
        i_totl += i_i * tfrc_lh_weights[i-1];
    } else {
        i_duration = skb_mstamp_us_delta(&now, &tmp->li_mstamp);
        cur_i_short = i_duration < (rtt * 2);
    }
    if (i < k) {
        i_tot0 += i_i * tfrc_lh_weights[i];
        w_tot += tfrc_lh_weights[i];
    }
    prev = tmp;
}

if (cur_i_short) {
    lh->i_mean = i_totl;
} else {
    lh->i_mean = max(i_tot0, i_totl);
}

/* Handle the extra restriction defined in rfc4828 to only include the latest interval if its older than 2 * RTT. */
if (k > 1) {
    struct rdb_tfrc_loss_interval *cur = tfrc_lh_peek(lh);
    ktime_t now = ktime_get_real();
    s64 diff = ktime_us_delta(now, cur->li_tstamp);
    // If the current loss interval I_0 is "short"
    if (diff < (rtt * 2)) {
        lh->i_mean = i_totl;
    }
}
lh->i_mean /= w_tot;

The function tfrc_lh_calc_i_mean_sp located in loss_history.c that calculates the average loss interval.

Code Listing B.19: Function tfrc_lh_calc_i_mean_sp in RDBv2
The function `tcp_rdbcong_avoid_tfrc` located in `rdb_cc.c`. This function is used when the TFRC loss response mechanism is enabled.

Code Listing B.20: Function `tcp_rdbcong_avoid_tfrc` located in `rdb_cc.c`.
B.4. TFRC Congestion Control implementation

The function `tcp_rdbcong_avoid` located in `rdb_cc.c`. This function is used when the TFRC loss response mechanism is enabled.

Code Listing B.21: Function `tcp_rdbcong_avoid` located in `rdb_cc.c`

```c
21 } }
22 else {
23     // Do reno.
24     tcp_reno_cong_avoid_3_15(sk, ack, acked);
25 }
26 }
```

The function `tcp_reno_ssthresh` located in `tcp_cong.c` returns the slow-start threshold.

Code Listing B.22: `ssthresh` implementation for TCP New Reno

```c
1 /* Slow start threshold is half the congestion window (min 2) */
2 u32 tcp_reno_ssthresh(struct sock *sk) {
3     const struct tcp_sock *tp = tcp_sk(sk);
4     return max(tp->snd_cwnd >> 1U, 2U);
5 }
```

The function `tcp_rdb_ssthresh` located in `rdb_cc.c` in the RDBv2 returns the slow-start threshold.

Code Listing B.23: `ssthresh` implementation in RDBv2

```c
1 /* Slow start threshold is set to a quarter of the congestion window (min 2) */
2 u32 tcp_rdb_ssthresh(struct sock *sk) {
3     const struct tcp_sock *tp = tcp_sk(sk);
4     return max(tp->snd_cwnd >> 2U, 2U);
5 }
```
Appendix C

Patches

C.1 RDB prototype v1 patch

```
---
1 include/linux/sysctl.h | 5 +
2 include/linux/tcp.h | 8 ++
3 include/net/sock.h | 5 +-.
4 include/net/tcp.h | 20 +++++
5 net/ipv4/sysctl_net_ipv4.c | 32 ++++++
6 net/ipv4/tcp.c | 195
7 ++++++++++++++++++++++++++++++++-------------------
8 net/ipv4/tcp_input.c | 148
9 ++++++++++++++++++++++++++++++++-------------------
10 net/ipv4/tcp_output.c | 181
11 ++++++++++++++++++++++++++++++++-------------------
12 net/ipv4/tcp_timer.c | 24 +++
13 9 files changed, 592 insertions(+), 26 deletions(-)
14
15 diff --git a/include/linux/sysctl.h b/include/linux/sysctl.h
16 index 483050c..f0edacd 100644
17 --- a/include/linux/sysctl.h
18 +++ b/include/linux/sysctl.h
19 @@ -355,6 +355,11 @@ enum
20 NET_IPV4_ROUTE=18,
21 NET_IPV4_FIB_HASH=19,
22 NET_IPV4_NETFILTER=20,
23 + NET_IPV4_TCP_FORCE_THIN_RDB=29, // Added @ Simula */
24 + NET_IPV4_TCP_FORCE_THIN_RM_EXPB=30, // Added @ Simula */
25 + NET_IPV4_TCP_FORCE_THIN_DUPACK=31, // Added @ Simula */
26 + NET_IPV4_TCP_RDB_MAX.BundlegetBytes=32, // Added @ Simula */
27 + NET_IPV4_TCP_TIMESTAMP=33,
28 NET_IPV4_TCP_WINDOW_SCALING=34,
29 diff --git a/include/linux/tcp.h b/include/linux/tcp.h
30 index c6b9f92..c11a564 100644
31 --- a/include/linux/tcp.h
```
Appendix C. Patches

```c
++ b/include/linux/tcp.h
@@ -97,6 +97,10 @@ enum {
  #define TCP_CONGESTION 13 /* Congestion control algorithm */
  #define TCP_MD5SIG 14 /* TCP MD5 Signature (RFC2385) */
  +#define TCP_THIN_RDB 15 /* Added @ Simula -
                   Enable redundant data bundling */
  +#define TCP_THIN_RM_EXPB 16 /* Added @ Simula -
                   Remove exponential backoff */
  +#define TCP_THIN_DUPACK 17 /* Added @ Simula -
                   Reduce number of dupAcks needed */
  +
  #define TCPI_OPT_TIMESTAMPS 1
  #define TCPI_OPT_SACK 2
  #define TCPI_OPT_WSCALE 4
@@ -296,6 +300,10 @@ struct tcp_sock {
  u8 nonagle; /* Disable Nagle algorithm? */
  u8 keepalive_probes; /* num of allowed keep alive probes */
  + u8 thin_rdb; /* Enable RDB */
  + u8 thin_rm_expb; /* Remove exp. backoff */
  + u8 thin_dupack; /* Remove dupack */
  +
  /* RTT measurement */
  u32 srtt; /* smoothed round trip time << 3 */
  u32 mdev; /* medium deviation */
-diff --git a/include/net/sock.h b/include/net/sock.h
-index dfeb8b1..af831d1 100644
-  --- a/include/net/sock.h
-+++ b/include/net/sock.h
@@ -462,7 +462,10 @@ static inline void sk_stream_set_owner_r(struct
                   sk->sk_stream_set_owner_r(s, skb, struct sock *sk)
  static inline void sk_stream_free_skb(struct sock *sk, struct
                   skb *skb)
-    {
-      skb_truesize_check(skb);
+      /* Modified @ Simula
+       skb_truesize_check creates unnecessary
+        noise when combined with RDB */
+      skb_truesize_check(skb);
+      sock_set_flag(sk, SOCK_QUEUE_SHRUNK);
+      sk->sk_wmem_queued = skb->truesize;
+      sk->sk_forward_alloc = skb->truesize;
-diff --git a/include/net/tcp.h b/include/net/tcp.h
-index 54053de..411cc9b 100644
-  --- a/include/net/tcp.h
-+++ b/include/net/tcp.h
@@ -188,9 +188,19 @@ extern void tcp_time_wait(struct sock *sk,
                   int state, int timeout);
  #define TCP_NAGLE_CORK 2 /* Socket is corked */
  #define TCP_NAGLE_PUSH 4 /* Cork is overridden for already
                   queued data */
+/* Added @ Simula - Thin stream support */
+ +#define TCP_FORCE_THIN_RDB 0 /* Thin streams: exp.
                   backoff default off */
+ +#define TCP_FORCE_THIN_RM_EXPB 0 /* Thin streams:
                   dynamic dupack default off */
+ +#define TCP_FORCE_THIN_DUPACK 0 /* Thin streams:
                   smaller minRTO default off */
```
```c
+define TCP_RDB_MAX_BUNDLE_BYTES 0 /* Thin streams: Limit maximum bundled bytes */
+ extern struct inet_timewait_death_row tcp_death_row;
+ /* sysctl variables for tcp */
+extern int sysctl_tcp_force_thin_rdb; /* Added @ Simula */
+extern int sysctl_tcp_force_thin_rm_expb; /* Added @ Simula */
+extern int sysctl_tcp_force_thin_dupack; /* Added @ Simula */
+extern int sysctl_tcp_rdb_max_bundle_bytes; /* Added @ Simula */
+ extern int sysctl_tcp_timestamps;
+ extern int sysctl_tcp_window_scaling;
+ extern int sysctl_tcp_sack;
+@ -723,6 +733,16 @@ static inline unsigned int tcp_packets_in_flight(const struct tcp_sock *tp)
  return (tp->packets_out - tp->left_out + tp->retrans_out);
+ /* Added @ Simula */
+ *
+ * To determine whether a stream is thin or not
+ * return 1 if thin, 0 otherwise
+ */
+ static inline unsigned int tcp_stream_is_thin(const struct tcp_sock *tp)
+{
+  return tp->packets_out < 4 ? 1 : 0;
+}
+ /* If cwnd > ssthresh, we may raise ssthresh to be half-way to cwnd.
+ * The exception is rate halving phase, when cwnd is decreasing towards
+ * ssthresh.
+ diff --git a/net/ipv4/sysctl_net_ipv4.c b/net/ipv4/sysctl_net_ipv4.c
+index 53ef0f4..58ac82b 100644
+- a/net/ipv4/sysctl_net_ipv4.c
+++ b/net/ipv4/sysctl_net_ipv4.c
@@ -187,6 +187,38 @@ static int strategy_allowed_congestion_control(ctl_table *table, int __user *nam
+         ctl_table ipv4_table[] = {
+          /* Added @ Simula for thin streams */
+          .ctl_name = NET_IPv4_TCP_FORCE_THIN_RDB,
+          .procname = tcp_force_thin_rdb,
+          .data = &sysctl_tcp_force_thin_rdb,
+          .maxlen = sizeof(int),
+          .mode = 0644,
+          .proc_handler = &proc_dointvec
+          },
+          /* Added @ Simula for thin streams */
+          .ctl_name = NET_IPv4_TCP_FORCE_THIN_RM_EXPB,
+          .procname = tcp_force_thin_rm_expb",
+          .data = &
+          sysctl_tcp_force_thin_rm_expb,
+          .maxlen = sizeof(int),
+          .mode = 0644,
```

