Lightweight and Flexible Single-Path Congestion Control Coupling

PhD Thesis

Safiqul Islam

March 9, 2017
To Mom and Monita
Abstract

Communication between two Internet hosts using parallel connections may result in unwanted interference between the connections. In this dissertation, we propose a sender-side solution to address this problem by letting the congestion controllers of the different connections collaborate, correctly taking congestion control logic into account. In addition, this dissertation proposes a simple light-weight encapsulation mechanism that multiplexes the connections onto a single UDP-connection to ensure the same bottleneck. Real-life experiments and simulations show that our solution works for a wide variety of congestion control mechanisms, provides great flexibility when allocating application traffic to the connections, and results in lower queuing delay and less packet loss.
Acknowledgments

I would like to express my heartiest gratitude to my Ph.D. supervisors Professor Michael Welzl and Professor Stein Gjessing for helping me with critical reviews, providing detailed background knowledge, and technical advice. Working with Michael and Stein has taught me how to carry out research work and instilled courage in me to become a confident public speaker. Two “best paper” awards in one Ph.D. journey came out just because of them. They even helped me in non-technical matters; especially Stein, for teaching me how to tie my shoelace on a train in Tokyo, Japan. It was quite embarrassing, but I learned it now. I especially want to thank Michael for his continuous encouragement and inspiration that helped me to look beyond the boundaries. I hope to be as energetic and enthusiastic as Michael so that I can also command the audience as he does. This is not balderdash, I mean it.

I would want to thank David Hayes for his support and advice, especially for allowing me to disturb him for every silly question. I would also like to acknowledge Professor Grenville Armitage from Swinburne University of Technology, and Professor Joe Touch from University of Southern California for their valuable suggestions regarding TCP congestion control coupling.

I would like to thank my colleagues who have made my Ph.D. journey memorable, especially Abhishek Singh, Amir Taherkodi, Jonas Modahl Mørkás, Katja Elisabeth Andersson, Kashif Sana Dar, Kristian Hioirth, Lucas Provensi, Naeem Khademi, Narasimha Raghavan, Navneet Kumar Pandey, Runa Barik, Vinay Setty and Øystein Dale. I would like to extend my appreciation to Eskil Brun from the administration who helped me recover my data when I badly needed them. I would also like to thank Razib Hayat Khan and Rezaul Hoque for cheering me up whenever I felt down.

I am grateful for the financial support during my research from the Department of Informatics, University of Oslo, Huawei Technologies Co., Ltd., the European Unions Horizon 2020 research and innovation programme (under grant agreement No. 644334 (NEAT)), and the European Union through the FP7-ICT project RITE (under contract number 317700). In addition, I would like to thank the IETF RMCAT and IRTF ICCRG Working Groups, and all the partners from RITE and
NEAT for their fruitful and constructive discussions throughout my research work. I want to express my gratitude to my parents and my brother Khursheed Alam for an immense amount of support throughout my life. Whatever I am, I owe especially to you, Mom! Last but not least, I would like to thank my wife Zannatul Naim Monita for bearing with me throughout the Ph.D. work. I know thanking you would not be enough, but, seriously, thank you.

Safiqul Islam
Oslo, March 9, 2017
# Contents

Abstract iii

Acknowledgments v

I Overview 1

1 Introduction 3

1.1 Motivation ........................................... 3

1.2 Research Questions ................................. 6

1.3 Research Objectives ................................. 7

1.4 Research Methodology ............................... 7

1.4.1 Network Simulation ............................... 8

1.4.2 Real-life Experiments ............................ 8

1.5 Published Works ..................................... 8

1.5.1 Research Papers in this Thesis .................... 9

1.5.2 Related Internet Drafts ........................... 11

1.5.3 Short Summaries / Posters (Not Included in the Thesis) ... 13

2 Background 15

2.1 Internet Congestion Control ......................... 15

2.2 Coupled Congestion Control ........................ 17

2.2.1 Single-Path Congestion Control Coupling .......... 17

2.2.2 Multiplexing ..................................... 19

2.2.3 Datacenter Capacity Management ................ 19

2.2.4 Multi-Path Congestion Control Coupling .......... 19

2.3 Shared Bottleneck Detection ........................ 20

3 Contributions 21

3.1 Overview of the Main Contributions ................ 22

3.2 ctrlTCP in Datacenters ............................. 28
4 Conclusions

4.1 Addressing Research Questions .................................................. 33
4.2 Future Directions ................................................................. 35
  4.2.1 Further Research ............................................................. 35
  4.2.2 Deployment ................................................................. 36

References .................................................................................. 45

II Research Papers

5 Coupled Congestion Control for RTP Media .................................... 49
  5.1 Introduction .............................................................................. 49
  5.2 Related Work ......................................................................... 50
  5.3 The Flow State Exchange ....................................................... 51
  5.4 Evaluation ............................................................................... 56
  5.5 Conclusions ............................................................................ 64
  5.6 Acknowledgments .................................................................... 65

References .................................................................................. 65

6 Managing Real-Time Media Flows through a Flow State Exchange .... 67
  6.1 Introduction .............................................................................. 68
  6.2 Background and related work .................................................. 69
  6.3 The Flow State Exchange (FSE) ............................................. 70
    6.3.1 Active vs. passive updates ............................................... 70
    6.3.2 Increasing and decreasing rates ........................................ 73
    6.3.3 Application to NADA ....................................................... 74
    Building a stable queue .......................................................... 74
    Accelerated ramp up ............................................................... 74
    Rate update frequency ........................................................... 74
    6.3.4 Application to GCC ......................................................... 75
    Builds a stable queue ............................................................... 75
    6.3.5 A simple passive FSE algorithm for NADA and GCC ....... 76
  6.4 Evaluation ............................................................................... 77
    6.4.1 Prioritization results ....................................................... 78
    6.4.2 Test case results ............................................................. 78
    6.4.3 Delayed feedback tests ................................................... 80
  6.5 Conclusions ............................................................................ 83
  6.6 Acknowledgments .................................................................... 83

References .................................................................................. 83
III  Internet Drafts  

10 Coupled Congestion Control for RTP Media  

11 TCP-CCC: Single-Path TCP Congestion Control Coupling  

12 TCP Control Block Interdependence
List of Figures

1.1 Average queue length (in packets) as the number of RAP flows is varied .......................................................... 4
1.2 Sending rates of two vic processes across a 1 Mbit/s, 50 ms link, with and without coupling ................................... 5

3.1 Fairness between two VMs, with 1 flow in VM1 and 1 to 4 flows in VM2. Without coupling, fairness deteriorates; our ctrlTCP algorithm achieves perfect fairness. ............................................ 28
3.2 Abstract ctrlTCP architecture, showing an example where the hypervisor removes selfish VM behavior. ................. 30

5.1 Average queue length (TFRC) .............................................. 55
5.2 Average queue length (RAP) .................................................. 55
5.3 Loss ratio (TFRC) ............................................................... 55
5.4 Loss ratio (RAP) ............................................................... 55
5.5 Queue growth over time for 3 RAP flows, with FSE .............. 55
5.6 Queue growth over time for 3 RAP flows, without FSE .......... 55
5.7 Average queue length (TFRC) .............................................. 57
5.8 Average queue length (RAP) .................................................. 57
5.9 Loss ratio (TFRC) ............................................................... 58
5.10 Loss ratio (RAP) ............................................................... 58
5.11 Link Utilization (TFRC) ..................................................... 58
5.12 Link Utilization (RAP) ..................................................... 58
5.13 Flow 1 changing its priority coupled via the FSE .................. 59
5.14 Average queue length for 10 RAP flows while changing the queue length from 0.5 BDP (62 Packets) to 1.5 BDP (167 packets) .............................. 60
5.15 Average queue length for 15 RAP flows while changing the queue length from 0.5 BDP (62 Packets) to 1.5 BDP (167 packets) .............................. 60
5.16 Average queue length for 10 TFRC flows while changing the queue length from 0.5 BDP (62 Packets) to 1.5 BDP (167 packets) .............................. 60
5.17 Average queue length for 15 TFRC flows while changing the queue length from 0.5 BDP (62 Packets) to 1.5 BDP (167 packets) .............................. 60
5.18 Loss ratio percentage as the queue length for 10 RAP flows is varied, with and without FSE .......................................................... 61
5.19 Loss ratio percentage as the queue length for 15 RAP flows is varied, with and without FSE .......................................................... 61
5.20 Fairness index as the number of TFRC flows is varied, with and without FSE .......................................................... 61
5.21 Fairness index as the number of RAP flows is varied, with and without FSE .......................................................... 61
5.22 Fairness index for 2 RAP flows as the RTT ratio is varied, with and without FSE .......................................................... 62
5.23 Fairness index for 3 RAP flows as the RTT ratio is varied, with and without FSE .......................................................... 62
5.24 Fairness index for 4 RAP flows as the RTT ratio is varied, with and without FSE .......................................................... 62
5.25 Fairness index for 5 RAP flows as the RTT ratio is varied, with and without FSE .......................................................... 62
5.26 Fairness index for 4 TFRC flows as the RTT ratio is varied, with and without FSE .......................................................... 62
5.27 Fairness index for 5 TFRC flows as the RTT ratio is varied, with and without FSE .......................................................... 62
5.28 Application limited flow and greedy flow – with FSE .......................................................... 63
5.29 Application limited flow and greedy flow – without FSE .......................................................... 63
5.30 High-priority (1) application-limited flow #1 is hardly affected by a low-priority (0.2) greedy flow #2 as long as there is enough capacity for flow #1 .......................................................... 63
5.31 Goodput of two FSE-controlled flows competing with synthetic traffic .......................................................... 64

6.1 System architecture, showing the relationship between the FSE and two sources, S1 and S2 .......................................................... 71
6.2 Active and Passive versions of the FSE. CC_R is the rate received from the flow’s congestion controller. FSE_R(f) is the rate calculated by the FSE. Variables are explained in Section 6.3.5 and Table 6.1 .......................................................... 72
6.3 Jain’s fairness index [20] for two TCP flows with heterogeneous RTTs coupled with the passive FSE. TCP’s fairness deteriorates as the flow RTT ratio decreases due to the lag in adopting the FSE assigned rate. .......................................................... 73
6.4 One-way delay of one NADA flow and one GCC flow across a 1 Mbps, 50 ms base delay link (separate simulations) .......................................................... 74
6.5 Topology used in our experiments .......................................................... 77
6.6 Sending rates of 3 NADA flows and 3 GCC flows as the priorities of flows are varied at around 50 seconds. (Note that markers identify the line and not plotted points) .......................................................... 79
6.7 Sending rates and delays of five NADA flows with one way delays of 10 ms, 25 ms, 50 ms, 100 ms, and 150 ms. The FSE not only enforces perfect fairness but also helps the congestion control mechanism to converge quickly. Delay is largely unaffected. (Note that markers identify the line and not plotted points) 79

6.8 Sending rates and delays of five GCC flows with one way delays of 10 ms, 25 ms, 50 ms, 100 ms, and 150 ms. The FSE not only enforces perfect fairness but also helps the congestion control mechanism to converge quickly. Delay is largely unaffected. (Note that markers identify the line and not plotted points) 80

6.9 Sending rates and delays of two continuous and one intermittent NADA flows, with and without the FSE. (Note that markers identify the line and not plotted points) 81

6.10 Sending rates and delays of two continuous and one intermittent GCC flows, with and without the FSE. (Note that markers identify the line and not plotted points) 81

6.11 Two GCC flows coupled via the FSE, where delay between stream 1 and the FSE is varied based on (a) fixed delay of 50 ms (b) fixed delay of 100 ms, and (c) uniformly distributed random delay, between 1 and 100 ms. 82

7.1 Coupling of 3 connections when connections 2 and 3 join after 5 and 6 seconds 90

7.2 Flow completion time (FCT) of short flows 91

8.1 cwnd of 4 closed TCP flows 97

8.2 cwnd of 4 OpenTCP flows 98

8.3 Average queue length of LEDBAT connections, with and without the OpenTCP algorithm 103

8.4 Link utilization of LEDBAT connections, with and without the OpenTCP algorithm 103

8.5 Closed TCP: LEDBAT gets “pushed aside” 104

8.6 OpenTCP: TCP and LEDBAT can be fair if needed 104

9.1 Message sequence chart of communication between a TCP session and the Coupled Congestion Control (CCC) entity. *Response is not sent if the session is in Fast Recovery (FR) 113

9.2 cwnd of two TCP Reno flows 117

9.3 Coupling of 2 flows when flow 2 joins after 5 seconds. Packet sequence plots and cwnd plots are shown with and without the use of ack-clocking mechanism 118

9.4 Emulated testbed topology 119
9.5 Throughput of 2 flows, for ctrlTCP, E-TCP, and EFCM while increasing the number of timeouts

9.6 cwnd (in Kbytes) plot of two TCP connections using coupled congestion control with priorities compared to a single TCP flow scenario. The aggregate line depicts the sum of cwnds in two connections scenario. cwnd going down all the way to 0 whenever cwnd is reduced is not related to ctrlTCP but a result of how cwnd is internally updated in FreeBSD.