139 + proc_handler = &proc_dointvec
140 + }, /* Added @ Simula for thin streams */
141 + {
142 + .ctl_name = NET_IPV4_TCP_FORCE_THIN_DUPACK,
143 + .procntype = "tcp_force_thin_dupack",
144 + .data = &
145 + ,
146 + .maxlen = sizeof(int),
147 + .mode = 0644,
148 + .proc_handler = &proc_dointvec
149 + }, /* Added @ Simula for thin streams */
150 + .ctl_name = NET_IPV4_TCP_RDB_MAX_BUNDLE_BYTES,
151 + .procntype = "tcp_rdb_max_bundle_bytes",
152 + .data = &
153 + ,
154 + .maxlen = sizeof(int),
155 + .mode = 0644,
156 + .proc_handler = &proc_dointvec
157 }
158 .ctl_name = NET_IPV4_TCP_TIMESTAMPS,
159 .procntype = "tcp_timestamps",
160 #if defined(__read_mostly)
161 int sysctl_tcp_fin_timeout __read_mostly = TCP_FIN_TIMEOUT;
162 +/* Added @ Simula */
163 +int sysctl_tcp_force_thin_rdb __read_mostly =
164 +TCP_FORCE_THIN_RDB;
165 +int sysctl_tcp_rdb_max_bundle_bytes __read_mostly =
166 +TCP_RDB_MAX_BUNDLE_BYTES;
167 +
168 #if defined(__read_mostly)
169 DEFINE_SNMP_STAT(struct tcp_mib, tcp_statistics) __read_mostly
170 ;
171 }
172 atomic_t tcp_orphan_count = ATOMIC_INIT(0);
173 @-658,6 +662,167 @@ static inline int select_size(struct sock *
174 *sk)
175 return tmp;
176 }
177 }
178 +#if defined(__read_mostly)
179 +/* Added at Simula to support RDB */
180 +static int tcp_trans_merge_prev(struct sock *sk, struct
181 +sk_buff *skb, int mss_now)
182 +{
183 + struct tcp_sock *tp = tcp_sk(sk);
184 +
185 + /* Make sure that this isn’t referenced by somebody else */
186 +
187 + if(!skb_cloned(skb))
188 + struct sk_buff *prev_skb = skb->prev;
189 + int skb_size = skb->len;
190 + int old_headlen = 0;
191 + int uad_data = 0;
192 + int uad_head = 0;
193 + int uad_frags = 0;
194 + int uad_nr_frags = 0;
195 + int uad_frags_diff = 0;
287

C.1. RDB prototype v1 patch

195 + /* Since this technique currently does not support SACK, I
196 +  return -1 if the previous has been SACK’d. */
197 +  if(TCP_SKB_CB(prev_skb)->sacked & TCP_CB_SACKED_ACKED){
198 +  return -1;
199 + }
200 + }
201 + /* Current skb is out of window. */
202 + if (after(TCP_SKB_CB(skb)->end_seq, tp->snd_una+tp->snd_wnd)
203 + ){
204 +  return -1;
205 + }
206 + /* TODO: Optimize this part with regards to how the
207 + variables are initialized */
208 + /* Calculates the ammount of unacked data that is available
209 + */
210 + ua_data = (TCP_SKB_CB(prev_skb)->end_seq - tp->snd_una >
211 +  prev_skb->len ? prev_skb->len :
212 +  TCP_SKB_CB(prev_skb)->end_seq - tp->snd_una);
213 + ua_frags_diff = ua_data - prev_skb->data_len;
214 + uad_frags = (ua_frags_diff > 0 ? prev_skb->data_len :
215 +  ua_data);
216 + uad_head = (ua_frags_diff > 0 ? ua_data - uad_frags : 0);
217 +
218 + if(uad_data <= 0)
219 +  return -1;
220 +
221 + if(uad_frags > 0){
222 +  int i = 0;
223 +  int bytes_frags = 0;
224 +  if(uad_frags == prev_skb->data_len)
225 +    ua_nr_frags = skb_shinfo(prev_skb)->nr_frags;
226 +  else{  
227 +    for(i=skb_shinfo(prev_skb)->nr_frags - 1; i>=0; i--){
228 +      if(skb_shinfo(prev_skb)->frags[i].size
229 +        + bytes_frags == uad_frags)
230 +        ua_nr_frags += 1;
231 +      break;
232 +    }
233 +  }
234 +  ua_nr_frags += 1;
235 +  bytes_frags += skb_shinfo(prev_skb)->frags[i].size;
236 +  }
237 +
238 +
239 + /* Do the diffrent checks on size and content, and return
240 + */
241 + if (something will not work.
242 +
243 + /* TODO: Support copying some bytes
244 + */
245 +
246 + /* 1. Larger than MSS.
247 + 2. Enough room for the stuff stored in the linear area
248 + * 3. Enough room for the pages
249 + * 4. If both skbs have some data stored in the linear area
250 + * also has some stored in the paged area, they cannot be
251 + * 5. If prev_skb is linear, then this one has to be it as
252 + */
+ if ((sysctl_tcp_rdb_max_bundle_bytes == 0 && ((skb_size +
|   ua_data) > mss_now))
+     || (sysctl_tcp_rdb_max_bundle_bytes > 0 && ((skb_size +
|     ua_data) >
|     sysctl_tcp_rdb_max_bundle_bytes))){
+     return -1;
+ }
+ }
+ /* We need to know tailroom, even if it is nonlinear */
+ if(uad_head > (skb->end - skb->tail)){
+     return -1;
+ }
+ if(skb_is_nonlinear(skb) && (uad_frags > 0)){
+     if((ua_nr_frags +
+         skb_shinfo(skb)->nr_frags) > MAX_SKBFRAGS){
+         return -1;
+     }
+     if(skb_headlen(skb) > 0){
+         return -1;
+     }
+ }
+ /* To avoid duplicate copies (and copies
+     where parts have been acked) */
+ if(TCP_SKCB(skb)->seq <= (TCP_SKCB(prev_skb)->end_seq -
+     ua_data)){
+     return -1;
+ }
+ /*SYN's are holy*/
+ if(TCP_SKCB(skb)->flags & TCPCB_FLAG_SYN || TCP_SKCB(skb)->flags & TCPCB_FLAG_FIN){
+     return -1;
+ }
+ /* Copy linear data */
+ if(uad_head > 0){
+     /* Add required space to the header. Can't use put due to
+      linearity */
+     old_headlen = skb_headlen(skb);
+     skb->tail += uad_head;
+     skb->len += uad_head;
+     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);
+ }
+ /*Copy paged data*/
+ if(uad_frags > 0){
+     int i = 0;
+     /*Must move data backwards in the array.*/
+     if(skb_is_nonlinear(skb)){
+         memmove(skb_shinfo(skb)->frags + ua_nr_frags,
+                 skb_shinfo(skb)->frags,
+                 skbuad_nr_frags(skb) * skb_shinfo(skb)->frags_size);
+     else if(skb_is_paged(skb)){
+         int j = 0;
+         skb_copy_to_linear_data(skb, prev_skb->data + (skb_headlen(prev_skb) - uad_head), uad_head);
+     }
C.1. RDB prototype v1 patch

```c
 312  + skb_shinfo(skb)->nr_frags*sizeof(skb_frag_t));
 313  + }
 314  + /*Copy info and update pages*/
 315  + memcpy(skb_shinfo(skb)->frags,
 316  + skb_shinfo(prev_skb)->frags + (skb_shinfo(prev_skb)
 317  + )->nr_frags - ua_nr_frags),
 318  + ua_nr_frags*sizeof(skb_frag_t));
 319  +
 320  + for(i=0; i<ua_nr_frags;i++){
 321  +   get_page(skb_shinfo(skb)->frags[i].page);
 322  + }
 323  +
 324  + skb_shinfo(skb)->nr_frags += ua_nr_frags;
 325  + skb->data_len += uad_frags;
 326  + skb->len += uad_frags;
 327  + }
 328  + TCP_SKB_CB(skb)->seq = TCP_SKB_CB(prev_skb)->end_seq -
 329  + ua_data;
 330  +
 331  + if(skb->ip_summed == CHECKSUM_PARTIAL)
 332  + skb->csum = CHECKSUM_PARTIAL;
 333  + else
 334  + skb->csum = skb_checksum(skb, 0, skb->len, 0);
 335  + }
 336  +
 337  + return 1;
 338  +}
 339  +
 340  + int tcp_sendmsg(struct kiocb *iocb, struct socket *sock,
 341  + struct msghdr *msg,
 342  + size_t size)
 343  + {
 344  + @-825,6 +990,16 @@ new_segment:
 345  +  from += copy;
 346  +  copied += copy;
 347  +  /* Added at Simula to support RDB */
 348  +  if(((tp->thin_rdb || sysctl_tcp_force_thin_rdb) &
 349  +     skb->len < mss_now) &
 350  +     (skb->prev != (struct sk_buff*) &sk->sk_write_queue
 351  +      &
 352  +      (TCP_SKB_CB(skb)->flags & TCP_CB_FLAG_SYN)
 353  +      &
 354  +      tcp_trans_merge_prev(sk, skb, mss_now);
 355  +      )}
 356  +  } /* End - Simula */
 357  +
 358  + if ((seglen -= copy) == 0 &
 359  +      iovlen == 0) goto out;
 360  +
 361  + @-1870,7 +2845,25 @@ static int do_tcp_setsockopt(struct sock
 362  + *sk, int level,
 363  + tcp_push_pending_frames(sk);
 364  + } break;
 365  +
 366  + /* Added @ Simula. Support for thin streams */
 367  + case TCP_THIN_RDB:
 368  +    if (val)
 369  +      tp->thin_rdb = 1;
 370  +    break;
 371  +```
Appendix C. Patches

```c
+ /* Added @ Simula. Support for thin streams */
+ case TCP_THIN_RM_EXPB:
+     if (val)
+       tp->thin_rm_expb = 1;
+ break;
+ /* Added @ Simula. Support for thin streams */
+ case TCP_THIN_DUPACK:
+     if (val)
+       tp->thin_dupack = 1;
+ break;
+ case TCP_KEEPIDLE:
+     if (val < 1 || val > MAX_TCP_KEEPIDLE)
+       err = -EINVAL;
+     dif
+     ++ b/net/inet4/tcp_input.c
+ # @ -89,6 +89,9 @@ int sysctl_tcp_frt0 __read_mostly;
+ |
+ # @ -89,6 +89,9 @@ int sysctl_tcp_frt0_response __read_mostly;
+ int sysctl_tcp_nometrics_save __read_mostly;
+ /* Added @ Simula */
+ int sysctl_tcp_force_thin_dupack __read_mostly =
+   TCP_FORCE_THIN DUPACK;
+ int sysctl_tcp moderate_rcvbuf __read_mostly = 1;
+ int sysctl_tcp_abc __read_mostly;
+ @ -1704,6 +1707,12 @@ static int tcp_time_to_recover(struct
+ sock *sk) */
+     return 1;
+ } }
+ /* Added at Simula to modify fast retransmit */
+ if (((tp->thin_dupack || sysctl_tcp_force_thin_dupack) &&
+   tcp_fackets_out(tp) > 1 && tcp_stream_is_thin(tp))(
+   return 1;
+ )
+ return 0;
+ @ -2437,30 +2446,127 @@ static int tcp_clean_rtx_queue(struct
+ sock *sk, __u32 *seq_rtt_p)
+ { struct tcp_sock *tp = tcp_sk(sk);
+ const struct inet_connection_sock *icsk = inet_csk(sk);
+ struct sk_buff *skb;
+ struct sk_buff *skb = tcp_write_queue_head(skb);
+ struct sk_buff *next_skb;
+ __u32 now = tcp_time_stamp;
+ int acked = 0;
+ int prior_packets = tp->packets_out;
+ /* Added at Simula for RDB support*/
+ __u8 done = 0;
+ int remove = 0;
+ int remove_head = 0;
+ int remove_frags = 0;
+ int no_frags;
+ int data_frags;
+ int i;```
C.1. RDB prototype v1 patch

```
434 + __s32 seq_rtt = -1;
435 + ktime_t last_ackt = net_invalid_timestamp();
436 +
437 + while ((skb = tcp_write_queue_head(sk)) &&
438 + skb != tcp_send_head(sk)) {
439 +
440 + while (skb != NULL)
441 + &
442 + ((tp->thin_rdb || sysctl_tcp_forceThin_rdb)
443 + &
444 + skb != tcp_send_head(sk)
445 + &
446 + skb != (struct sk_buff *)&sk->sk_write_queue)
447 + || ((tp->thin_rdb || sysctl_tcp_forceThin_rdb)
448 + &
449 + skb !(struct sk_buff *)&sk->sk_write_queue))
450 +
451 + struct tcp_skb_cb *scb = TCP_SKCB(skb);
452 + __u8 sacked = scb->sacked;
453 +
454 + if(skb == NULL)
455 + break;
456 +
457 +
458 + if(skb == tcp_send_head(sk))
459 + break;
460 +
461 +
462 + /* If our packet is before the ack sequence we can
463 + * discard it as it's confirmed to have arrived at
464 + * the other end.
465 + */
466 + if((after(skb->end_seq, tp->snd_una)) {
467 + if (tcp_skb_pcount(skb) > 1 &&
468 + after(tp->snd_una, skb->seq))
469 + acked |= tcp_tso_acked(skb, skb,
470 + now, &seq_rtt);
471 +
472 + break;
473 +
474 + if (tcp_skb_pcount(skb) > 1 && after(tp->snd_una, skb->seq))
475 + acked |= tcp_tso_acked(skb, skb, now, &seq_rtt);
476 +
477 +
478 + /* Added at Simula for RDB support*/
479 + if (((tp->thin_rdb || sysctl_tcp_forceThin_rdb) && after(
480 + tp->snd_una, skb->seq)) {
481 + if (!skb_cloned(skb) && !(skb->flags & TCPCB_FLAG_SYN))
482 +
483 + remove = tp->snd_una - skb->seq;
484 + remove_head = (remove > skb_headlen(skb) ?
485 + skb_headlen(skb) : remove);
486 + remove_frags = (remove > skb_headlen(skb) ?
487 + remove - remove_head : 0);
488 +
489 + /* Has linear data */
490 + if(skb_headlen(skb) > 0 && remove_head > 0){
491 + memmove(skb->data,
492 + skb->data + remove_head,
493 + skb_headlen(skb) - remove_head);
494 +
495 + skb->tail -= remove_head;
496 +
497 + }
```
Appendix C. Patches

```c
if(skb_is_nonlinear(skb) && remove_frags > 0){
  no_frags = 0;
  data_frags = 0;
  /*Remove unecessary pages*/
  for(i=0; i<skb_shinfo(skb)->nr_frags; i++){/gif
    if(data_frags + skb_shinfo(skb)->frags[i].size
      == remove_frags){
      put_page(skb_shinfo(skb)->frags[i].page);
      no_frags += 1;
      break;
    }
    else
    
      put_page(skb_shinfo(skb)->frags[i].page);
      no_frags += 1;
    data_frags += skb_shinfo(skb)->frags[i].size;
  }
  /*/}
  if(skb_shinfo(skb)->nr_frags > no_frags)
    memmove(skb_shinfo(skb)->frags, 
            skb_shinfo(skb)->frags + no_frags, 
            (skb_shinfo(skb)->nr_frags 
             - no_frags)*sizeof(skb_frag_t));
  skb->data_len -= remove_frags;
  skb_shinfo(skb)->nr_frags -= no_frags;
  scb->seq += remove;
  skb->len -= remove;
  if(skb->ip_summed == CHECKSUM_PARTIAL)
    skb->csum = CHECKSUM_PARTIAL;
  else
    skb->csum = skb_checksum(skb, 0, skb->len, 0);
  /*/}
  if(done == 1 || tcp_skb_is_last(sk,skb)){
    break;
  } else if(done == 2){
    skb = skb->next;
    done = 1;
    continue;
  }
  /*/
  
  /* Initial outgoing SYN's get put onto the write_queue 
  * just like anything else we transmit. It is not
  * true data, and if we misinform our callers that
  @@ -2474,14 +2580,14 @@ static int tcp_clean_rtx_queue(struct 
  sock *sk, __s32 *seq_rtt_p)
    acked |= FLAG_SYN_ACKED;
    tp->retrans_stamp = 0;
  }
  ...
```
C.1. RDB prototype v1 patch

```c
/* MTU probing checks */
if (icsk->icsk_mtuup.probe_size) {
  if (!after(tp->mtu_probe.probe_seq_end, TCP_SKB_CB(skb)->
    end_seq)) {
    tcp_mtuup_probe_success(sk, skb);
  }
}
+
if (sacked) {
  if (sacked & TCP_CB_RETRANS) {
    @@ -2505,24 +2611,32 @@ static int tcp_clean_rtx_queue(struct sock sk, __s32 *seq_rtt_p)
      seq_rtt = now - scb->tstamp;
      last_ackt = skb->tstamp;
    }
    + tcp_dec_pcount_approx(&tp->packets_out, skb);
    tcp_packets_out_dec(tp, skb);
    + next_skb = skb->next;
    tcp_unlink_write_queue(skb, sk);
    sk_stream_free_skb(skb, sk);
    clear_all_retrans_hints(tp);
    + /* Added at Simula to support RDB */
    + skb = next_skb;
  }
  + tcp_send_head(skb) {
    tcp_advance_send_head(skb);
    tcp_dec_pcount_approx(&tp->packets_out, skb);
    tcp_packets_out_dec(tp, skb);
    next_skb = skb->next;
    tcp_unlink_write_queue(skb, sk);
    sk_stream_free_skb(skb, sk);
    clear_all_retrans_hints(tp);
    + /* Added at Simula to support RDB */
    + skb = next_skb;
}
+
if (ackd&FLAG_ACKED) {
  u32 pkts_acked = prior_packets - tp->packets_out;
  const struct tcp_congestion_ops *ca_ops
    = inet_csk(skb)->icsk_ca_ops;
  tcp_ack_update_rtt(skb, acked, seq_rtt);
  tcp_ack_packets_out(skb);
  +
  if (ca_ops->pkts_acked) {
    s32 rtt_us = -1;
    001 -
    + /* Is the ACK triggering packet unambiguous? */
    if (!((ackd & FLAG_RETRANS_DATA_ACKED)) {
      /* High resolution needed and available? */
      diff --git a/net/ipv4/tcp_output.c b/net/ipv4/tcp_output.c
      index 666d8a5..daa580d 100644
      --- a/net/ipv4/tcp_output.c
      +++ b/net/ipv4/tcp_output.c
      @@ -1653,7 +1653,7 @@ static void tcp_retrans_tryCollapse(
        struct sock *sk, struct sk_buff *skb, int m
      BUG_ON(tcp_skb_pcount(skb) != 1 ||
        tcp_skb_pcount(next_skb) != 1);
    -
    + /* changing transmit queue under us so clear hints */
    clear_all_retrans_hints(tp);
```

@ -1702,6 +1702,166 @ static void tcp_retrans_tryCollapse(
 struct sock *sk, struct sk_buff *skb, int m
 }
 }
+ /* Added at Simula. Variation of the regular collapse,  
+ * adapted to support ROB */
+ static void tcp_retrans_merge_redundant(struct sock *sk,  
+ struct sk_buff *skb,  
+ int mss_now)
+ {
+ struct tcp_sock *tp = tcp_sk(sk);
+ struct sk_buff *next_skb = skb->next;
+ skb_size = skb->len;
+ int new_data = 0;
+ int new_data_head = 0;
+ int new_data_frags = 0;
+ int new_frags = 0;
+ int old_headlen = 0;
+ +
+ int i;
+ +
+ +
+ int data_frags = 0;
+ +
+ /* Loop through as many packets as possible  
+ *(will create a lot of redundant data, but WHATEVER).  
+ * The only packet this MIGHT be critical for is  
+ * if this packet is the last in the retrans-queue.  
+ *  
+ * Make sure that the first skb isn't already in  
+ * use by somebody else. */
+ +
+ if (!skb_cloned(skb)) {
+ /* Iterate through the retransmit queue */
+ for (; (next_skb != (sk)->sk_send_head) &&
+ (next_skb != (struct sk_buff *) &sk)->
+ sk_write_queue);
+ next_skb = next_skb->next) {
+ +
+ /* Reset variables */
+ new_frags = 0;
+ data_frags = 0;
+ new_data = TCP_SK_CB(next_skb)->end_seq - TCP_SK_CB(skb  
+ )->end_seq;
+ +
+ /* New data will be stored at skb->start_add +  
+ some_offset.  
+ * in other words the last N bytes */
+ +
+ new_data_frags = (new_data > next_skb->data_len ?
+ next_skb->data_len : new_data);
+ +
+ new_data_head = (new_data > next_skb->data_len ?
+ new_data - skb->data_len : 0);
+ +
+ /*
+ * 1. Contains the same data  
+ * 2. Size  
+ * 3. Sack  
+ * 4. Window  
+ * 5. Cannot merge with a later packet that has linear  
+ * 6. The new number of frags will exceed the limit  
+ * 7. Enough tailroom  
+ */
+ +
+ if(new_data <= 0){
+ return;
C.1. RDB prototype v1 patch

```c
if ((sysctl_tcp_rdb_max_bundle_bytes == 0) && ((skb_size + new_data) > mss_now))
    sysctl_tcp_rdb_max_bundle_bytes = skb_size;
new_data = mss_now;
    return;

else {
    if(TCP_SKCB_CB(next_skb)->flags & TCB_FLAG_FIN)
        return;

    else {
        if(TCP_SKCB_CB(skb)->sacked & TCB_PKT_ACKED) ||
            TCP_SKCB_CB(next_skb)->sacked & TCB_PKT_ACKED)
            return;

        else {
            if(after(TCP_SKCB_CB(skb)->end_seq + new_data, tp->snd_una + tp->snd_wnd))
                return;

            else {
                if(skb_shinfo(skb)->frag_list || skb_shinfo(skb)->frag_list)
                    return;

                else {
                    /* Calculate number of new fragments. Any new data will be
                       stored in the back. */
                    i = skb_shinfo(next_skb)->nr_frags == 0 ?
                        0 : skb_shinfo(next_skb)->nr_frags - 1;
                    for (; i>=0; i--){
                        if(data_frags + skb_shinfo(next_skb)->frags[i].size ==
                            new_data_frags)
                            new_frags += 1;
                    }

                    data_frags += skb_shinfo(next_skb)->frags[i].size;
                    new_frags += 1;
                }
            }
            /* If dealing with a fragmented skb, only merge
               with an skb that ONLY contain frags */
            else {
                /* Due to the way packets are processed, no later data*/
                if(skb_headlen(next_skb) && new_data_head > 0)
                    return;

                else {
                    if(skb_is_nonlinear(next_skb) &&
                        new_data_frags > 0) &&
                        (sk_buff_headlen(skb) >
                            MAX skb FRAGS))
                        return;

                    else {
```

if(skb_headlen(next_skb) && (new_data_head > (skb->end - skb->tail)))
{
    return;
}
}