9.7 Flow completion time (FCT) of short flows, with and without ack-clocking (simulation)

9.8 Flow completion time (FCT) of short flows without ack-clocking (emulation)

9.9 Average delay (in milliseconds) as the number of TCP connections is varied, with and without coupled congestion control (emulation).

9.10 Average goodput as the number of TCP connections is varied, with and without coupled congestion control (emulation)

9.11 Loss ratio as the number of TCP connections is varied, with and without coupled congestion control (emulation)

9.12 Throughput ratio as the priorities of two TCP connections are varied

9.13 Fairness between two VMs, with 1 flow in VM1 and 1 to 4 flows in VM2. Without coupling, fairness deteriorates; our ctrlTCP algorithm achieves perfect fairness.

9.14 Complete TCP-in-UDP header

9.15 Complete TCP-in-UDP SYN and SYN/ACK header

9.16 TiU setup TCP option

9.17 TCP-in-UDP Happy Eyeballs
# List of Tables

3.1 Lessons learned in this dissertation .................................. 27

5.1 Names of variables used in algorithms 1 and 2 .................... 53

6.1 Names of variables used in Algorithm 3 ............................. 77

8.1 Overview of work related to sharing congestion information, using two flows as an example ........................................ 99

9.1 Names of variables used in Algorithms 5 to 7 ........................ 116
Part I

Overview
Chapter 1

Introduction

An argument is a connected series of statements intended to establish a proposition . . . Argument is an intellectual process. Contradiction is just the automatic gainsaying of anything the other person says.

Monty Python
The Argument Sketch

1.1 Motivation

When two Internet hosts communicate, several connections may be opened in order to exchange multiple files or streams of data or video. This could for example be the case when a user downloads a web page containing several images and other items from a server, or when a user is involved in interactive real-time communication using WebRTC\(^1\). These parallel connections, each having their own congestion control mechanism, compete over the Internet, resulting in undefined behaviour and unwanted interference between the flows. That is, such competition can induce undesired queuing delay and losses.

Reducing latency and loss is important for applications, especially for multimedia applications. In order to understand the magnitude of the connection competition problem, we ran an ns-2 simulation where we show the increase in queue growth with Rate Adaptation Protocol (RAP) [1]. RAP is a simple rate-based TCP-like congestion control mechanism. Fig. 1.1 illustrates how the average queue length grows as we vary the number of RAP flows.

Generally, there are two approaches to combine the connections that are known to share a common path: by merging application layer data streams onto a single transport layer connection (SPDY [2] and HTTP/2 address this problem by

\(^1\)The IETF counterpart of the W3C WebRTC standard is called RTCWEB. For simplicity, we will use the term “WebRTC” for the whole set of standards.
Introduction

Multiplexing all data on one single TCP connection. Doing this at the application layer leads to transport layer Head-of-Line blocking delay; this has recently been addressed by QUIC [3], which operates over UDP; or by combining the congestion controls of the connections at the transport layer. This thesis emphasizes on the latter, which is often referred to as coupled congestion control.

To highlight the impact on the fairness between flows, we use two instances of the open source video conferencing tool “vic”\(^2\) that we have extended to talk to a simple congestion manager using Unix domain sockets. The vic variant that we use includes TFRC [4] congestion control (implemented and tested by Soo-Hyun Choi for his Ph.D. thesis [5]). Our experiments are carried out with a single physical host, using VirtualBox with Linux for the sender, running two instances of vic for the senders as well as a simple congestion manager, and VirtualBox with Linux for the receiver, running two instances of vic for the receivers. The two VirtualBox instances are logically interconnected on our Mac OS X host system, and the outgoing interface of the sender is set to have a maximum rate of 1 Mbit/s and introduce a propagation delay of 50 ms using tc / netem.

Here, we present some results (see Fig. 1.2) from a test where we have configured the manager to let the two flows share the bandwidth equally, similar to what TFRC would automatically converge to. we have made vic play a file (the common “foreman” test sequence), causing it to adjust the frame rate, which translates

\(^2\)http://mediatools.cs.ucl.ac.uk/nets/mmedia/
Motivation

Figure 1.2: Sending rates of two vic processes across a 1 Mbit/s, 50 ms link, with and without coupling

(a) when TFRC congestion controls are not coupled

(b) when TFRC congestion controls are coupled

Almost a decade ago, there were some efforts to combine the congestion controls of multiple TCP connections sharing the same bottleneck. To the best of our knowledge, Congestion Manager (CM) [6] is the oldest such effort. However, it was hard to implement. Ensemble TCP (E-TCP) [7] and Ensemble Flow Congestion Management (EFCM) [8] are also two proposals along these lines; while E-TCP tried to be no more aggressive than one flow, EFCM tried to be as aggressive as \( n \) flows. Neither E-TCP nor EFCM correctly considered TCP congestion states.

All the aforementioned approaches have an additional, entirely different problem: they assume that multiple TCP connections sending to the same destination would take the same path. This is not always true – load-balancing mechanisms such as Equal-Cost Multi-Path (ECMP) and LAG [9] may force them to take different paths. Therefore, in this particular scenario, combining the congestion controllers...
would incur wrong behavior.

### 1.2 Research Questions

The aforementioned problem statement leads us to formulate the following research questions (RQ) that this dissertation attempts to answer.

- **RQ1:** Can a solution be simple and flexible enough to be gradually deployable without changing the underlying network?

  Due to the sheer design complexity, the Congestion Manager never reached beyond the experimental stack albeit it is a proposed IETF standard [10]. A congestion management solution should be simple and flexible enough so that it can gradually be deployed in the Internet, while supporting a wide range of applications. To address this, this dissertation sets out to design a simple and flexible solution.

- **RQ2:** Can we always apply a solution between the same pair of hosts?

  Due to mechanisms like ECMP and LAG, connections between the same pair of hosts may not share the same bottleneck. This may incur a wrong result, if, e.g., a large Initial Window (IW) from the aggregate is given to a newly joining connection. Are there any obvious cases where a congestion management algorithm can be applied? We set out to explore workable solutions for this issue, thereby answering the question when to apply such a solution.

- **RQ3:** Can we find a simple generic mechanism that can be applied to different congestion control mechanisms?

  A simple sender-side congestion management algorithm cannot readily be applied to different applications with different congestion control mechanisms. For example, a congestion control with receiver-side calculation may not react immediately upon changes at the sender-side only. In this dissertation, we set out to explore how a simple generic mechanism needs to be changed for different congestion control mechanisms.

- **RQ4:** What are the potential benefits of such schemes that have not been considered in the prior approaches?

  As explained in the previous section, the competition between overlapping connections can lead to two performance issues: high latency and increased packet loss. A single congestion management instance can eliminate such competition, and allows hosts to precisely allocate the available capacity to the flows. A congestion management scheme can also improve the latency and reduce packet losses,
Research Objectives

depending on the aggression of the congestion control mechanisms. Comparative studies are required to find the efficacy of a congestion management solution, covering a wide range of tests. We set out to provide such a comparison, evaluating the performance impact on latency and packet loss that prior approaches have not considered.

1.3 Research Objectives

The overarching objective of this dissertation is to find an efficient and feasible (deployable) way to combine the congestion controllers of parallel connections between the same pair of hosts. To achieve this, we address the research questions RQ1 - RQ4 as follows:

1. Make our design simple and flexible while minimizing the changes to the existing mechanisms.

2. Ensure that connections take the same path.

3. Apply our solution to different congestion control mechanisms ranging from bulk transfer to real-time media. This lets us derive general guidelines for applying it to congestion control mechanisms in the future.

4. Reduce overall loss and delay.

1.4 Research Methodology

Analyzing a research problem and evaluating the performance of a solution can be performed in two different ways: (1) formal proof and mathematical modeling, and (2) event-driven simulation and real-life experimentation. This dissertation employs the latter. To ensure the statistical significance, experiments – except for the results of illustrative tests in the form of evolution over time – were repeated many times, and the details of each of the evaluation methodologies are discussed in the published works (see Section 1.5.1 and Chapters 5 to 9).

The dissertation began with an investigation of the current solution space for the congestion control coupling. The previously proposed coupled congestion control solutions were reviewed. The output from this study was then used to formulate a solution to optimize the performance by eliminating the deficiencies of the prior approaches.
1.4.1 Network Simulation

The dissertation verified the solution using two simulators: Network Simulator (ns-2)\(^3\), a popular and widely-used discrete-event simulation tool used by the network researchers which provides substantial support for the simulation of various network and transport protocols, and WebRTC Simulator\(^4\), an open source software package for real-time communication between the browsers which uses Google Congestion Control [11]. The Google implementation in the version of Chromium that was used in this dissertation was 47.0.2494.0. Chapters 5 to 9 used ns-2 and Chapter 6 used the WebRTC simulator for the performance evaluations.

A wide range of transport protocols, ranging from rate-based media congestion controls to less-than-best-effort transport protocols, was used for the performance evaluation. Except for GCC, ns-2 was used for all other protocols. LEDBAT [12] and NADA [13] were imported into ns-2, using the latest code (at the time of writing), provided by the publishers. For the evaluation of our papers in Chapters 7 to 9 the Linux TCP congestion control suite was updated with the code shipped from Linux 3.17.4. A preprocessed version of the TMIX traffic, taken from a 60-minute trace of campus traffic at the University of North Carolina [14], was used as a cross traffic, and it can be accessed from the common TCP evaluation suite [15].

1.4.2 Real-life Experiments

The TCP solution was implemented in the FreeBSD kernel in two different ways: 1) with state shared across the freely available VirtualBox\(^5\) hypervisor, and 2) as a single-OS implementation that couples connections from the same OS only. The Common Open Research Emulator (CORE) [16] version 4.7, a tool for emulating networks on one or more machines, was used to form the topology for the experimentation. To impose cross-traffic, D-ITG [17] was used to generate bursty Internet-like traffic. A more detailed explanation of CORE’s network setup in our real-life experiments is given in Section 9.3 of Chapter 9.