/* Copy linear data. This will only occur if both are linear, */
if(skb_headlen(next_skb) && (new_data_head > 0)){
    old_headlen = skb_headlen(skb);
    skb->tail += new_data_head;
    skb->len += new_data_head;

    /* The new data starts in the linear area, and the correct offset will then be given by */
    // removing new_data amount of bytes from length. */
    skb_copy_to_linear_data_offset(skb, old_headlen,
        next_skb->tail - new_data_head, new_data_head);
}

if(skb_is_nonlinear(next_skb) && (new_data_frags > 0)){
    memcpy(skb_shinfo(skb)->frags + skb_shinfo(skb)->nr_frags,
        skb_shinfo(next_skb)->frags +
        (skb_shinfo(next_skb)->nr_frags - new_frags),
        new_frags*sizeof(skb_frag_t));

    for(i=skb_shinfo(skb)->nr_frags;
        i < skb_shinfo(skb)->nr_frags + new_frags; i++)
        get_page(skb_shinfo(skb)->frags[i].page);

    skb_shinfo(skb)->nr_frags += new_frags;
    skb->data_len += new_data_frags;
    skb->len += new_data_frags;
}

TCP_SKB_CB(skb)->end_seq += new_data;

if(skb->ip_summed == CHECKSUM_PARTIAL)
    skb->csum = CHECKSUM_PARTIAL;
else
    skb->csum = skb_checksum(skb, 0, skb->len, 0);

skb_size = skb->len;

/* Do a simple retransmit without using the backoff mechanisms in */
/* tcp_timer. This is used for path mtu discovery. */
/* The socket is already locked here. */

tcp_simple_retransmit(struct sock * sk)
/* This retransmits one SKB. Policy decisions and retransmit queue */
/* state updates are done by the caller. Returns non-zero if an */
/* error occurred which prevented the send. */
/* Modified at Simula to support thin stream optimizations */
/* TODO: Update to use new helpers (like tcp_write_queue_next ()
*/
C.1. RDB prototype v1 patch

```c
/*
 * int tcp_retransmit_skb(struct sock *sk, struct sk_buff *skb)
 */
794 int tcp_retransmit_skb(struct sock *sk, struct sk_buff *skb) {
795 @-1882.10 +1964.21 @ int tcp_retransmit_skb(struct sock *sk, 
796 struct sk_buff *skb) {
797 (skb->len < (cur_mss >> 1)) &&
798 tcp_write_queue_next(sk, skb) != tcp_send_head(sk)) &&
799 (!tcp_skb_is_last(sk, skb)) &&
800 - (skb_shinfo(skb)->nr_frags == 0 && skb_shinfo( 
801 tcp_write_queue_next(sk, skb))->nr_frags == 0) &&
802 - (tcp_skb_pcount(skb) == 1 && tcp_skb_pcount( 
803 tcp_write_queue_next(sk, skb)) == 1) &&
804 - (sysctl_tcp_retrans Collapse skb_skb->nr_frags == 0
805 + skb_skb->nr_frags == 0)
806 + 
807 +
808 + else
809 tcp_retrans merge_redundant(skb_skb, cur_mss);
810 +
811 +
812 +
813 +
814 +
815 +
816 +
817 +
818 +
819 if (inet_csk(sk)->icsk_af_ops->rebuild_header(sk))
820 return -EHOSTUNREACH; /* Routing failure or similar. */
821 diff --git a/net/ipv4/tcp_timer.c b/net/ipv4/tcp_timer.c
822 index e9b151b..addde35 100644
823 --- a/net/ipv4/tcp_timer.c
824 +++ b/net/ipv4/tcp_timer.c
825 @@ -32,6 +32,9 @@ int sysctl_tcp_retries1 __read_mostly = 
826 TCP_RETR1;
827 int sysctl_tcp_retries2 __read_mostly = TCP_RETR2;
828 int sysctl_tcp_orphan_retries __read_mostly;
829 +/* Added @ Simula */
830 +int sysctl_tcp_force thin_rm_expb __read_mostly = 
831 TCP_FORCE_THIN_RM_EXPB;
832 +
833 static void tcp_write_timer(unsigned long);
834 static void tcp_delack_timer(unsigned long);
835 static void tcp_keepalive_timer (unsigned long data);
836 @-368,13 +371,28 @@ static void tcp_retransmit_timer(struct 
837 sock *skk)
838 */
839 icsk->icsk_backoff++;
840 icsk->icsk_retransmits++;
841 -
842 + out_reset_timer:
843 - icsk->icsk_rto = min(icsk->icsk_rto << 1, TCP_RTO_MAX);
844 + /* Added @ Simula removal of exponential backoff for thin 
845 + We only want to apply this for an established stream */
846 + if ((tp->thin_rm_expb || (sysctl_tcp_force_thin_rm_expb)
847 + && tcp_stream_is_thin(tp) && sk->sk_state == 
848 TCP_ESTABLISHED) {
```
Appendix C. Patches

This is the initial patch sent to Linux devs containing all the three thin-stream modifications.

Code Listing C.1: Patch for RDBv1, mFR and LT for Linux kernel 2.6.23
C.2 Netem with fixed loss