1.5 Published Works

This section describes two important contributions in this dissertation: research papers and Internet drafts. Section 1.5.1 provides the summaries of all the published research papers, while Section 1.5.2 provides the summaries of all the published Internet-Drafts.

\(^{3}\)http://www.isi.edu/nsnam/ns/
\(^{4}\)https://webrtc.org
\(^{5}\)https://www.virtualbox.org
1.5.1 Research Papers in this Thesis

Paper I (Chapter 5)

Title: Coupled Congestion Control for RTP Media

Authors: Safiqul Islam, Michael Welzl, Stein Gjessing and Naeem Khademi


Achievement: Best paper and presentation awards.

My Contributions: Main authorship; idea of the paper; developed all ns-2 simulation scripts and codes, and conducted all simulation tests; data analysis and producing results (e.g., graphs, figures, etc.); contributed text and editorial work.

Summary: This paper introduces a coupled congestion control mechanism for real-time media flows that share a bottleneck. Simulations with two congestion control mechanisms, Rate Adaption Protocol (RAP) and TCP Friendly Rate Control (TFRC), show its benefits in terms of precise rate allocation, reduced overall delay and losses.

Note: An extended technical report is included in this dissertation.

Paper II (Chapter 6)

Title: Managing Real-Time Media Flows through a Flow State Exchange

Authors: Safiqul Islam, Michael Welzl, David Hayes and Stein Gjessing


My Contributions: Main authorship; idea of the paper; developed all ns-2 and Chromium simulation scripts and codes, and conducted all simulation tests; data analysis and producing results (e.g., graphs, figures, etc.); contributed text and editorial work.

Summary: Having shown in Paper I how our solution performs with the Rate Adaption Protocol (RAP) and TCP Friendly Rate Control (TFRC), this paper
Introduction

evaluates two pertinent RMCAT congestion control mechanisms: Network-Assisted Dynamic Adaptation (NADA) and Google Congestion Control (GCC). While showing how the solution can be adapted based on the congestion control algorithms, this paper finds that both these mechanisms exhibit aspects that allows us to use a simpler passive coupling algorithm. Passive coupling works well with relaxed time constrains and requires less signaling from the flows, which in turn enables the solution to run as a stand-alone application tool.

Paper III (Chapter 7)

Title: Start Me Up: Determining and Sharing TCP’s Initial Congestion Window

Authors: Safiqul Islam and Michael Welzl


My Contributions: Main authorship; idea of the paper; developed all ns-2 simulation scripts and codes, and conducted all simulation tests; data analysis and producing results (e.g., graphs, figures, etc.); contributed text and editorial work.

Summary: This paper introduces a simple method that paces packets by correctly maintaining the ACK clock to distribute a large initial window to a newly joined flow without creating bursts in the network. Simulation results show significant improvements in the completion time of short flows without incurring disadvantages of timer-based mechanisms.

Paper IV (Chapter 8)

Title: OpenTCP: Combining Congestion Controls of Parallel TCP Connections

Authors: Safiqul Islam, Michael Welzl, Stein Gjessing and Jianjie You

Venue: IEEE IMCEC 2016, Xi’an, China, 3-5 October 2016.

Achievement: Best paper award.

My Contributions: Main authorship; idea of the paper; developed all ns-2 simulation scripts and codes, and conducted all simulation tests; data analysis and producing results (e.g., graphs, figures, etc.); contributed text and editorial work.
Summary: This paper describes the requirements and design goals of implementing a coupled congestion control for TCP. Using ns-2 simulations, this paper reports experiments which show the benefits of combining not only the parallel LEDBAT congestion controllers, but also two different congestion control mechanisms: a LEDBAT and a TCP connection.

Paper V (Chapter 9)

Title: Single-Path TCP Congestion Control Coupling

Authors: Safiqul Islam, Michael Welzl, Kristian Hiorth, David Hayes, Øystein Dale, Grenville Armitage and Stein Gjessing

Venue: Under submission.

My Contributions: Main authorship; contribution to the idea of the paper; developed all ns-2 simulation scripts, codes and conducted all simulation tests; supervised two master students (Kristian Hiorth and Øystein Dale) for FreeBSD implementation and evaluations; data analysis and producing results (e.g., graphs, figures, etc.); contributed text and editorial work.

Summary: This paper presents a method, ctrlTCP, to combine the congestion controls of parallel TCP connections. Using both ns-2 simulations and real-life tests using a FreeBSD kernel implementation, this paper shows that ctrlTCP reduces overall queuing delay and packet losses while precisely allocating the available capacity based on the application needs. In addition, this paper presents a simple light-weight encapsulation method, TCP-in-UDP (TiU), to ensure the same path by encapsulating multiple TCP connections onto a UDP port pair.

Note: An extended technical report is included in this dissertation.

1.5.2 Related Internet Drafts

Internet Draft I (Chapter 10):

Title: Coupled Congestion Control for RTP Media (draft-ietf-rmcat-coupled-cc)
Introduction

Authors: Safiqul Islam, Michael Welzl and Stein Gjessing

Status (at the time of writing): Active Internet draft in the IETF Real-Time Media Congestion Avoidance Techniques (RMCAT) working group; the draft has passed the working group last call, and is currently waiting for IESG reviews.

My Contributions: Editorship; contributed text and editorial work as well as the IETF meeting presentations and discussions.

Summary: This draft describes a simple and flexible method to combine RTP flows originating from the same sender while minimizing the amount of changes needed to existing RTP applications. This draft specifies how this method can be applied to two RMCAT mechanisms: NADA and GCC, and provides recommendations on how to apply it to different congestion control mechanisms.

Internet Draft II (Chapter 11):

Title: TCP-CCC: Single-Path TCP Congestion Control Coupling (draft-welzl-tcp-ccc)

Authors: Michael Welzl, Safiqul Islam, Kristian Hiorth and Jianjie You

Status (at the time of writing): Active independent Internet draft under submission to the IRTF Internet Congestion Control Research Group (ICCRG).

My Contributions: Editorship; contributed text and editorial work as well as the IRTF meeting presentations and discussions.

Summary: This draft presents a TCP congestion control coupling method, TCP-CCC, which not only precisely allocates the share of the available bandwidth based on the applications’ priorities but also reduces overall delay and loss. This document highlights that connections between the same pair hosts may not traverse the same bottleneck due to load-balancing mechanisms. To address this, it presents methods to ensure that the connections traverse the same path.

Internet Draft III (Chapter 12):

Title: TCP Control Block Interdependence (draft-touch-tcpm-2140bis)

Authors: Joe Touch, Michael Welzl, Safiqul Islam and Jianjie You
**Status (at the time of writing):** Active independent Internet draft under submission to the IETF TCPM working group.

**My Contributions:** Editorship; contributed text and editorial work as well as IETF meeting presentations and discussions.

**Summary:** This draft proposes an update of RFC2140. It discusses TCP state sharing (e.g., connection state, congestion control information), often maintained on a per-connection basis in the TCP Control Block (TCB). Some of the TCB states can be shared to improve the overall transient transport performance, affecting only the TCB initialization phase. This document encompasses two ways of TCB sharing: (1) Temporal Sharing—where a newly joining connection uses the cached info from a closed connection; and (2) Ensemble sharing—where a newly joined connection is initialized using information other active concurrent connections.

1.5.3 Short Summaries / Posters (Not Included in the Thesis)


Chapter 2

Background

This section is split into three parts so that the reader is presented with necessary background information before delving into the details: Section 2.1 gives an overview of the congestion control mechanisms employed in this dissertation, Section 2.2 outlines previous works and recent advancements on coupled congestion control, and Section 2.3 introduces three different ways to detect a shared bottleneck.

2.1 Internet Congestion Control

Congestion occurs along a path when resource demands exceed the capacity. The goal of Internet congestion control mechanisms is to efficiently utilize the link capacity while maintaining a low loss ratio and small delay.

TCP is the predominant transport protocol on the Internet, and works well for applications, such as bulk file transfer. TCP congestion control [18] consists of four intertwined algorithms: slow-start, congestion avoidance, fast retransmit, and fast recovery. The first two algorithms must be used to control the amount of outstanding data that is supposed to be injected to the network. The slow-start algorithm plays two important roles at the beginning of the transmission: (i) probe the network capacity, and (ii) start the “ack-clock” that will be used to control the data transmission. Each TCP connection maintains two important state variables to regulate the sending rate, congestion window (cwnd)—this controls the number of outstanding packets in the network; and slow-start threshold (ssthresh)—this controls whether a connection will use the slow-start or congestion avoidance algorithm.

However, window based control such as TCP is not well-suited for multimedia applications. For such applications, rate based congestion control mechanisms, such as Rate Adaptation Protocol (RAP) [1] and TCP-Friendly Rate Control (TFRC)
Background

[4, 19], are suitable because they do not stop in case of no feedback. RAP—perhaps the oldest and first of its kind to consider TCP-friendliness and rate-based control—mimics TCP’s Additive-Increase Multiplicative-Decrease (AIMD). To govern the time between sending two packets, it maintains a variable, the “inter-packet gap (IPG)”. On the other hand, TFRC is the only standardized congestion control mechanism for video conferencing applications; it uses an equation to constantly update its sending rate. The receiver piggybacks the necessary input parameters such as RTO (derived from an RTT estimate) and loss event rate (calculated at the receiver end) that are fed to the equation in order to calculate the sending rate.

The aforementioned congestion control mechanisms are not well-suited for real-time interactive communication such as WebRTC. The “RTP Media Congestion Avoidance Techniques” (RMCAT) IETF working group has been established to develop suitable congestion control algorithms and a method to combine the congestion control mechanism to achieve a better sending rate allocation for the new WebRTC standard, which provides interactive communication through web browsers. Three congestion control mechanisms were proposed in RMCAT: Network-Assisted Dynamic Adaptation (NADA) [13, 20], Google Congestion Control (GCC) [11], and SCReAM [21]. Among all these mechanisms, only GCC has been widely deployed: it is used by Google Chrome, Chromium (an open-source version of Google Chrome), Firefox, and Opera browsers. In this dissertation, we only focus on NADA and GCC.

In NADA, a sender regulates its sending rate based on a receiver’s RTCP feedback. That is, a receiver periodically reports implicit (one-way delay measurements and packet drops) and explicit signals piggybacked in an RTCP feedback to a sender, where is is used to calculate the sending rate. A NADA sender uses two different modes to update its sending rate: (1) gradual rate update (updates rates based on the periodic feedback); and (2) accelerated ramp up (increases rates faster when the reported queuing delay is close to zero).

GCC employs two methods of congestion control: (i) a loss-based controller (controls the bandwidth based on packet loss), and (ii) a delay-based controller (controls the bandwidth based on delay). Both these controllers are designed to regulate the sending rate. The loss-based controller only reacts to losses when it is over 10%.

Low Extra Delay Background Transport (LEDBAT) [12, 22] is a delay-based experimental congestion control mechanism where a sender calculates its sending rate using one-way delay measurements deduced from a receiver’s acknowledgments. LEDBAT provides a less-than-best-effort (also known as “scavenger”) service for “background” applications (e.g. updates) where a strict data delivery deadline is not required.
2.2 Coupled Congestion Control

This section briefly outlines the prior significant attempts at single path coupled congestion control, application layer multiplexing, data center capacity management, and coupled congestion control for multi-path TCP.

2.2.1 Single-Path Congestion Control Coupling

Here we provide an overview of the most relevant related research works, which we categorize according to the scopes of the mechanisms.

**RFC2140 - TCP Control Block Interdependence**  To the best of our knowledge, RFC2140 [23] is the first work to outline a mechanism for coupling TCP connections by sharing TCP’s Control Block (TCB) in order to better initialize new connections. Two cases of TCB sharing were described: Temporal Sharing (a TCB of a closed connection is used to initialize a new connection), and Ensemble Sharing (TCBs of active concurrent connections are used to initialize a new connection). Both cases only improve the transient transport performance, and thus have no impact on the long-term behavior of the connections. Often, connections – with different 5-tuples – do not share the same path along the way due to mechanisms like ECMP and LAG [9]. However, RFC2140 assumes that connections between the same host-pair traverse the same path. Hence, using an Initial Window (IW) from the shared states for a new connection, not sharing the same path, can result in incorrect behavior and create sudden bursts in the network. Moreover, RFC2140 neither mentions ssthresh sharing (which can lead to incorrect states) nor suggests the use of pacing to avoid sudden bursts when using a large IW.

**TCP Fast Start**  TCP Fast Start [24] uses the concept of RFC2140 temporal sharing in order to improve the start-up performance of new connections. As mentioned earlier, giving a large IW can potentially create bursts. To avoid this problem, TCP Fast Start marks extra packets sent during the fast-start phase with a higher-drop priority flag. However, this mechanism requires in-network support, making it difficult to deploy on the Internet.

**Ensemble TCP (E-TCP)**  Ensemble TCP (E-TCP) [7] extended RFC2140’s TCB state sharing concept to allow concurrent flows to benefit from each other beyond initialization, working together so that the aggregate of the parallel concurrent connections is no more aggressive than a single TCP/Reno connection. Simulation results showed that E-TCP improves the performance of HTTP 1.0 (where a new TCP connection is initiated to download for each and every object).
Background

In E-TCP, when a flow experiences a timeout, this forces all other flows to behave in a very conservative manner, i.e., going to slow-start from congestion avoidance. A Retransmission Timeout (RTO) should only occur when no more acknowledgments arrive (TCP has been greatly improved over the years to avoid RTOs when ACKs arrive in the Fast Recovery mode). Thus, if one connection sees no ACKs in Fast Recovery, yet they do arrive for another connection, it is a mistake to force all flows to go to slow-start. Similar to RFC2140, E-TCP also assumes that connections do share a common bottleneck.

Ensemble Flow Congestion Management (EFCM) The Ensemble Flow Congestion Management (EFCM) [8] shares state like E-TCP but allows the aggregate to be collectively as aggressive as the combination of separately controlled TCP connections.

TCP operates on loss events (one or more packet losses per RTT), not individual packet losses. When congestion controls are combined, this logic should be preserved; we explain this by assuming two connections traversing the same bottleneck. A single packet drop from connection 1, two drops from connections 1 and 2 or multiple packet drops from connection 2 only should all result in the same behavior of the traffic aggregate. EFCM does not adopt this rationale, and therefore flows remain aggressive when there is a drop from connection 1 because the aggregate is not halved. Simply sharing TCP variables such as cwnd or ssthresh cannot achieve this.

EFCM also has an issue of initial ssthresh sharing: a new connection joining an aggregate with a large ssthresh value in the EFCM algorithm can potentially create bursts in the network and move the ensemble from congestion avoidance into slow-start. Similar to the aforementioned approaches, EFCM also assumes that connections between the same host-pair share a common bottleneck.

Congestion Manager The Congestion Manager (CM) [6, 10] takes the state sharing concept to the next level, using a generic congestion control by moving congestion control functionalities outside of TCP connections. It provides fully dynamic state sharing capabilities by simply maintaining all the states required to perform a congestion control mechanism in one place. It is, however, exceedingly hard to dynamically tune (i.e., turning on/off) when connections do not share a common bottleneck as it revamps the congestion control functionalities with a completely new congestion control mechanism. CM requires explicit feedback from the receiver’s transport protocol. However, many unreliable transport protocols do not explicitly provide feedback to the sender. To address this, CM also suggests to send probe messages to actively measure congestion episodes. The deployment of CM never reached beyond the experimental stack even though it is a proposed standard; Duke et al. state in RFC7414 [25]:
“Although a Proposed Standard, some pieces of the Congestion Manager sup-
port architecture have not been specified yet, and it has not achieved use or imple-
mentation beyond experimental stacks.”

2.2.2 Multiplexing

Another method to avoid the competition of multiple flows is to merge application-
layer data streams above a single transport instead of changing the congestion
control mechanisms to work together. This is done by SPDY [2] and HTTP/2 [26],
for example; these protocols multiplex web sessions on top of a single TCP connec-
tion between client and server. Multiplexing application flows onto a single TCP
connection can result in head-of-line (HoL) blocking due to TCP’s strict in-order
data delivery. Solving HoL blocking usually involves entirely different transport
protocols, such as QUIC [3] or SCTP [27].

2.2.3 Datacenter Capacity Management

There are several efforts to fairly control and share the network capacity of complete
datacenter. In EyeQ [28] bandwidth is shared and guaranteed across all users of
the datacenter and access is controlled at the edges. To achieve fairness, FairCloud
[29] uses per-flow queues in the switches and HyGenICC [30] presents a network
abstraction layer to each VM. In order to ensure that all VMs get their fair share,
somewhat static allocation of bandwidth is performed by Oktopus [31] (coordinated
centrally), SecondNet [32] (between pairs of VMs) and Gatekeeper [33].

Generally, most schemes to manage datacenter traffic operate on the data chan-
nel. This has the advantage that on-host mechanisms Seawall [34] and EyeQ, for
example, can control all traffic leaving the VM, not only TCP, and apply func-
tions ranging from congestion control to traffic shaping or scheduling. However, it
is not uncommon for hypervisors to enable direct access to hardware drivers (e.g.
VMWare’s ESXi hypervisor supports TCP Segment Offloading (TSO) for VMs),
and this puts such traffic management functions on a very time-critical critical
path.

2.2.4 Multi-Path Congestion Control Coupling

Coupled congestion control is an important part of MultiPath TCP (MPTCP) [35].
There are several proposals, e.g. LIA [36, 37], OLIA [38] and BALIA [39]. Sim-
ilar to E-TCP and the CM, these mechanisms try to make multiple flows behave
like a single flow when they traverse a single bottleneck (and [40] proposes to de-
tect whether shared bottlenecks exist and switch behavior accordingly). However,
Background

MPTCP’s coupling assumes that flows could take a different path, and ideally also traverse different bottlenecks.

MPTCP’s subflows also use different tuples in order to be able to use different paths. This is exactly the opposite of what we try to achieve in this dissertation. Using a mechanism like ours in MPTCP would result in incorrect behavior. For example, giving a large share of the aggregate to a new connection can result in using a very large Initial Window on an entirely different path. Similarly, changing priorities of flows on-the-fly would be no problem with our solution, but could result in detrimental behaviour when flows do not share a bottleneck.

2.3 Shared Bottleneck Detection

Using a coupled congestion management technique as described in this dissertation is only be appropriate when connections traverse a common bottleneck. Shared Bottleneck Detection (SBD) is therefore pivotal in order to identify such connections and group them. There are three different ways to derive whether connections share a bottleneck:

1. Multiplexing is considered to be a completely reliable measure to apply a congestion management solution. Connections having the same five-tuple ((IP source and destination address, protocol, and transport layer port number pair) are (supposed to be) treated the same way along the path. This classification is only applicable between the same pair of hosts, and is well suited for certain VPN tunnels (e.g., Generic UDP Encapsulation (GUE) [41]), or RTP flows multiplexed onto a single transport protocol [42].

2. Using a system configuration can be used to decide a shared bottleneck (e.g., a common wireless uplink). Such methods require a presumption of the network environment.

3. Measurements of e.g. correlations among measured delay and loss can be used to deduce whether flows overlap in time and share a bottleneck. This enables to combine not only flows from the same sender and receiver but also flows destined for different receivers. However, this method is not completely reliable but a recent Shared Bottleneck Detection (SBD) method [43, 44] has shown promising results. Since such methods use knowledge from the past, they cannot be perfectly reliable. We should therefore take cautionary measures to dynamically enable/disable coupled congestion control. That is, coupled congestion control mechanism should be disabled if it significantly increases delay or loss.
Chapter 3

Contributions

This chapter highlights the main contributions of this work. We contribute to a better understanding of coupling congestion controls by covering a wide range of congestion control mechanisms used for real-time media, background-transfer, and web-like traffic. An extensive experimentation of the proposed solution has been done, using the following six congestion control mechanisms:

1. Rate Adaptation Protocol (RAP), a simple rate based AIMD protocol, representing a whole class of TCP-like mechanisms.

2. TCP Friendly Rate Control (TFRC), because it updates the rates based on an equation, and is currently the only standardized congestion control mechanism aimed at supporting media flows.

3. Network Assisted Dynamic Adaptation (NADA) - because it is work in progress in RMCAT.

4. Google Congestion Control (GCC), another work-in-progress congestion control mechanism in RMCAT, and is currently deployed in web browsers (Chrome, Firefox).

5. Low Extra Delay Background Transport (LEDBAT) - because it is a delay-based mechanism, and very well known as a less-than-best-effort mechanism for services such as operating system updates. A variant of LEDBAT is used by BitTorrent.

6. Transmission Control Protocol (TCP), because it is the most widely used transport protocol for web-like traffic as well as bulk transfers.

From these experiments, we derive several key findings which make up the main contributions of this dissertation.
3.1 Overview of the Main Contributions

**Active vs Passive:** We devise a method which combines the congestion controls of multiple RTP based real-time media flows (see Chapters 5, 6 and 10). The method involves a central storage element, the “Flow State Exchange (FSE)”, and can be initiated in two ways:

1. Active FSE – initiates communication with the flows actively.
2. Passive FSE – maintains the state of the ensemble and makes it available to only the flow requesting a new rate.

As a first step, a very simple active version of the FSE has been applied to the RAP and TFRC congestion control mechanisms where we keep track of the aggregate of all the flows and assign each flow a weighted proportion of the aggregate. We have found with simulations in Chapter 5 that the mechanism yields perfect control over fairness, however it did not reduce overall queuing delay and loss. This, in turn, leads us to an important finding regarding synchronization and de-synchronization of flows: since, our solution de-synchronizes the flows, a flow experiencing congestion and halving its rate in an ensemble of 5 flows reduces the aggregate e.g. from 5Mbps to 4.5Mbps, whereas, without the coupling, they sometimes synchronize and halve their rates as well as the aggregate. We solve the problem by extending the mechanism (see Section 5.3) to emulate the behavior of one flow on congestion.

**Takeaway#1** A simple active algorithm that only assigns flows a share of the aggregate without otherwise influencing their congestion control does not significantly reduce the overall delay and losses.

For a better understanding, a high-level algorithmic description of our solution is given below:

```plaintext
When a flow updates rate:
Calculate sum of all rates

if (active){
    if (conservative) {
        If congestion:
            Calculate proportionally reduced rate for ALL flows
            Use a timer to avoid over-compensation:
            Do not allow other flows to simultaneously reduce the aggregate rate
    }
    Assign all flows their share based on their priorities
}
else // passive
    Assign flow its share based on its priority
```

22
Overview of the Main Contributions

Despite the passive version requiring less signaling and minimal changes to the flow’s congestion control, it can create problems with connections that do not update the rates for relatively long periods – typically because RTTs are significantly different (most congestion control algorithms operate on RTT timescales). This is shown in Fig. 6.3 with TCP flows where large RTT differences delay the feedback between the FSE and flows, thus preventing the solution from applying correct rates.