Source Code

```c
#include <uapi/linux/pkt_sched.h>

#define NETEM_LOSS_UNSPEC, NETEM_LOSS_GI, /* General Intuitive - 4 state model */
#define NETEM_LOSS_GE, /* Gilbert Elliot models */
#define NETEM_LOSS_FIXED, /* Lose fixed packets */
#define __NETEM_LOSS_MAX

#define NETEM_LOSS_MAX (__NETEM_LOSS_MAX - 1)
#endif

/** Fixed loss model */
struct tc_netem_fixedmodel {
  /**< u32 loss[50];
  int _u32 loss_length;
  /**< u32 flow_length;
  /**< u32 packets_processed;
  /**< u32 packets_dropped;
  /**< u32 verbose;
  
#endif

#define NETEM_DIST_SCALE 8192
#define NETEM_DIST_MAX 16384

#include <linux/ftrace.h>
#include <net/tcp.h>

#define VERSION "1.3"

/* Network Emulation Queuing algorithm.
 @ -84.6 +87.7 @ struct netem_sched_data {
  /**< u32 ecn;
  /**< u32 limit;
  /**< u32 counter;
  /**< u32 counter_fixed;
  /**< u32 gap;
  /**< u32 duplicate;
  /**< u32 reorder;
  @ -108.6 +112.7 @ struct netem_sched_data {
  /**< CLG_RANDOM,
  /**< CLG_4_STATES,
  /**< CLG_GILB_ELL,
  /**< CLG_FIXED,
  }
 } loss_model;
```
Appendix C. Patches

```c
/* Correlated Loss Generation models */
@@ -123,6 +128,16 @@ struct netem_sched_data {
    u32 a5; /* p23 used only in 4-states */
 } clg;

+ struct fixed_loss_model {
+     __u32 loss_length; /* The number of packets to drop */
+     __u32 loss[50]; /* Array containing which packets to drop */
+     __u32 loss_index; /* Current index in loss */
+     __u32 flow_length; /* Length of the packet flow to work on */
+     __u32 flow_accounting; /* Counts from 0 to flow_length */
+     __u32 packets_processed;
+     __u32 packets_dropped;
+     __u32 verbose;
+ } flm;
};

/* Time stamp put into socket buffer control block
@@ -279,6 +294,33 @@ static bool loss_gilb_ell(struct
    return false;
 }
+static bool loss_fixed(struct netem_sched_data *q)
+{
+    bool drop = false;
+    struct fixed_loss_model *flm = &q->flm;
+    
+    + flm->packets_processed++;
+    + // Advance the flow accounting (which packet in the flow)
+    + if (++(flm->flow_accounting) == flm->flow_length) 
+        + flm->flow_accounting = 0;
+    +
+    + // Drop packet
+    + if (flm->flow_accounting == flm->loss[flm->loss_index]) { 
+        + drop = true;
+        + flm->packets_dropped++;
+        + if (++(flm->loss_index) == flm->loss_length) 
+            + flm->loss_index = 0;
+        +
+        + if (flm->verbose) { 
+            + printk(KERN_INFO "NETEM: Dropping packet \%d, next to drop:
+                \%d (index: \%d) (Total: \%d)\n", flm->flow_accounting,
+                + flm->loss[flm->loss_index], flm->loss_index, flm->
+                + packets_dropped);
+        + }
+    }
+    +
+    + return drop;
+  +
+    +
+    +
+    + static bool loss_event(struct netem_sched_data *q)
+    + {
+        switch (q->loss_model) {
+        @@ -301,6 +343,9 @@ static bool loss_event(struct
+            + the kernel logs
+        */
+        + return loss_gilb_ell(q);
+        +
+        + return loss_fixed(q);
```
C.2. Netem with fixed loss

```c
C.2. Netem with fixed loss

118  }
119  return false; /* not reached */
120  @} -406,6 +451,7 @) static int netem_enqueue(struct sk_buff * skb, struct Qdisc *sch)
121  /* We don't fill cb now as skb_unshare() may invalidate it */
122  struct netem_skb_cb *cb;
123  struct sk_buff *sk2;
124  + struct fixed_loss_model *flm = &q->flm;
125  int count = 1;
126  +
127  */ Random duplication */
128  @} -414,13 +460,30 @} static int netem_enqueue(struct sk_buff * skb, struct Qdisc *sch)
129  /* Drop packet? */
130  if (loss_event(q)) {
131 - if (q->ecn && INET_ECN_set_ce(skb))
132 + if (q->ecn && INET_ECN_set_ce(skb)) {
133 + qstats.drops++ ; /* mark packet */
134   - else
135 +   if (flm->verbose) {
136 +     printk(KERN_INFO "NETEM: Setting ecn and increased qstats.drops to \%d\n", sch->qstats.drops);
137 +   } +
138 +   }
139   +
140   +
141   + else {
142 +     --count;
143   +   }
144 +
145 +
146 + if (count == 0) {
147 +   sch->qstats.drops++ ;
148 +   +
149 +   + if (flm->verbose) {
150 +     unsigned int hash = 0;
151   +     if (skb->sk) {
152 +       hash = (skb->sk)->sk_hash;
153 +     +
154 +     printk(KERN_INFO "NETEM: Drop: HASH: %u, SEQ: %u ENDSEQ: %u WHEN: %u ACK_SEQ: %u\n", hash,
155 +   + TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq,
156 +   + TCP_SKB_CB(skb)->when, TCP_SKB_CB(skb)->ack_seq);
157 +   +
158 +   +
159 +   kfree_skb(skb);
160 +   return NET_XMIT_SUCCESS | __NET_XMIT_BYPASS;
161 +   }
162 @} -501,6 +564,7 @} static int netem_enqueue(struct sk_buff * skb, struct Qdisc *sch)
163 + cb->time_to_send = now + delay;
164 + cb->tstamp_save = skb->tstamp;
165 + q->counter ;
166 + ++q->counter_fixed ;
167 + tfifo_enqueue(skb, sch);
168 } else {
169 /*
170 @} -766,6 +830,28 @} static int get_loss_clg(struct Qdisc *sch, const struct nlattr *attr)
171 + break ;
172 +
173 + case NETEM_LOSS_FIXED: {
174 + const struct tc_netem_fixedmodel *m = nla_data(la);
175 + int i = 0;
```
Code Listing C.2: Netem fixed loss patch

```c
q->loss_model = CLG_FIXED;
q->flm.loss_length = m->loss_length;
q->flm.flow_length = m->flow_length;
q->flm.loss_index = 0;
q->flm.flow_accounting = 0;
q->flm.packets_processed = 0;
q->flm.verbose = m->verbose;

printk(KERN_INFO "NETEM: Setting netem loss to fixed rate ": "");
while (i < q->flm.loss_length) {
  q->flm.loss[i] = m->loss[i];
  printk(" %"u", q->flm.loss[i]);
  i++;
}
printk(KERN_INFO "NETEM: Flow length: %d\n", q->flm.
  flow_length);
printk(KERN_INFO "NETEM: Verbose level: %d\n", q->flm.
  verbose);
break;
}
default:
  pr_info("netem: unknown loss type %u\n", type);
  return -EINVAL;
@@ -825,6 +911,7 @@ static int netem_change(struct Qdisc *sch,
 struct nlaattr *opt)
 q->limit = oopt->limit;
 q->gap = oopt->gap;
 q->counter = 0;
+q->counter_fixed = 0;
 q->loss = oopt->loss;
 q->duplicate = oopt->duplicate;
 nla_nest_end(skb, nest);
@@ -929,6 +1016,23 @@ static int dump_loss_model(const struct
 netem_sched_data *q,
 goto nla_put_failure;
 break;
 }
case CLG_FIXED: {
+  struct tc_netem_fixedmodel fm = {
+    .loss_length = q->flm.loss_length,
+    .flow_length = q->flm.flow_length,
+    .packets_processed = q->flm.packets_processed,
+    .packets_dropped = q->flm.packets_dropped,
+    .verbose = q->flm.verbose,
+  };
+  int i;
+  for (i = 0; i < q->flm.loss_length; i++) {
+    fm.loss[i] = q->flm.loss[i];
+  }
+  if (nla_put(skb, NETEM_LOSS_FIXED, sizeof(fm), &fm))
+    goto nla_put_failure;
+  break;
  }
  int i;
  for (i = 0; i < q->flm.loss_length; i++) {
    fm.loss[i] = q->flm.loss[i];
  }
  if (nla_put(skb, NETEM_LOSS_FIXED, sizeof(fm), &fm))
    goto nla_put_failure;
  break;
  }

nla_nest_end(skb, nest);
```
C.2. Netem with fixed loss

```c
#include <linux/pkt_sched.h>

enum {
    netem_loss_unspec, // General Intuitive - 4 state model
    netem_loss_GI,     // Gilbert Elliot models
    + netem_loss_FIXED, // Lose every Nth
    + netem_loss_MAX
};

#define NETEM_LOSS_MAX (__NETEM_LOSS_MAX - 1)

#include "linux/pkt_sched.h"

struct tc_netem_gemodel {
    __u32 k[ ];
    __u32 loss[ ];
    __u32 loss_length; // Number of specified packets to lose
    __u32 flow_length;
    __u32 packets_processed;
    __u32 packets_dropped;
    __u32 verbose;
    +);
    +);
    #define NETEM_DIST_SCALE 8192
    #define NETEM_DIST_MAX 16384
}

static void explain( ) {
    loss random PERCENT [CORRELATION]|n |
    loss state P13 [P31 [P32 [P23 P14]]]|n |
    loss gmodel PERCENT [R [1-H [1-K]]]|n |
    + [ loss fixed NTH [NTH] ... |n |
    [ reorder PRECENT [CORRELATION] [ gap
    DISTANCE ]|n |
    [ rate RATE [PACKETOVERHEAD] [CELLSIZE] |
    CELLOVERHEAD]|n |);
}

static int parse_fixed_loss(char *str, uint32_t *store) {
    +
```
+ char *endptr;
+ long val;
+ int range = 0;
+ int MAX LOSS LENGTH = 49;
+ uint32_t last_value = -1;
+ int loss_list_index = 0;
+ long next_digit(char *str, char **endptr, long *result) {
    errno = 0; /* To distinguish success/failure after call */
    *result = strtol(str, endptr, 10);
    + /* Check for various possible errors */
    + if ((errno == ERANGE && (*result == LONG_MAX || *result ==
    + LONG_MIN))
    + || (errno != 0 && *result == 0)) {
    + return -1;
    + }
    + // No digits found
    + if (*endptr == str) {
    + return 0;
    + }
    + return 1;
    + }
    + while (1) {
    + int ret = next_digit(str, &endptr, &val);
    + if (ret == -1) {
    + fprintf(stderr, "Failed to parse '%s'\n", str);
    + perror("strtol");
    + return -1;
    + }
    + else if (ret == 0) {
    + fprintf(stderr, "No digits found in '%s'\n", str);
    + break;
    + }
    + if (loss_list_index > 0 && last_value >= val) {
    + fprintf(stderr, "Values must be in increasing order. %ld
is smaller than %d\n", val, last_value);
    + exit(0);
    + }
    + // Insert range
    + if (range) {
    + while (last_value < val) {
    + if (loss_list_index == MAX LOSS LENGTH) {
    + fprintf(stderr, "Too many fixed losses specified! Max
is %d\n", MAX LOSS LENGTH);
    + exit(0);
    + }
    + last_value = last_value + 1;
    + if (store)
    + store[loss_list_index] = last_value;
    + loss_list_index++;
    + }
    + }
    + else {
    + if (loss_list_index == MAX LOSS LENGTH) {
    + fprintf(stderr, "Too many fixed losses specified! Max
is %d\n", MAX LOSS_LENGTH);
    + exit(0);
    + }
C.2. Netem with fixed loss

179

175

170

168

163

162

161

160

159

158

157

156

155

153

152

150

149

148

146

147

static int netem_parse_opt(struct qdisc_util *qu, int argc, char **argv,    
struct nlmshdr *n)
{
  @-176,6 +266,7 @- static int netem_parse_opt(struct qdisc_util *qu, int argc, char **argv,    
struct tc_netem_corrupt corrupt;    
struct tc_netem_gimodel gimodel;    
struct tc_netem_gemodel gemodel;    
+ struct tc_netem_fixedmodel fixedmodel;    
struct tc_netem_rate rate;    
__u16 *dist_data = NULL;    
__u16 loss_type = NETEM_LOSS_UNSPEC;    
@-321,6 +412,54 @- static int netem_parse_opt(struct qdisc_util *qu, int argc, char **argv,    
   explain1("loss gimodel k");    
return -1;
}
  }
  else if (!strncmp(*argv, "fixed", 5)) {
    int verbose = 0;
    + // Enable verbose
    + if (!strcmp(*argv, "fixedv")) {
      + verbose = 1;
      + }
    +
    + if (!NEXT_ARG_OK()) {
      + explain1("\'loss fixed\' requires that you specify which packets to drop.\n");
      + return -1;
      + }
    + NEXT_ARG();
    +
    + loss_type = NETEM_LOSS_FIXED;
    + fixedmodel.packets_processed = 0;
    + fixedmodel.packets_dropped = 0;
    + fixedmodel.verbose = verbose;

+ last_value = val;
+ if (store)
+   store[loss_list_index] = val;
+ loss_list_index++;
+ }
+ // End of string
+ if (*(endptr == '\0')
+   break;
+ // Range
+ + range = (*endptr == ':' ? 1 : 0;
+ + if (*endptr != ':' && *endptr != ',') {
+   fprintf(stderr, "Illegal value '%c'. Values must be separated with ',', or ':' for range.\n", *endptr);
+   exit(0);
+   }
+   endptr++; // Skip ',' or ',",
+ str = endptr;
+ +}
+ +
+ if (store)
+   store[loss_list_index] = 0;
+ // Return number of specified packets to drop
+ return loss_list_index;
+ })
+ +
+ static int netem_parse_opt(struct qdisc_util *qu, int argc, char **argv,    
struct nlmshdr *n)
int count = parse_fixed_loss(*argv, NULL);
if (count <= 0) {
  explain1("'loss fixed' requires that you specify
which packets to drop.\n");
  return -1;
}

__u32 *loss = malloc((count + 1) * sizeof(__u32*));
count = parse_fixed_loss(*argv, loss);
fixedmodel.loss_length = (__u32) count;
fixedmodel.flow_length = loss[count - 1];

// Copy packets to drop
int i;
for (i = 0; i < count; i++) {
  fixedmodel.loss[i] = loss[i];
  free(loss);
}

if (NEXT_IS_NUMBER()) {
  NEXT_ARG();
  fixedmodel.flow_length = (__u32) atoi(*argv);
  if (fixedmodel.flow_length < loss[count - 1]) { 
    fprintf(stderr, "'loss fixed' flow length ('%d')
" "less than last specified packet ('%d').\n",
      fixedmodel.flow_length, loss[count - 1]);
    return -1;
  }
} else {
  fprintf(stderr, "Unknown loss parameter: %s\n", *argv);
}

if (addattr_l(n, @-478.11 +609.15 @ static int netem_parse_opt(struct qdisc_util *qu, int argc, char **argv,
  if (addattr_l(n, 1024, NETEM_LOSS_GE,
     &gimodel, sizeof(gimodel)) < 0)
   return -1;
} else if (loss_type == NETEM_LOSS_FIXED) {
  if (addattr_l(n, 1024, NETEM_LOSS_FIXED, &fixedmodel,
      sizeof(fixedmodel)) < 0)
    return -1;
  }
} else {
  fprintf(stderr, "loss in the weeds!\n");
  return -1;
}