**Takeaway #2** Both the active and passive version of the algorithms can be applied to connections with homogeneous RTTs. However, an active version of the solution, which triggers updates to all the flows, should be used for connections with heterogeneous RTTs. Since the update interval between the connections and the solution can be relatively long because of significant RTT differences, the passive version of the algorithm should not be used.

Having applied our mechanism to RAP and TFRC, we scrutinize on applying the FSE to two proposed RMCAT congestion control mechanisms, GCC and NADA. We have shown that applying the active conservative algorithm of Chapter 5, which emulates the behavior of one flow in the presence of congestion, can provoke NADA to trigger its “accelerated ramp-up” more often. This, in turn, makes the flows aggressive.

**Takeaway #3** Reducing the aggression of an aggregate is generally better in terms of delay and loss, but it can violate the underlying CC’s assumption. This can lead to undesired behaviour.

We have also found that applying the FSE to different congestion control mechanisms requires a small adaptation to the algorithm. In doing so, we have discovered that these two mechanisms exhibit aspects which lead us to an interesting finding: both GCC and NADA update their rates at fixed intervals—not as a function of the RTT. Thus, we can use a simple passive version of the FSE—a less time-constrained request-response style of signaling between the FSE and the congestion control mechanisms. Our evaluation covers a range of pertinent test cases [45] to show the efficacy of applying passive coupling on NADA and GCC; results are detailed in Chapter 6. In addition, we show—with experiments where we have delayed the feedback between a flow and the FSE—that this less time-constrained request-response style of signaling opens the possibilities to run the FSE as a stand-alone management tool. Here is our takeaway from these tests:

**Takeaway #4** A passive congestion control coupling method can be used to combine congestions control mechanisms that update their rates at a fixed interval—not as a function of RTT (e.g., NADA, GCC). Such solution works well with relaxed time constraints, and thus, it can run as a stand-alone management tool.
Contributions

**Sender-Side vs Receiver-Side Congestion Control Decisions:** Results from the active conservative version were positive for RAP, but less favorable for TFRC. This is because TFRC increases its sending rate by the deterministic length of loss intervals, calculated at the receiver side. Therefore, forcing a TFRC flow to a lower rate than what its congestion controller has derived makes it more aggressive. Here is our takeaway:

**Takeaway #5** Any receiver-side calculation must be taken into consideration.

**Stateless vs Stateful:** WebRTC media flows, used in Chapters 5 and 6, are rate-based and stateless, hence it is easier to combine their congestion controls than with TCP. To couple TCP flows, we adopt a different design approach by correctly honouring the stateful nature of the TCP congestion control algorithms. We introduce \textit{ctrlTCP} in Chapter 9, a minimally-invasive solution that is flexible enough to cater for needs ranging from weighted fairness to potentially offering Internet-wide benefits from reduced inter-flow competition.

With experimental results, we have first shown that simply sharing TCP variables such as \textit{cwnd} or \textit{ssthresh} without taking the TCP states into consideration (as it was done in previous work) can either make an algorithm too conservative or aggressive (see Section 9.2 in Chapter 9).

We have shown the problems of not sharing states carefully using simulations with both E-TCP (Fig. 9.2(c)) and EFCM (Fig. 9.2(b)) (see discussion related to TCP state sharing problem of E-TCP and EFCM in detail in Section 2.2.1). To solve this, \textit{ctrlTCP} basically keeps the multiple TCP connections intact and lets the TCP congestion controllers communicate by correctly taking the TCP states. Chapter 9 highlights how \textit{ctrlTCP} is built on the state-of-the-art, and showcases the advantages of the solution using the ns-2 simulations and an implementation in the FreeBSD kernel.

**Takeaway #6** Whenever a congestion control mechanism is stateful (e.g., TCP, with Slow Start, Congestion Avoidance and Fast Recovery), state should also be shared to make the overall state of the aggregate correct.

We have shown that we even can control TCP connections originating from different VMs by placing our solution in a hypervisor. This allows us to take a major step forward where our solution can be applied in multi-tenant data-center networks. We illustrate how a data-center network can benefit from using a coupled congestion control mechanism in the next section.

**Ensuring a Common Bottleneck:** All the prior approaches in Section 2.2 assume that connections traverse the same path between the same pair of hosts. In practice, this becomes problematic due to load-balancing mechanisms like ECMP.
and LAG. Such a case has not been considered before, and thus gives a new insight: how can we ensure that flows traverse the same path? In the course of answering this, we survey mechanisms that ensure the same path between the same pair of hosts. In addition, we propose a light-weight, dynamically configured TCP-in-UDP (TiU) encapsulation scheme—several TCP connections in UDP datagrams carrying a single port number pair—that ensures our coupled flows do indeed share all bottlenecks along a single path. TiU is optional. Our coupled congestion control strategy is applicable to scenarios wherever overlapping TCP flows must follow the same path (such as when routed over a VPN tunnel). Chapter 9 documents the study and our encapsulation method in detail.

Takeaway#7 Whenever connections are encapsulated or multiplexed (e.g., WebRTC flows onto a transport protocol), a coupled congestion control mechanism can readily be applied all along the path.

Note that this is different form of congestion control coupling than in case of MPTCP, where the idea is that flows should take different paths. These differences are explained in detail in Section 2.2.1.

Flow Initialization - Pacing vs No Pacing: Coupled congestion control can let a window-based connection quickly increase its window by giving it a large share of the aggregate, e.g., when a connection joins, or resumes after a quiescent period. The crux of such sharing is that it can produce bursts and this can be mitigated using some form of pacing. We show the problem in Chapter 7. Prior approaches use timer-based pacing which is known to have some disadvantages. We therefore propose a novel solution to pace the packets by correctly maintaining the ACK clock instead of using a timer. This mechanism is an add-on to our ctrlTCP mechanism, but can easily be ported to better initialize new connections in order to improve the transient performance (“temporal sharing” and “ensemble sharing” of RFC2140). Chapters 7 and 9 show its positive impact on web-like short flows which can complete much faster with our ctrlTCP.

Takeaway#8 Whenever a new flow joins or resumes after a quiescent period, it may get a large share of the aggregate determined by a coupled congestion control mechanism. In case of window-based congestion controls, this may produce a sudden burst in the network. To avoid sudden rate jumps, packets therefore should be paced.

Combining a Heterogeneous Set of Congestion Controllers: Combining a heterogeneous set of congestion control mechanisms can yield several performance benefits, especially when one of the mechanisms reacts on a congestion event earlier than the others. As a first step, we have applied our solution to combine parallel
Contributions

LEDBAT flows, and after that, having shown our results with LEDBAT flows, we have tested a combination of a LEDBAT and a TCP connection (see Chapter 8). This leads us to an important finding: since LEDBAT notices the increasing delay as soon as the queue grows, our solution, in fact, ensures that the queue does not grow even for TCP. This step—combining a heterogeneous set of congestion controllers—could allow to combine the WebRTC data channel and video flows (multiplexed onto a UDP port pair) that use a loss-based and a delay-based congestion control mechanisms, respectively.

**Takeaway #9** Combinations of two different congestion mechanisms can avoid bad interaction and improve the overall performance; for example, a loss-based controller can benefit from a delay-based controller which reacts on a congestion episode earlier.

Table 3.1 summarizes the major findings in this dissertation. To apply a coupled congestion control mechanism to a new Congestion Control (CC) mechanism that exhibits the aforementioned aspects, the lessons learned in this work can help towards implementing a generic coupling solution.
Changes in algorithm aggression can improve performance (Chapter 5), but there are exceptions: it can violate the underlying CC algorithm’s assumption. This, in turn, can make the CC counteract on the imposed decision (Chapters 6 and 10).

Connections with homogeneous RTTs can use both active (Chapter 5) and passive coupling (Chapters 6, 8 and 9). However, it is recommended to use an active version for connections with heterogeneous RTTs (Chapter 5).

Congestion control mechanisms that update their rates not as a function of RTTs but e.g. at a fixed interval can use simple passive version (Chapter 6).

If the CC decisions of a connection are influenced by receiver-side CC logic, this should be incorporated into the design of a coupled congestion control solution (Chapter 5).

It is recommended to incorporate states in a coupling solution when a congestion mechanism is stateful, e.g., TCP (Chapters 8, 9 and 11). The design approaches for the stateless mechanisms are simpler (Chapters 5 and 6).

Whenever it is enforced that connections take a common path, e.g., connections are multiplexed (e.g., WebRTC flows) or encapsulated (e.g., VPNs), a coupled congestion control mechanism can always be used (Chapters 5, 6 and 8 to 11).

Giving a large share of the aggregate creates sudden bursts for window based congestion control, and therefore some form of pacing is required (Chapter 7). This can be achieved with a timer or by gradually handing over the share of the aggregate. Avoiding any increased burstiness due to CC coupling requires an algorithm to be active.

Combinations of two different congestion control mechanisms can avoid bad interaction; for example, a loss-based controller can benefit from a delay-based controller which reacts on a congestion episode earlier (Chapter 8).

<table>
<thead>
<tr>
<th>Takeaway</th>
<th>Algorithm aspect</th>
<th>Recommendation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,3</td>
<td>Changing algorithm aggression</td>
<td>Reducing aggression can improve performance (Chapter 5), but there are exceptions: it can violate the underlying CC algorithm’s assumption. This, in turn, can make the CC counteract on the imposed decision (Chapters 6 and 10).</td>
</tr>
<tr>
<td>2</td>
<td>RTT</td>
<td>Connections with homogeneous RTTs can use both active (Chapter 5) and passive coupling (Chapters 6, 8 and 9). However, it is recommended to use an active version for connections with heterogeneous RTTs (Chapter 5).</td>
</tr>
<tr>
<td>4</td>
<td>Rate updates</td>
<td>Congestion control mechanisms that update their rates not as a function of RTTs but e.g. at a fixed interval can use simple passive version (Chapter 6).</td>
</tr>
<tr>
<td>5</td>
<td>Receiver-side Logic</td>
<td>If the CC decisions of a connection are influenced by receiver-side CC logic, this should be incorporated into the design of a coupled congestion control solution (Chapter 5).</td>
</tr>
<tr>
<td>6</td>
<td>Statefulness</td>
<td>It is recommended to incorporate states in a coupling solution when a congestion mechanism is stateful, e.g., TCP (Chapters 8, 9 and 11). The design approaches for the stateless mechanisms are simpler (Chapters 5 and 6).</td>
</tr>
<tr>
<td>7</td>
<td>Ensured Common Bottleneck</td>
<td>Whenever it is enforced that connections take a common path, e.g., connections are multiplexed (e.g., WebRTC flows) or encapsulated (e.g., VPNs), a coupled congestion control mechanism can always be used (Chapters 5, 6 and 8 to 11).</td>
</tr>
<tr>
<td>8</td>
<td>Pacing</td>
<td>Giving a large share of the aggregate creates sudden bursts for window based congestion control, and therefore some form of pacing is required (Chapter 7). This can be achieved with a timer or by gradually handing over the share of the aggregate. Avoiding any increased burstiness due to CC coupling requires an algorithm to be active.</td>
</tr>
<tr>
<td>9</td>
<td>Combining Different CCs</td>
<td>Combinations of two different congestion control mechanisms can avoid bad interaction; for example, a loss-based controller can benefit from a delay-based controller which reacts on a congestion episode earlier (Chapter 8).</td>
</tr>
</tbody>
</table>

Table 3.1: Lessons learned in this dissertation
Contributions

3.2 ctrlTCP in Datacenters

Datacenters have become a cornerstone of today’s networked IT infrastructure. When the owner of the datacenter controls all communicating endpoints, a wide range of congestion control or traffic management mechanisms can be employed without having to worry about backwards compatibility. Examples of such mechanisms are DCTCP [46], TIMELY [47], EyeQ [28], HyGenICC [30], Oktopus [31] SecondNet [32], FairCloud [29].

Problem

In multi-tenant datacenters, however, backwards compatibility is an issue, yet the guest OSes of clients may be diverse and utilize an Internet-like mix of old and new TCP congestion control implementations, with and without ECN, following Reno, Cubic, Westwood or any other TCP “flavor” that e.g. Linux and FreeBSD allow to configure via their pluggable congestion control frameworks [48]. This may put some users at a disadvantage, depending on how aggressively their congestion control probes for the available capacity. Moreover, unfair users may have an incentive to obtain a larger share of the capacity by opening multiple TCP connections. This is illustrated in Figure 3.1, which shows Jain’s Fairness Index [49] for $N = 2$ aggregate flows $x_1(t)$ and $x_2(t)$, calculated using the traffic that originated from two VMs across a 10 Mbit/s $\times 100$ ms bottleneck with and without ctrlTCP.

![Figure 3.1: Fairness between two VMs, with 1 flow in VM1 and 1 to 4 flows in VM2. Without coupling, fairness deteriorates; our ctrlTCP algorithm achieves perfect fairness.](image)
Efforts are underway to address this problem by harmonizing the traffic coming from senders directly at the source; this approach has been found to have advantages in terms of scalability and resilience to churn over using switch and router mechanisms such as CoS tags, Weighted Fair Queuing, reservations [34]. Bandwidth allocation schemes in general (e.g., EyeQ [28], Gatekeeper [50], Oktopus [31], Secondnet [32], Netshare [51], FairCloud [29], Proteus [52] and Silo [53]) tend to operate on a VM-level, making them insufficient to relieve the network of congestion [54].

Mechanisms such as Seawall [34], VCC [55] and AC/DC [54] successfully achieve this sender-side control by running dedicated congestion control algorithms as part of the hypervisor infrastructure. However, they all face a common difficulty, which we address in this paper: how should the new algorithm that is running as part of the hypervisor communicate with the guest OS?

VCC and AC/DC do not require updating the guest OS at all, which is a significant advantage: it does not require cooperation of tenants to update the OS (if they do bring their own OS), which reduces burden and allows to enforce cooperative behavior. However, these approaches also have disadvantages: they have to resort to changing the receive window (rwnd) as a means to control TCP’s behavior. A sender can therefore only increase the sending rate as quickly as the TCP implementation inside the guest OS allows. A hypervisor could speed up the TCP sender inside the guest OS by splitting the TCP connection to shorten the control loop, and sending ACKs faster than the real receiver; this requires managing an additional buffer inside the hypervisor, making the solution significantly more complex than the approach that we present in this thesis.

Seawall takes a different approach: in the guest OS, congestion control implementations need to defer all congestion control decisions to the hypervisor by always asking for allowance before sending a packet [34] (similar to the Congestion Manager (CM) [6, 10]). Seawall alone takes care of congestion control. According to [34], the sheer performance gain should provide enough incentive for tenants to upgrade their OS; this is confirmed by some of our findings (e.g. the significantly shorter completion times of short flows in Fig. 9.7 and 9.8). However, Seawall needs more drastic changes to the infrastructure than e.g. VCC and AC/DC: both the sender and receiver side are altered, and bits from the header are re-purposed to implement the necessary signaling.

Many of the alternatives discussed in [55] have similar limitations: buffering packets or ACKs, duplicating ACKs, splitting connections, etc. The only viable alternative listed is to directly access the guest memory, albeit with some disadvantages as also discussed in [55].
Contributions

Solution

We argue that a middle ground can be found when keeping the guest OS congestion control intact, yet allowing a controlling entity to overrule its decisions. Making use of existing congestion control code is however close to impossible with the “ask to send” interface of the Congestion Manager: because the CM does not have knowledge about the congestion control mechanisms it is talking to, it is limited to either carrying out relatively simple scheduling decisions or implementing a complete congestion control mechanism by itself (which is what it really does).

We present a new interface to communicate between TCP in the guest OS and a hypervisor. We call a set of TCP connections that are controlled via this interface \textit{ctrlTCP\_int}. We use our \textit{ctrlTCP} algorithm (see Chapter 9) that emulates the behavior of a single TCP congestion controller (much like the CM) and supports priorities (for practical management of both inter- and intra-VM capacity allocation). \textit{ctrlTCP} can operate in a hypervisor (as shown in Fig. 3.2, and done for the test shown in Fig. 3.1) or in an OS.

Using the \textit{ctrlTCP\_int} interface, it is possible to develop a variety of algorithms that combine the individual congestion controls in some way. For example, [56] presents a simple algorithm called LISA that shares the cwnd of MPTCP subflows in slow start. This happens only at the moment when new subflows join (while a range of coupled congestion control mechanisms exist for MPTCP, they only work in congestion avoidance, leading to bursts in slow start as multiple subflows start up at the same time, potentially across the same bottleneck. MPTCP is discussed

\footnote{The interface in [6] contains a \texttt{cm\_update\_call}, which conveys information such as type of loss and RTT. While this call is better aligned with our proposal, it also does not fit the bill: the conveyed information is \textit{input} to a congestion control mechanism — but leveraging existing congestion control code requires to convey the \textit{output} of a congestion control mechanism instead.}
in more detail in Section 2.2.1). With ctrlTCP_int, the LISA algorithm would only share cwnd in the “Register” phase, provided that an existing flow in the same coupled group is in slow start. The “Update” phase would only be used to note which flows have left slow start, and the “Leave” phase would remove flows from the list of flows the CCC keeps track of.

Implementing the CM with ctrlTCP_int is also straightforward: the input from parameters in the “register” and “update” calls could generally be ignored, and the “accept” and “response” calls would be used to dictate the cwnd that a flow should use. Implementing “ask to send” would just mean that the internally-maintained CCC cwnd value would increase or decrease as determined by the CCC logic and then be given to TCP such that it can either send more data or not. This makes an internal variable explicitly visible to the outside, but changes nothing else.

To summarize, the required changes to TCP are:

- This function call, to be executed at the beginning of a TCP session:

\[
\text{register}(c, P, \text{cwnd}, \text{sssthresh}); \text{ returns: } \text{cwnd}, \text{sssthresh}, \text{state}
\]

- This function call, to be executed whenever TCP newly calculates cwnd:

\[
\text{update}(c, \text{cwnd}, \text{sssthresh}, \text{state}); \text{ returns: } \text{cwnd}, \text{sssthresh}, \text{state}
\]

- This function call, to be executed whenever a TCP session ends:

\[
\text{leave}(c)
\]

We have implemented our mechanism in the FreeBSD 11 kernel with state shared across the freely available VirtualBox\(^3\) hypervisor. Figure 3.1 was produced with this implementation.

\(^3\)https://www.virtualbox.org
Chapter 4

Conclusions

This dissertation has proposed and evaluated a simple and flexible coupled congestion control solution, built on top of the state-of-the-art, to manage parallel connections between the same host-pair. The solution has been applied to different congestion control mechanisms, and the results have shown that it can reduce overall delay and loss.

This chapter consists of two parts: Section 4.1 addresses the research questions stated in Section 1.2, and Section 4.2 explores the potential future directions and deployments.

4.1 Addressing Research Questions

- RQ1: Can a solution be simple and flexible enough to be gradually deployable without changing the underlying network?

Simple and flexible: we have shown in Chapters 5 to 9 how our proposed sender-side solution can easily be applied to different congestion control mechanisms while requiring minimal changes to the existing applications. It can also be dynamically tuned when measurement based Shared Bottleneck Detection (SBD) is used. We have also identified the implication and challenges of the prior approaches and developed our solution by mitigating them.

Gradually deployable: the proposed solutions have been implemented in ns-2, Chromium, and FreeBSD. Results from simulations and real-life experiments have confirmed that our mechanism can divide the share of the available bandwidth between the flows based on the applications’ priorities while reducing overall delay and losses. We have also shown that our mechanism can combine connections originating not only from the same host (Chapters 5 to 9), but from different hosts using it as a stand-alone tool (Chapter 6), and also from different virtual machines...
Conclusions

(Chapter 9). This large variety of deployment possibilities makes it more likely that the mechanism can be gradually introduced into the Internet.

We conclude based on the findings from Chapters 5 to 9 that our mechanism is deployable on the Internet without changing the underlying network.

− RQ2: Can we always apply a solution between the same pair of hosts?

Packets may not always traverse the same path due to load-balancing mechanisms such as ECMP and LAG. It would therefore be inappropriate for one connection to influence another connection that would take a different path. RQ2 has been addressed with respect to detecting a shared bottleneck in Section 2.3 where we discuss three approaches for Shared Bottleneck Detection.

Chapters 5 and 6 conclude that the solution can always be applied to WebRTC flows which are multiplexed on a UDP port pair. In addition, Chapter 9 also mentions several application scenarios where we can always apply our solution. To ensure the same path between the same endpoints, Chapter 9 has proposed a dynamically configurable TCP-in-UDP (TiU) encapsulation scheme where several TCP connections are encapsulated in UDP datagrams carrying a single port number pair.

− RQ3: Can we find a simple generic mechanism that can be applied to different congestion control mechanisms?

RQ3 examines how hard it is to apply our solution to different congestion control mechanisms. Our first major finding in Chapter 5 has confirmed that a simple generic mechanism cannot be blindly applied to a congestion control that involves receiver-side calculation, e.g. TFRC. Chapter 6 has used two RMCAT mechanisms to show that coupling can violate an algorithm’s underlying assumptions, requiring further changes to compensate for this problem.

While the congestion control mechanisms used in Chapters 5 and 6 are stateless, Chapter 9 shows that a coupled congestion control mechanism, when used with a stateful congestion control mechanism such as TCP, must take states into consideration. We have shown the problems in Chapter 9 and built a solution upon the state-of-the-art technologies.

Putting together, we conclude that applying a coupled congestion mechanism requires a case-by-case analysis of the congestion control algorithm (see Table 3.1). That means, our approach is quite generic but does require small adaptations.

− RQ4: What are the potential benefits of such schemes that have not been considered in the prior approaches?

RQ4 highlights the potential benefits of coupling as well as what others have not considered in their prior approaches. Almost a decade ago, the inventors of the
prior approaches designed their coupling mechanisms considering TCP flows, not real-time media flows. Therefore, impact on flow completion time and throughput gain was more important than reducing latency and losses. Chapters 5 to 9 have opted for design choices which focus on precise allocation of the share of the available bandwidth as well as reducing delay and losses.

It has been shown that simple design choices with combinations of parallel connections produce throughput gains by eliminating competition. Chapters 5 to 9 present evaluations of our mechanism, showing a significant reduction in the queuing delay and packet losses while precisely allocating a share of the available bandwidth that is available to a flow aggregate.

4.2 Future Directions

The dissertation has created a platform for additional research work. This section is split into two parts: Section 4.2.1 describes further research activities, while Section 4.2.2 highlights the IETF activities related to deployments and proposed-standard follow-up.

4.2.1 Further Research

With measurement based SBD, it could be possible to combine connections between different host pairs. It would therefore be interesting to explore this further.

Chapter 6 has shown that our solution can work with relaxed time constraints for the signaling between congestion control mechanisms and the coupling entity; it could therefore be possible to use a configuration-based SBD approach where we place our solution on a wireless access point in order to control multiple stations associated with it.