// Copy packets to drop
addattr_nest_end(n, start);

const struct tc_netem_corrupt *corrupt = NULL;
const struct tc_netem_gimodel *gimodel = NULL;
const struct tc_netem_gemodel *gemodel = NULL;
const struct tc_netem_fixedmodel *fixed_loss_model = NULL;
struct tc_netem_qopt qopt;
const struct tc_netem_rate *rate = NULL;
int len = RTA_PAYLOAD(opt) - sizeof(qopt);
### Code Listing C.3: iproute2 fixed loss patch

```
@@ -536,13 +681,15 @@ static int netem_print_opt(struct qdisc_util *qu, FILE *f, struct rtattr *opt)
  }
  
  if (tb[TCA_NETEM_LOSS]) {
-    struct rtattr *lb[NETEM_LOSS_MAX + 1];
+    struct rtattr *lb[NETEM_LOSS_MAX];
    parse_rtattr_nested(lb, NETEM_LOSS_MAX, tb[TCA_NETEM_LOSS]);
    if (lb[NETEM_LOSS_MAX])
      gemodel = RTA_DATA(lb[NETEM_LOSS_MAX]);
    if (lb[NETEM_LOSS_MAX])
      gemodel = RTA_DATA(lb[NETEM_LOSS_MAX]);
  }
  +  if (lb[NETEM_LOSS_MAX])
  +    fixed_loss_model = RTA_DATA(lb[NETEM_LOSS_MAX]);
  +  }
  if (tb[TCA_NETEM_RATE]) {
    if (RTA_PAYLOAD(tb[TCA_NETEM_RATE]) < sizeof(*rate))
      return -1;
  }
@@ -584,6 +731,17 @@ static int netem_print_opt(struct qdisc_util *qu, FILE *f, struct rtattr *opt)
    fprintf(f, " 1-k %s", sprint_percent(gemodel->k1, b1));
  }
+  if (fixed_loss_model) {
+    fprintf(f, " loss fixedmodel (flow length %d)\n",
+      fixed_loss_model->flow_length);
+    fprintf(f, " packets to lose (%d):", fixed_loss_model->
+      loss_length);
+    int i = 0;
+    while (i < fixed_loss_model->loss_length) {
+      fprintf(f, " %u", fixed_loss_model->loss[i]);
+      i++;
+    }
+  }
  if (qopt.duplicate) {
    fprintf(f, " duplicate %s",
      sprint_percent(qopt.duplicate, b1));
```
Appendix D

Comments from Ilpo Järvinen on Linux mailing list

Quote D.1

“Also, this approach is extremely intrusive and adding non-linear seqno things into write queue will require _you_ to do _full audit_ over every single place to verify that seqno leaps backwards won’t break anything (and you’ll still probably miss some cases). I wonder if you realize how easily this kind of change manifests itself as a silent data corruption on stream level and have taken appropriate actions to validate that not a single one of scenario leads to data coming as different out as was sent in (every single byte, it’s not enough to declare that application worked which could well happen with corrupted data too). TCP is very coreish and such bugs will definitely hit people hard.”

On the approach of modifying the SKBs in the TCP output queue. (Järvinen [2009, b12])

Quote D.2

“It seems very intrusive solution in general. I doubt you succeed in pulling it off as is without breaking something. To me it seems rather fragile approach to make write queue seqno backleaps you’re proposing. It also leads to troubles in the true size as you have noticed. Why not just building those redundancy containing segments at the write time in case the stream is thin, then all other parts would not have to bother about
dealing these things? Number of sysctls should be minimized, if they’re to be added at all. Skb work functions should be separated from tcp layer things.

If you depend on non-changing sysctl value to select right branch, you’re asking for trouble as the userspace is allowed to change it during the flow as well and even during the ack processing.”

Remarks at the end of email. (Järvinen [2009, b12])
# Appendix E

## Errata

<table>
<thead>
<tr>
<th>Erratum</th>
<th>Type</th>
<th>Action</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(1)</td>
<td>Added</td>
<td>Prof.</td>
<td>v (L2)</td>
</tr>
<tr>
<td>Err(2)</td>
<td>Added</td>
<td>Prof.</td>
<td>v (L3)</td>
</tr>
<tr>
<td>Err(3)</td>
<td>Added</td>
<td></td>
<td>xxiii (L8)</td>
</tr>
<tr>
<td>Err(4)</td>
<td>Replaced</td>
<td>th → the</td>
<td>xxiii (L8)</td>
</tr>
<tr>
<td>Err(5)</td>
<td>Added</td>
<td></td>
<td>xxiii (L9)</td>
</tr>
<tr>
<td>Err(6)</td>
<td>Replaced</td>
<td>described → describes</td>
<td>xxiv (L15)</td>
</tr>
<tr>
<td>Err(7)</td>
<td>Replaced</td>
<td>that behaves → should behave</td>
<td>xxvii (L33)</td>
</tr>
<tr>
<td>Err(8)</td>
<td>Replaced</td>
<td>greedy stream → greedy streams</td>
<td>xxviii (L25)</td>
</tr>
</tbody>
</table>

### 2.1

<table>
<thead>
<tr>
<th>Err(9)</th>
<th>Note</th>
<th>Merged paragraph</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>7 (L27)</td>
</tr>
</tbody>
</table>

### 2.3.2

<table>
<thead>
<tr>
<th>Err(10)</th>
<th>Added</th>
<th></th>
<th>13 (L13)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(11)</td>
<td>Replaced</td>
<td>thin stream → thin streams</td>
<td>13 (L15)</td>
</tr>
</tbody>
</table>

### 2.4.5.2

| Err(12) | Deleted | the | 24 (L37) |

### 2.4.7

| Err(13) | Replaced | results → result | 27 (L18) |

### 2.5.3

<table>
<thead>
<tr>
<th>Err(14)</th>
<th>Deleted</th>
<th>the</th>
<th>29 (L24)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(15)</td>
<td>Deleted</td>
<td>competes</td>
<td>29 (L26)</td>
</tr>
<tr>
<td>Err(16)</td>
<td>Added</td>
<td></td>
<td>29 (L36)</td>
</tr>
</tbody>
</table>

### 3

| Err(17) | Deleted | the | 45 (L8) |

### 3.1

<table>
<thead>
<tr>
<th>Err(18)</th>
<th>Replaced</th>
<th>the → all</th>
<th>45 (L24)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(19)</td>
<td>Replaced</td>
<td>Thin stream → Thin-stream</td>
<td>46 (L7)</td>
</tr>
<tr>
<td>Err(20)</td>
<td>Deleted</td>
<td>pattern</td>
<td>47 (L6)</td>
</tr>
<tr>
<td>Err(21)</td>
<td>Replaced</td>
<td>better → lower</td>
<td>51 (L2)</td>
</tr>
</tbody>
</table>

### 3.1.1

| Err(22) | Replaced | remove hyphen → re-transmission | 51 (L20) |

### 3.2.1

<table>
<thead>
<tr>
<th>Err(23)</th>
<th>Replaced</th>
<th>typo → At</th>
<th>51 (L33)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(24)</td>
<td>Deleted</td>
<td>a</td>
<td>51 (L39)</td>
</tr>
</tbody>
</table>
## Appendix E. Errata

<table>
<thead>
<tr>
<th>Err(25)</th>
<th>Added</th>
<th>the data in frame in</th>
<th>52 (L3)</th>
</tr>
</thead>
</table>

### 3.2.1.1 Err(26)  
| Replaced | packet → packets | 52 (L15) |

| 3.3 | Err(27) | Deleted | , | 55 (L19) |
|      | Err(28) | Added   | the | 55 (L20) |

### 3.4 Err(29)  
| Replaced | thin streams → thin stream | 56 (L8) |
| Err(30) | Replaced | cause → causes | 56 (L9) |
| Err(31) | Deleted | the | 56 (L15) |
| Err(32) | Replaced | slow-start → slow-start | 56 (L21) |
| Err(33) | Replaced | it → the | 56 (L42) |
| Err(34) | Added | a | 56 (L42) |

### 3.5 Err(35)  
| Added | , | 57 (L35) |

### 4 Err(36)  
| Replaced | thin stream → thin streams | 59 (L16) |
| Err(37) | Replaced | Next, → Finally, we | 59 (L18) |

### 4.1 Err(38)  
| Added | , | 59 (L32) |
| Err(39) | Added | , | 59 (L33) |

### 4.2 Err(40)  
| Replaced | incomming → incoming | 62 (L37) |
| Err(41) | Replaced | incomming → incoming | 62 (L39) |
| Err(42) | Replaced | occured → occurred | 63 (L13) |
| Err(43) | Replaced | ambiguous → ambiguous | 63 (L18) |
| Err(44) | Replaced | unambigously → unambiguously | 64 (L1) |

### 4.3.2 Err(45)  
| Replaced | We show shortened versions of the most relevant functions for performing the bundling, and explain their main tasks: Following comes a short explanation of the mains tasks of the most relevant functions for performing the bundling. | 67 (L4) |

| Err(46) | Replaced | wether → whether | 67 (L18) |
| Err(47) | Replaced | occured → occurred | 68 (L8) |
| Err(48) | Deleted | , | 68 (L17) |
| Err(49) | Replaced | depicted → depicted | 68 (L19) |

### 4.4.1 Err(50)  
| Added | , | 69 (L12) |
| Err(51) | Replaced | (RFC5348) → RFC5348 | 69 (L12) |
| Err(52) | Added | , | 69 (L12) |
| Err(53) | Added | on | 69 (L21) |
| Err(54) | Added | treat | 69 (L28) |
| Err(55) | Added | , | 69 (L29) |

### 4.4.1.2 Err(56)  
| Deleted | the | 70 (L20) |
4.4.1.3 Err(57) Deleted the 72 (L4)

4.4.1.4 Err(58) Replaced with the same bit rate as standard TCP sending fewer MSS sized packets, → , competing with a standard TCP stream (sending with the same bit rate, but with fewer MSS sized packets), 72 (L26)

Err(59) Replaced TCP → the TCP stream 72 (L28)
Err(60) Added , 72 (L31)
Err(61) Added only 72 (L31)
Err(62) Added , 72 (L33)
Err(63) Replaced per packet → per-packet 73 (L7)
Err(64) Replaced is → can be 73 (L8)

4.4.1.5 Err(65) Added TFRC 73 (L29)

4.5 Err(66) Replaced sub-directory → subdirectory 76 (L32)
Err(67) Replaced TFRC specific → TFRC-specific 76 (L33)

4.5.3.1 Err(68) Replaced ACK analysis → ACK-analysis 81 (L6)
Err(69) Replaced can → must 81 (L7)
Err(70) Deleted ) 81 (L9)
Err(71) Replaced incomming → incoming 81 (L28)

4.5.3.2 Err(72) Replaced otherwhise → otherwise 82 (L5)
Err(73) Deleted a detects a loss, 82 (L11)

4.6 Err(74) Added the 82 (L35)
Err(75) Replaced for → on 82 (L35)
Err(76) Replaced differantiate → differentiate 83 (L11)
Err(77) Added , 83 (L14)
Err(78) Replaced utilise → utilize 83 (L17)
Err(79) Replaced paradigm → paradigm 83 (L17)