Preliminary results from Chapter 8 illustrate that a loss-based congestion control mechanism can benefit from a latency-sensitive delay-based mechanism when they are coupled together. This could allow us to combine WebRTC data and video flows since they use a loss-based and a delay-based congestion control mechanism.

Adapting our homogeneous approach to be more or less aggressive is straightforward, e.g. by changing the increase/decrease behavior as a function of the number of flows in a coupled group similar to the way in which EFCM [8] differs from E-TCP [7], or the way in which MulTCP [57] differs from TCP. We therefore encourage others to develop such extensions of our algorithm in order to investigate controlling a larger variety of congestion control mechanisms with it.

Finally, the lessons learned in this dissertation (summarized in Table 3.1) provide the necessary information to apply our generic coupling solution to other congestion control mechanisms.
Conclusions

4.2.2 Deployment

We have also proposed and presented our mechanism to the Internet Engineering Task Force (IETF) community in order to bridge the gap between the industry and academia. We have actively contributed to three IETF/IRTF Working Groups: RTP Media Congestion Avoidance Techniques (RMCAT), Internet Congestion Control Research Group (ICCRG), and TCP Maintenance and Minor Extensions (TCPM) as follows:

1. We have proposed our solution to the IETF RMCAT Working Group (WG) and provided necessary guidelines to the implementers on how to apply the mechanisms to not only WebRTC congestion control mechanisms, NADA and GCC, but also congestion control mechanisms beyond WebRTC (see Chapter 10). At the time of writing, the draft has passed the WG last call, and is now currently waiting for final reviews. Results presented in the IETF meetings are available in [58, 59, 60, 61, 62, 63, 64]. In addition, we have actively participated in the design of the RMCAT congestion control mechanisms and its requirements.

2. Chapter 11 presents a coupled congestion control mechanism for TCP flows along with the ACK-Clock mechanism (Chapter 7) and TiU encapsulation (Section 9.4). This draft is currently an Active independent Internet draft under submission to the IRTF ICCRG.

3. We have also proposed an update of RFC2140 in the IETF TCPM WG (see Chapter 12). The key changes are as follows:
   
   (a) Text on load-balancing mechanism.
   
   (b) Recommendations on sharing TCB states.
   
   (c) Discusses the impact of sharing certain states (e.g., ssthresh, RTT), and also mentions that it currently works well only for connections which are neither too short nor too long.
   
   (d) Includes the current implementation status related to TCB sharing (Temporal sharing) in Linux kernel version 4.6, FreeBSD 10 and Windows (as of October 2016).
   
   (e) Suggests using a pacing mechanism in order to avoid sudden bursts.

All these items require further contributions and participation in IETF meetings to change:

– Change the status of draft-ietf-rmcat-coupled-cc (Chapter 10) from experimental to proposed standard.
Future Directions

− Further progress on draft-welzl-tcp-ccc (Chapter 11) in the most suitable IETF WG.

− Get draft-touch-tcpm-2140bis (Chapter 12) adopted in the IETF TCPM WG.

More standardization work beyond the three items above may be needed in support of Internet-wide deployment of congestion control coupling—for example, to leverage encapsulation methods such as GUE [41] or ensure correct usage of the IPv6 flow label (see the discussion in Chapter 12).
Conclusions
References


Conclusions


41
Conclusions


Conclusions


Conclusions
Part II
Research Papers
Chapter 5

Coupled Congestion Control for RTP Media

Authors. Safiqul Islam, Michael Welzl, Stein Gjessing, and Naeem Khademi

Affiliation. Department of Informatics, University of Oslo, Norway
{safiquli, michawe, steing, naeemk}@ifi.uio.no


Abstract. Congestion occurs at a bottleneck along an Internet path; multiple flows between the same sender and receiver pairs can benefit from using only a single congestion control instance when they share the same bottleneck. These benefits include the ability to control the rate allocation between flows and reduced overall delay (multiple congestion control instances cause more queuing delay than one since each has no knowledge of the congestion episodes experienced by the others). We present a mechanism for coupling congestion control for real-time media and show its benefits by coupling multiple congestion controlled flows that share the same bottleneck.

5.1 Introduction

Multiple congestion controlled flows (e.g., TCP) between the same two hosts usually have separate congestion control instances, even when the path used by them is the same. There may be several reasons for this separation. For example, one cannot always be sure if the path is indeed the same – routing mechanisms like Equal-Cost
Coupled Congestion Control for RTP Media

Multi-Path (ECMP) may assign different flows to different paths to achieve load balancing, even when they have the same destination IP address.

Routers or other middle-boxes usually identify flows using a five-tuple of source and destination IP addresses, transport protocol, and the transport protocol’s source and destination port numbers. When – as it will be possible with the new WebRTC standard for interactive communication between web browsers – multiple flows are multiplexed over a single UDP port pair, they are normally regarded as a single flow inside the network and therefore treated in the same way. In such a setup, congestion management can be readily applied.

The new “RTP Media Congestion Avoidance Techniques” (RMCAT) IETF Working Group develops standards for RTP-based interactive real-time media. WebRTC being the major use case for these standards, RMCAT will also standardize methods for coupled congestion control, with the goal of having the best possible control over the send rate allocation. Here, we describe the first proposal for RMCAT’s coupled congestion control and show its feasibility and some of its benefits.

After a review of related work in the next section, we will introduce our method for coupling congestion control in RMCAT in Section 5.3. In Section 5.4, we show some performance evaluation results using ns-2 simulations, and Section 5.5 concludes the paper.

5.2 Related Work

The Congestion Manager (CM) [1] is the best known, and perhaps the oldest related work. It provides a common congestion management framework for all the flows from a sender going to the same receiver. Flows pass information to the CM which uses a scheduler to distribute the available bandwidth. Since the CM replaces each flow’s congestion controller with an overarching one, it is hard to implement, which may be the reason why it has never been widely deployed.

In any standard TCP implementation, each connection maintains state (e.g. the current round-trip time (RTT) and congestion window \((cwnd)\)) in a data structure called Transport Control Block (TCB). RFC 2140 [2] describes that TCB data can be shared among multiple connections in two ways: 1) Temporal Sharing, and 2) Ensemble Sharing. Temporal Sharing can be used to cache state of a closed connection, and this previous connection state can be used to later instantiate a similar connection and avoid inefficiencies. Ensemble Sharing occurs when an active host opens another concurrent connection. Among other variables, RFC 2140 discusses how \(cwnd\) can be shared in order to couple the congestion control of multiple flows.

Ensemble TCP (E-TCP) [3] utilizes the concept of TCB information reusing and sharing among existing connections. It has been designed to show the aggregate network transmission behavior of an ensemble (parallel TCP connections) as a single
TCP/Reno connection. The authors of [3] compared it with persistent HTTP 1.1, showing benefits. E-TCP does not discuss what RFC 2140 calls Temporal Sharing, i.e. reusing cached information when the network is idle because network properties might change during an idle period.

Based on E-TCP, Savoric et al. [4] proposed an Ensemble Flow Control Mechanism (EFCM) where a controller actively probes for information from the flows, and calculates the new rate for a flow by aggregating congestion properties (e.g. RTT, cwnd). They showed that EFCM increases the throughput and fairness for the flows sharing the same bottleneck.

Both E-TCP and EFCM are similar in style to the mechanism presented in this paper. However, there are some important differences: these mechanisms focus exclusively on TCP congestion control, which is window based, whereas our mechanism targets rate-based RTP applications. Neither [3] nor [4] present an evaluation of the mechanism’s impact on queuing delay or packet loss; reducing both is an important goal for us (RMCAT targets low-latency interactive applications). Since we tried to minimize changes needed to existing congestion controls, we only share rates between flows, whereas E-TCP and EFCM share not only cwnd but also other TCP-specific information such as SRTT and ssthresh.

Rather than trying to directly combine the congestion control of multiple flows, a similar behavior can also be attained by multiplexing application-level data streams onto a single connection. This can be done using e.g. SCTP, where it can lead to a significant performance benefit [5]. In [6], a performance gain was attained by transparently mapping TCP connections onto a single SCTP association. Connection reuse – with the goal of allowing TCP’s congestion window to grow larger and reduce transport-layer overhead – can also be implemented at the application layer, e.g. via persistent HTTP 1.1. However, HTTP 1.1 only allows delivery of application-level streams in the sequence in which they were requested, which can cause Head-Of-Line (HOL) blocking, e.g. when the first request involves a slow database access. This has recently been addressed by SPDY, which multiplexes data streams onto a single TCP connection [7].

5.3 The Flow State Exchange

RMCAT’s congestion control should be applicable but not limited to WebRTC. This means that we may need to jointly control flows that reside within a single application (a web browser, in case of WebRTC) or in multiple applications. In the latter case, WebRTC’s benefit of knowing that packets from multiple flows will be routed in the same way is lost. There are, however, measurement based methods to determine whether multiple flows share a bottleneck in the network; being able to make use of measurements when necessary, and supporting various intra- as well as inter-application scenarios calls for a congestion management architecture that
Coupled Congestion Control for RTP Media

is much simpler than, e.g., the well-known CM.

We have opted for an approach [8] that minimizes the amount of necessary changes to existing applications. It involves a central storage element called “Flow State Exchange” (FSE). The elements of our architecture for coupled congestion control are: the Flow State Exchange (FSE), Shared Bottleneck Detection (SBD) and Flows. The FSE is a storage element that can be implemented in two ways: active and passive. In the active version, it initiates communication with flows and SBD. However, in the passive version, it does not actively initiate communication with flows and SBD, and its only task is internal state maintenance (e.g., an implementation could use soft state to remove a flow’s data after long periods of inactivity).

Every time a flow’s congestion control mechanism would normally update its sending rate, the flow instead updates information in the FSE and performs a query on the FSE, leading to a sending rate that can be different from what the congestion controller originally determined. In the active version, the FSE additionally calculates the rates for all the other flows in the same Flow Group (FG) and actively informs their congestion controllers with a callback function. A Flow Group consists of flows which should be controlled together, i.e. they have a common network bottleneck. A FG is determined by an SBD module based on measurements or knowledge about multiplexing. An SBD module can be a part of one of the applications using the FSE, or it can be a standalone entity. We plan to develop a measurement-based SBD as future work; in this paper, we assume that FGs are known by multiplexing flows over the same UDP port pair in WebRTC.

The FSE contains a list of all flows that have registered with it. For each flow, it stores:

1. A unique flow number to identify the flow
2. The Flow Group Identifier (FGI) of the FG that it belongs to
3. A priority P, which here is assumed to be represented as a floating point number in the range from 0.1 (unimportant) to 1 (very important)
4. The calculated rate $FSE_R$, i.e. the rate that was most recently calculated by the flow’s congestion controller

Flows register themselves with SBD and FSE when they start, deregister from the FSE when they stop, and carry out an UPDATE function call every time their congestion controller calculates a new sending rate. Via UPDATE, they provide the newly calculated rate. The FSE then calculates rates for all the flows and sends them back. When a flow $f$ starts, $FSE_R$ is initialized with the congestion controller’s initial rate. SBD will assign the correct FGI. When a flow is assigned
an FGI, it adds its $FSE_R$ to $S_{CR}$. When a flow stops, its entry is removed from the list.

As a first step, we designed Algorithm 1, which simply keeps track of the total rate of all flows and assigns each flow a share that is weighted by the flow’s priority. Variables are explained in Table 5.1. Intuitively, it might seem that this simple algorithm would perform well, but our initial tests have shown that it is in fact unsatisfactory. Before we proceed to an improved version of the algorithm, we now illustrate the problem with some of our intermediate results.