5 Err(80) Replaced described → describe 85 (L4)

5.1.1.1 Err(81) Added , 85 (L26)

5.1.1.3 Err(82) Replaced changes → change 86 (L41)

5.1.2 Err(83) Replaced The TCP-friendliness is how well the RDB modifications play with current TCP functionality. → TCP-friendliness, in this context, is how the RDB modifications affect the behaviour the streams, in regards to fairness towards competing streams using standard TCP. 87 (L6)
<table>
<thead>
<tr>
<th>Err(84)</th>
<th>Replaced</th>
<th>TCP friendly → TCP-friendly</th>
<th>87 (L9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(85)</td>
<td>Note</td>
<td>Merged paragraph</td>
<td>87 (L13)</td>
</tr>
<tr>
<td>Err(86)</td>
<td>Added</td>
<td>also</td>
<td>87 (L13)</td>
</tr>
<tr>
<td>Err(87)</td>
<td>Replaced</td>
<td>TCP friendly → TCP-friendly</td>
<td>87 (L14)</td>
</tr>
<tr>
<td>Err(88)</td>
<td>Added</td>
<td>in regards to throughput</td>
<td>87 (L14)</td>
</tr>
<tr>
<td>Err(89)</td>
<td>Deleted</td>
<td>,</td>
<td>87 (L15)</td>
</tr>
</tbody>
</table>

5.1.3.1 | Err(90) | Deleted | it | 88 (L33) |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(91)</td>
<td>Replaced</td>
<td>in → of</td>
<td>88 (L33)</td>
<td></td>
</tr>
<tr>
<td>Err(92)</td>
<td>Added</td>
<td>the</td>
<td>88 (L34)</td>
<td></td>
</tr>
<tr>
<td>Err(93)</td>
<td>Added</td>
<td>,</td>
<td>88 (L34)</td>
<td></td>
</tr>
<tr>
<td>Err(94)</td>
<td>Note</td>
<td>Merged paragraph</td>
<td>89 (L4)</td>
<td></td>
</tr>
</tbody>
</table>

5.1.3.2 | Err(95) | Note | Merged paragraph | 89 (L19) |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(96)</td>
<td>Added</td>
<td>code listing 2.3</td>
<td>90 (L14)</td>
<td></td>
</tr>
<tr>
<td>Err(97)</td>
<td>Added</td>
<td>code listing 2.3</td>
<td>90 (L16)</td>
<td></td>
</tr>
<tr>
<td>Err(98)</td>
<td>Replaced</td>
<td>As → Is is made even more complex due to the fact that</td>
<td>90 (L30)</td>
<td></td>
</tr>
</tbody>
</table>

5.2 | Err(99) | Deleted | for | 91 (L6) |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(100)</td>
<td>Added</td>
<td>, which</td>
<td>91 (L7)</td>
<td></td>
</tr>
</tbody>
</table>

5.2.2 | Err(101) | Added | , | 92 (L3) |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(102)</td>
<td>Added</td>
<td>,</td>
<td>92 (L4)</td>
<td></td>
</tr>
<tr>
<td>Err(103)</td>
<td>Replaced</td>
<td>effect → affect</td>
<td>92 (L5)</td>
<td></td>
</tr>
</tbody>
</table>

5.2.3.2 | Err(104) | Added | (experiments 2 and 3) | 93 (L31) |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(105)</td>
<td>Replaced</td>
<td>5+20 * 2 + 5→ 16 * 2</td>
<td>93 (L32)</td>
<td></td>
</tr>
<tr>
<td>Err(106)</td>
<td>Added</td>
<td>(experiment 4)</td>
<td>93 (L33)</td>
<td></td>
</tr>
<tr>
<td>Err(107)</td>
<td>Replaced</td>
<td>20<em>3 → 13</em>3</td>
<td>93 (L33)</td>
<td></td>
</tr>
<tr>
<td>Err(108)</td>
<td>Deleted</td>
<td>of</td>
<td>93 (L35)</td>
<td></td>
</tr>
</tbody>
</table>

5.2.4 | Err(109) | Added | in experiment 1 | 94 (L33) |

5.2.4.1 | Err(110) | Replaced | for → on | 95 (L6) |

5.3.2 | Err(111) | Replaced | of → are | 95 (L21) |

5.3.3.2 | Err(112) | Replaced | or → of | 99 (L7) |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(113)</td>
<td>Replaced</td>
<td>packet → payload</td>
<td>99 (L9)</td>
<td></td>
</tr>
</tbody>
</table>

5.3.4 | Err(114) | Replaced | that we have used → , | 99 (L24) |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(115)</td>
<td>Added</td>
<td>,</td>
<td>99 (L25)</td>
<td></td>
</tr>
<tr>
<td>Err(116)</td>
<td>Replaced</td>
<td>Important → The following are the most important</td>
<td>99 (L29)</td>
<td></td>
</tr>
<tr>
<td>Err(117)</td>
<td>Added</td>
<td>the</td>
<td>99 (L35)</td>
<td></td>
</tr>
<tr>
<td>Err(118)</td>
<td>Replaced</td>
<td>retransmission → retransmissions</td>
<td>99 (L35)</td>
<td></td>
</tr>
<tr>
<td>Err(119)</td>
<td>Added</td>
<td>the</td>
<td>99 (L35)</td>
<td></td>
</tr>
<tr>
<td>Err(120)</td>
<td>Replaced</td>
<td>are → is</td>
<td>99 (L37)</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>Err(121)</td>
<td>Replaced</td>
<td>some → many</td>
<td>99 (L37)</td>
<td></td>
</tr>
<tr>
<td>Err(122)</td>
<td>Replaced</td>
<td>others → some scenarios, however</td>
<td>99 (L37)</td>
<td></td>
</tr>
<tr>
<td>Err(123)</td>
<td>Replaced</td>
<td>retransmission → retransmissions</td>
<td>99 (L39)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5.3.4.1</th>
<th>Err(124)</th>
<th>Replaced</th>
<th>has → have</th>
<th>100 (L8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(125)</td>
<td>Note</td>
<td>fixed incorrect newline</td>
<td>100 (L15)</td>
<td></td>
</tr>
<tr>
<td>Err(126)</td>
<td>Replaced</td>
<td>could → can</td>
<td>100 (L18)</td>
<td></td>
</tr>
<tr>
<td>Err(127)</td>
<td>Replaced</td>
<td>We list here some → Following is a list</td>
<td>100 (L19)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5.3.5</th>
<th>Err(128)</th>
<th>Note</th>
<th>Created paragraph</th>
<th>103 (L5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(129)</td>
<td>Replaced</td>
<td>enables → enable</td>
<td>103 (L6)</td>
<td></td>
</tr>
</tbody>
</table>

| 5.3.6 | Err(130) | Replaced | While this useful for experiments, it was not as useful while developing RDBv2. To easily reproduce a reliably and → During the development of RDBv2, to reliably reproduce a 104 (L3) |

| 5.4 | Err(131) | Replaced | experiments → experiment | 105 (L11) |

<table>
<thead>
<tr>
<th>5.4.1.2</th>
<th>Err(132)</th>
<th>Replaced</th>
<th>measured on → calculated from</th>
<th>106 (L18)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(133)</td>
<td>Replaced</td>
<td>included → includes</td>
<td>107 (L11)</td>
<td></td>
</tr>
</tbody>
</table>

| 5.4.1.3 | Err(134) | Replaced | a long → along | 108 (L4) |

<table>
<thead>
<tr>
<th>5.4.2</th>
<th>Err(135)</th>
<th>Added</th>
<th>are</th>
<th>109 (L12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(136)</td>
<td>Added</td>
<td>the</td>
<td>109 (L15)</td>
<td></td>
</tr>
<tr>
<td>Err(137)</td>
<td>Added</td>
<td>the</td>
<td>109 (L18)</td>
<td></td>
</tr>
</tbody>
</table>

| 5.4.2.1 | Err(138) | Replaced | is → are | 109 (L23) |

<table>
<thead>
<tr>
<th>5.4.2.2</th>
<th>Err(139)</th>
<th>Replaced</th>
<th>RDB 8 → RDB8</th>
<th>110 (L6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(140)</td>
<td>Replaced</td>
<td>RDB 4 → RDB4</td>
<td>110 (L8)</td>
<td></td>
</tr>
<tr>
<td>Err(141)</td>
<td>Replaced</td>
<td>4 → 3</td>
<td>110 (L9)</td>
<td></td>
</tr>
<tr>
<td>Err(142)</td>
<td>Replaced</td>
<td>RDB 4 → RDB4</td>
<td>110 (L11)</td>
<td></td>
</tr>
<tr>
<td>Err(143)</td>
<td>Replaced</td>
<td>RDB 8 → RDB8</td>
<td>110 (L11)</td>
<td></td>
</tr>
<tr>
<td>Err(144)</td>
<td>Replaced</td>
<td>RDB 4 → RDB4</td>
<td>110 (L12)</td>
<td></td>
</tr>
<tr>
<td>Err(145)</td>
<td>Note</td>
<td>Merged paragraph</td>
<td>110 (L12)</td>
<td></td>
</tr>
<tr>
<td>Err(146)</td>
<td>Replaced</td>
<td>observed → observe</td>
<td>110 (L13)</td>
<td></td>
</tr>
<tr>
<td>Err(147)</td>
<td>Replaced</td>
<td>RDB 4 → RDB4</td>
<td>110 (L15)</td>
<td></td>
</tr>
<tr>
<td>Err(148)</td>
<td>Replaced</td>
<td>RDB 8 → RDB8</td>
<td>110 (L16)</td>
<td></td>
</tr>
<tr>
<td>Err(149)</td>
<td>Added</td>
<td>due to</td>
<td>110 (L18)</td>
<td></td>
</tr>
<tr>
<td>Err(150)</td>
<td>Added</td>
<td>who</td>
<td>110 (L19)</td>
<td></td>
</tr>
<tr>
<td>Err(151)</td>
<td>Replaced</td>
<td>packet are → packets is</td>
<td>110 (L20)</td>
<td></td>
</tr>
<tr>
<td>Err(152)</td>
<td>Replaced</td>
<td>RDB 8 → RDB8</td>
<td>111 (L1)</td>
<td></td>
</tr>
<tr>
<td>Err(153)</td>
<td>Replaced</td>
<td>RDB 4 → RDB4</td>
<td>111 (L2)</td>
<td></td>
</tr>
<tr>
<td>Err(154)</td>
<td>Replaced</td>
<td>RDB 8 → RDB8</td>
<td>111 (L3)</td>
<td></td>
</tr>
</tbody>
</table>
### Appendix E. Errata

<table>
<thead>
<tr>
<th>Err(155)</th>
<th>Replaced</th>
<th>RDB 4 → RDB₄</th>
<th>111 (L7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(156)</td>
<td>Replaced</td>
<td>better → lower</td>
<td>111 (L8)</td>
</tr>
<tr>
<td>Err(157)</td>
<td>Replaced</td>
<td>RDB 8 → RDB₈</td>
<td>111 (L9)</td>
</tr>
</tbody>
</table>

#### 5.4.2.3

<table>
<thead>
<tr>
<th>Err(158)</th>
<th>Replaced</th>
<th>are → is</th>
<th>111 (L13)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(159)</td>
<td>Replaced</td>
<td>RDB 8 → RDB₈</td>
<td>111 (L18)</td>
</tr>
<tr>
<td>Err(160)</td>
<td>Note</td>
<td>Merged paragraph</td>
<td>112 (L7)</td>
</tr>
<tr>
<td>Err(161)</td>
<td>Replaced</td>
<td>the → that</td>
<td>112 (L8)</td>
</tr>
<tr>
<td>Err(162)</td>
<td>Deleted</td>
<td>In addition, disabling Nagle’s Algorithm and TCP delayed acknowledgment gives the sender and receiver less room to send data packets and ACKs at their own pace.</td>
<td>112 (L9)</td>
</tr>
</tbody>
</table>

#### 5.4.3

| Err(163) | Deleted | , | 113 (L5) |

#### 5.4.3.1

<table>
<thead>
<tr>
<th>Err(164)</th>
<th>Replaced</th>
<th>is → are</th>
<th>113 (L11)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(165)</td>
<td>Replaced</td>
<td>20 → 16</td>
<td>113 (L18)</td>
</tr>
<tr>
<td>Err(166)</td>
<td>Replaced</td>
<td>10:1, 30:3, 75:7, and 100:10 ms → 10:1, 30:3, 50:5, and 75:7 ms</td>
<td>113 (L20)</td>
</tr>
<tr>
<td>Err(167)</td>
<td>Replaced</td>
<td>20 → 16</td>
<td>113 (L29)</td>
</tr>
<tr>
<td>Err(168)</td>
<td>Replaced</td>
<td>10:1, 30:3, 75:7, and 100:10 ms → 10:1, 30:3, 50:5, and 75:7 ms</td>
<td>113 (L31)</td>
</tr>
</tbody>
</table>

#### 5.4.3.2

<table>
<thead>
<tr>
<th>Err(169)</th>
<th>Deleted</th>
<th>,</th>
<th>114 (L14)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(170)</td>
<td>Note</td>
<td>Merged paragraph</td>
<td>114 (L15)</td>
</tr>
<tr>
<td>Err(171)</td>
<td>Replaced</td>
<td>better → lower</td>
<td>114 (L28)</td>
</tr>
<tr>
<td>Err(172)</td>
<td>Added</td>
<td>(subfigure 5.11.(b))</td>
<td>114 (L33)</td>
</tr>
<tr>
<td>Err(173)</td>
<td>Replaced</td>
<td>T-5-10:1-RDB → E2-5-10:1-RDB</td>
<td>114 (L36)</td>
</tr>
<tr>
<td>Err(174)</td>
<td>Replaced</td>
<td>bigger → greater</td>
<td>114 (L39)</td>
</tr>
<tr>
<td>Err(175)</td>
<td>Replaced</td>
<td>reducing → increasing</td>
<td>115 (L1)</td>
</tr>
<tr>
<td>Err(176)</td>
<td>Note</td>
<td>Merged paragraph</td>
<td>115 (L3)</td>
</tr>
<tr>
<td>Err(177)</td>
<td>Replaced</td>
<td>better → lower</td>
<td>115 (L4)</td>
</tr>
<tr>
<td>Err(178)</td>
<td>Deleted</td>
<td>in</td>
<td>117 (L18)</td>
</tr>
<tr>
<td>Err(179)</td>
<td>Replaced</td>
<td>thin stream → thin-stream</td>
<td>118 (L7)</td>
</tr>
<tr>
<td>Err(180)</td>
<td>Replaced</td>
<td>better → lower</td>
<td>118 (L11)</td>
</tr>
<tr>
<td>Err(181)</td>
<td>Replaced</td>
<td>better → lower</td>
<td>118 (L16)</td>
</tr>
</tbody>
</table>

#### 5.4.3.3

<table>
<thead>
<tr>
<th>Err(182)</th>
<th>Replaced</th>
<th>achieves better → achieve lower</th>
<th>120 (L3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(183)</td>
<td>Replaced</td>
<td>better → lower</td>
<td>120 (L15)</td>
</tr>
<tr>
<td>Err(184)</td>
<td>Deleted</td>
<td>in</td>
<td>120 (L19)</td>
</tr>
</tbody>
</table>

#### 5.4.4

<table>
<thead>
<tr>
<th>Err(185)</th>
<th>Replaced</th>
<th>has the exact → uses the same</th>
<th>121 (L3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(186)</td>
<td>Replaced</td>
<td>included → include</td>
<td>121 (L6)</td>
</tr>
<tr>
<td>Err(187)</td>
<td>Added</td>
<td>In this experiment, they are limited to bundle one previous segment.</td>
<td>121 (L6)</td>
</tr>
<tr>
<td>5.4.4.1</td>
<td>Err(188)</td>
<td>Note</td>
<td>Added list of correction to test parameters.</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>5.4.4.2</td>
<td>Err(189)</td>
<td>Deleted</td>
<td>,</td>
</tr>
<tr>
<td>Err(190)</td>
<td>Deleted</td>
<td>,</td>
<td></td>
</tr>
<tr>
<td>Err(191)</td>
<td>Replaced</td>
<td>ITT → ITTs</td>
<td></td>
</tr>
<tr>
<td>Err(192)</td>
<td>Replaced</td>
<td>19 → 19 ms</td>
<td></td>
</tr>
<tr>
<td>Err(193)</td>
<td>Replaced</td>
<td>20 → 20 ms</td>
<td></td>
</tr>
<tr>
<td>Err(194)</td>
<td>Replaced</td>
<td>thin stream → thin streams</td>
<td></td>
</tr>
<tr>
<td>Err(195)</td>
<td>Replaced</td>
<td>21 → 21 ms</td>
<td></td>
</tr>
<tr>
<td>Err(196)</td>
<td>Replaced</td>
<td>23 → 23 ms</td>
<td></td>
</tr>
<tr>
<td>Err(197)</td>
<td>Replaced</td>
<td>thin stream → thin streams</td>
<td></td>
</tr>
<tr>
<td>Err(198)</td>
<td>Replaced</td>
<td>thin stream → thin streams</td>
<td></td>
</tr>
<tr>
<td>Err(199)</td>
<td>Replaced</td>
<td>thin stream → thin streams</td>
<td></td>
</tr>
<tr>
<td>Err(200)</td>
<td>Added</td>
<td>is</td>
<td></td>
</tr>
<tr>
<td>Err(201)</td>
<td>Deleted</td>
<td>the</td>
<td></td>
</tr>
<tr>
<td>Err(202)</td>
<td>Added</td>
<td>in</td>
<td></td>
</tr>
<tr>
<td>Err(203)</td>
<td>Replaced</td>
<td>thin stream → thin streams</td>
<td></td>
</tr>
<tr>
<td>Err(204)</td>
<td>Replaced</td>
<td>achieve → have</td>
<td></td>
</tr>
<tr>
<td>Err(205)</td>
<td>Added</td>
<td>ACK latencies</td>
<td></td>
</tr>
<tr>
<td>Err(206)</td>
<td>Replaced</td>
<td>where with TCP average is 246 ms and RDB average is 229 ms → where the average is 246 ms for the TCP streams, and 229 ms for the RDB streams</td>
<td></td>
</tr>
<tr>
<td>Err(207)</td>
<td>Replaced</td>
<td>thin stream → thin streams</td>
<td></td>
</tr>
<tr>
<td>Err(208)</td>
<td>Replaced</td>
<td>achieves → achieve</td>
<td></td>
</tr>
<tr>
<td>Err(209)</td>
<td>Deleted</td>
<td>,</td>
<td></td>
</tr>
</tbody>
</table>

| 5.4.4.3 | Err(210) | Replaced | shows → show | 125 (L2) |

| 5.4.5 | Err(211) | Replaced | great → high | 127 (L10) |
| Err(212) | Added | of | 127 (L10) |

| 5.4.5.1 | Err(213) | Replaced | is → are | 127 (L15) |

| 5.4.5.2 | Err(214) | Added | streams | 128 (L10) |
| Err(215) | Replaced | payloads → payload size | 128 (L11) |
| Err(216) | Deleted | the | 128 (L11) |
| Err(217) | Added | , | 128 (L18) |
| Err(218) | Replaced | is → are | 129 (L3) |
| Err(219) | Replaced | of → as | 129 (L5) |

| 5.4.5.3 | Err(220) | Replaced | these experiments → this experiment | 129 (L8) |
| Err(221) | Note | Merged paragraph | 129 (L10) |

<p>| 5.5 | Err(222) | Deleted | , | 131 (L2) |
| Err(223) | Note | Merged paragraph | 131 (L3) |</p>
<table>
<thead>
<tr>
<th>Err</th>
<th>Action</th>
<th>Change Description</th>
<th>Line(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err(224)</td>
<td>Replaced</td>
<td>significant less delay → significantly lower per-packet latency</td>
<td>131 (L4)</td>
</tr>
<tr>
<td>Err(225)</td>
<td>Replaced</td>
<td>we also tested → it is necessary to test</td>
<td>131 (L10)</td>
</tr>
<tr>
<td>Err(226)</td>
<td>Note</td>
<td>Merged paragraph</td>
<td>131 (L11)</td>
</tr>
<tr>
<td>Err(227)</td>
<td>Replaced</td>
<td>While → Whereas</td>
<td>131 (L11)</td>
</tr>
<tr>
<td>6.1</td>
<td>Err(228)</td>
<td>Replaced It’s → Its</td>
<td>133 (L6)</td>
</tr>
<tr>
<td>Err(229)</td>
<td>Note</td>
<td>Merged paragraph</td>
<td>133 (L20)</td>
</tr>
<tr>
<td>6.2</td>
<td>Err(230)</td>
<td>Replaced the → an RDB-stream</td>
<td>134 (L7)</td>
</tr>
<tr>
<td>6.3</td>
<td>Err(231)</td>
<td>Deleted Work specifically on RDB.</td>
<td>135 (L4)</td>
</tr>
<tr>
<td></td>
<td>Added</td>
<td>)</td>
<td>135 (L7)</td>
</tr>
<tr>
<td>Err(233)</td>
<td>Replaced</td>
<td>were → was</td>
<td>135 (L16)</td>
</tr>
<tr>
<td>Err(234)</td>
<td>Replaced</td>
<td>mechanisms and takes → different mechanisms, by taking</td>
<td>135 (L32)</td>
</tr>
<tr>
<td></td>
<td>Deleted</td>
<td>the</td>
<td>135 (L34)</td>
</tr>
</tbody>
</table>