![Table 5.1: Names of variables used in algorithms 1 and 2](image)

**Algorithm 1** Active FSE Rate Control

**Input:** $CC_R$ and $new_{DR}$

**Output:** $FSE_R$

1: $S_P ← 0$
2: $S_{CR} ← S_{CR} + CC_R - FSE_R(f)$
3: $FSE_R(f) ← CC_R$
4: for all flows $i$ in FG do
5:  $S_P = S_P + P(i)$
6: end for
7: for all flows $i$ in FG do
8:  $FSE_R(i) ← min(new_{DR}, ((P(i) * S_{CR})/S_P))$
9:  send $FSE_R(i)$ to the flow $i$
10: end for

We implemented the FSE in ns-2 and simulated the behavior of congestion controlled flows using a dumbbell network topology (bottleneck capacity 10Mbit/s,
Coupled Congestion Control for RTP Media

RTT 10 ms, packet size 1000 bytes, queue length of 13 packets\(^1\); for simplicity, unless otherwise mentioned, senders always had enough data to send.\(^2\) The current implementation only supports two rate-based protocols: Rate Adaptation Protocol (RAP) [10] (because it is a simple rate-based Additive Increase – Multiplicative Decrease (AIMD) scheme, hence representing a whole class of TCP-like mechanisms) and TCP Friendly Rate Control (TFRC) [11] (because it is the only standardized congestion control mechanism aimed at supporting media flows).

Jain’s fairness index is used to calculate the expected gains in fairness where a fairness index of 1 denotes that all \(n\) concurrent flows get a fair share of the total available bandwidth whereas a fairness index of \(1/n\) means that one of the \(n\) flows gets the entire available bandwidth. It is clear from the algorithm, and was also confirmed in our simulations, that the FSE achieves precise fairness among the flows. This is important, as it is a requirement for WebRTC [12] – but because coupling congestion controllers should help avoid competition at the bottleneck, we expected reduced queuing delay and packet loss, while achieving at least as much throughput as of a single flow. While the latter requirement was also fulfilled by this algorithm, the results with Algorithm 1 were disappointing regarding queuing delay and packet loss.

The loss ratio and average queue length with FSE-controlled vs. non-FSE-controlled RAP and TFRC flows are illustrated in Figures 5.1, 5.2, 5.3, and 5.4. Since we only highlight a problem, every data point in these graphs is the result of a single simulation run. It can be seen that, with the FSE, the loss ratio improves as the number of flows grows, but the average queue length is higher. Results were even worse with TFRC.

To address these problems, we investigated the queue growth over time with and without the FSE. As shown in Figure 5.5, the queue essentially oscillates between empty and full, but it does not always drain. In the same test without the FSE (see Figure 5.6), the queue failed to drain only once, in contrast to the 7 such occurrences in Figure 5.5. This is because the FSE de-synchronizes the flows. For example, consider two RAP flows, each sending at a rate \(X\). If one of these flows tries to increase its rate and immediately experiences congestion, it halves its rate, which reduces the aggregate rate from \(2X\) to \(1.5X\). However, without the FSE, when the two flows get synchronized, both halve their rate when congestion occurs which also halves the rate of the aggregate. Synchronization is usually regarded as

\(^1\)This is based on the bandwidth×delay product (BDP). We repeated our tests with different queue lengths and found no significant differences.

\(^2\)This may not be a totally unreasonable assumption for modern multimedia systems, which may be able to closely track the available bandwidth (cf. [9]). However, the actual behavior is codec-dependent and hard to characterize. At the time of writing, the RMCAT group is working on suitable test cases; in the absence of a solution in this space, we opted to investigate two extreme ends of the spectrum – the case where applications can always send data, and the case where a codec cannot adapt to the available bandwidth at all (Section 5.4).
a detrimental network effect, but in this case, it appears to play out positively.

In order to fix the loss ratio and average queue growth, we updated our algorithm
to emulate a similar behavior by proportionally reducing the aggregate rate on congestion (Algorithm 2). To better emulate the behavior of a single flow, we additionally limited the aggregate rate growth (in the absence of congestion) of \( N \) flows to \( I/N \), where \( I \) is the flow’s increase factor. In order to avoid over-reacting to congestion, we set a timer that prohibits flows other than the flow that just reduced its rate from changing their rate for two RTT periods (of the flow that reduced its rate). We decided to use 2 RTTs so that other flows do not react to the same loss interval. We assume a loss interval to persist for up to one RTT and added another RTT to compensate for fluctuations in the measured RTT value.

A local variable \( DELTA \) is used for calculating the difference between \( CC_R \) and previously stored \( FSE_R \). When \( DELTA \) is negative, we adjust the aggregate and set a timer for 2 RTTs. When the timer is not set or expired, flows operate as before and increase their rates by \( I/N \) until congestion is experienced. As we will show in the next section, these changes largely removed the problems that we observed with the first version of our algorithm.

**Algorithm 2** Conservative Active FSE Rate Control

**Input:** \( CC_R \), \( new_DR \) and RTT  
**Output:** \( FSE_R \)

1: \( S_P ← 0 \)
2: if \( Timer \) has expired or not set then
3: \( DELTA ← CC_R − FSE_R(f) \)
4: \( S_{CR} ← S_{CR} + DELTA \)
5: if \( DELTA < 0 \) then
6: \( S_{CR} ← S_{CR} - S_{CR} \times (1 - CC_R/FSE_R(f)) \)
7: Set \( Timer \) for 2 RTTs
8: end if
9: end if
10: for all flows \( i \) in \( FG \) do
11: \( S_P ← S_P + P(i) \)
12: end for
13: for all flows \( i \) in \( FG \) do
14: \( FSE_R(i) ← \min(new_DR, ((P(i) \times S_{CR})/S_P)) \)
15: send \( FSE_R(i) \) to the flow \( i \)
16: end for

5.4 Evaluation

Evaluations were carried out using ns-2 simulations\(^3\) with the same setup as described in the previous section, except that we used a larger RTT of 100ms (and

\(^3\)source code is available at: http://safiquili.at.ifi.uio.no/coupled-cc/cc-source.html
half-BDP queue of 62 packets – we also tested other queue lengths and saw consistently lower queuing delay. Different from Section 5.3, however, all tests reported here were carried out 10 times with different randomly picked start times over the first second. This produced results that had such a small standard deviation (the worst case was 0.2%) that we opted against showing error bars for the sake of clarity.

Figures 5.7 and 5.8 illustrate that the updated algorithm achieves a consistent reduction of the average queuing delay both for TFRC and RAP. Figure 5.10 shows that the loss ratio gain for FSE-controlled RAP flows also becomes noticeable as the number of flows increases. However, the result is less favorable for TFRC, as shown in Figure 5.9. This is because forcing TFRC to use a lower rate than what its congestion controller has derived causes it to increase its rate more aggressively. From [11], TFRC increases by at most 0.22 packets per RTT, as a result of the deterministic length of loss intervals measured by the receiver. When TFRC uses a lower rate than planned, the loss interval gets artificially prolonged at the receiver, which then calculates a lower value for the loss event ratio $p$, which in turn provokes a faster rate increase at the sender.

Figures 5.11 and 5.12 illustrate the link utilization for RAP and TFRC flows, with and without the FSE. The relevance of link utilization here is that sending very little obviously produces a small queue and reduces packet loss; however, because Algorithm 2 tries to emulate the behavior of one flow, it should not have a significantly smaller throughput than a single flow. As expected, in all tests, the link utilization with the FSE was at most equal or smaller than without the FSE. However, link utilization of the FSE-controlled RAP flows is higher than the link utilization of a single RAP flow. In contrast, for the FSE-controlled TFRC flows, link utilization is in some cases less than the link utilization of one flow, but the difference appears rather marginal (3% less in the worst case in our tests).

To achieve prioritization, one of the requirements of RMCAT, the FSE can calculate and assign rates based on a priority. Figure 5.13 shows how two FSE-
controlled flows change their rates based on the assigned priorities over time. The two flows started out with a priority of 1 each. After 100 seconds, the priority of flow 1 was decreased to 0.66, 0.42, 0.25 and 0.11 after 100, 150, 200 and 250 seconds, respectively. This means that a high priority flow can easily get the desired rate from the FSE without requiring any further changes in its congestion controller. The first 100 seconds of this graph also illustrate the perfect fairness that is enforced by our algorithm; we do not show Jain’s fairness index because the result was always 1 in our tests.

To illustrate the effect of changing the queue length, we also investigated the average queue length for 5, 10 and 15 RAP and TFRC flows, with and without FSE. It is clear from Figures 5.14, 5.15, 5.16 and 5.17 that average queue length is consistently lower for the FSE-controlled flows.

The loss ratio is lower for the FSE-controlled RAP flows when the queue length is half a BDP as it drains more often. However, the loss ratio is equal or slightly higher when the queue is larger. The results are less favorable with TFRC flows. The somewhat surprising increased loss despite a lower average queue is currently
Figures 5.20 and 5.21 show the positive influence on the fairness index while varying the number of RAP and TFRC flows with similar RTTs. We also investigated the fairness of 2-5 RAP and TFRC flows with different RTTs between them, with ratios up to 48:24:12:6:3. While the FSE enforces perfect fairness irrespective of the RTT, the fairness without the FSE degrades heavily in some cases. Figures 5.22, 5.23, 5.24 5.25, 5.26 and 5.27 illustrate the fairness index for 2-5 RAP flows and 4-5 TFRC flows as the RTT ratio is varied; the positive influence on the fairness for the FSE-controlled flows is noticeable. The loss ratio and average queue length are sometimes surprisingly equal or less when flows have different RTTs. This is also currently under investigation.

RMCAT targets interactive media flows, with a focus on video and audio. Other than the bulk data transfers that we have used in our evaluation so far, such flows do not always keep the send buffer full. Using such “greedy” traffic is a reasonable starting point because a mechanism that fails when its send buffer is constantly full has little chance of success when the buffer occasionally runs empty. There is an ongoing discussion in RMCAT on how to best evaluate congestion control mechanisms, given the multitude of available codecs and their different behaviors; but there is some consensus that modern codecs are able to track the transport’s calculated rate quite precisely.

In the face of these complications, we decided to use a simple approach to evaluate how well our mechanism would work with media traffic. From a transport point of view, the send buffer can either run empty or not, with variations in how quickly changes between these two states occur. We therefore ran a simulation with two flows: an application limited flow, sending based on a video trace, and a greedy flow. As it can be observed from Figure 5.28, in the presence of the congestion, FSE-
Coupled Congestion Control for RTP Media

Figure 5.14: Average queue length for 10 RAP flows while changing the queue length from 0.5 BDP (62 Packets) to 1.5 BDP (167 packets).

Figure 5.15: Average queue length for 15 RAP flows while changing the queue length from 0.5 BDP (62 Packets) to 1.5 BDP (167 packets).

Figure 5.16: Average queue length for 10 TFRC flows while changing the queue length from 0.5 BDP (62 Packets) to 1.5 BDP (167 packets).

Figure 5.17: Average queue length for 15 TFRC flows while changing the queue length from 0.5 BDP (62 Packets) to 1.5 BDP (167 packets).

coupled flows proportionally reduce their rates together, whereas synchronization causes the application-limited flows to over-react without the FSE (e.g., in the congestion events at t=5, 10 and 20 seconds in Figure 5.29).

Figure 5.30 illustrates the behavior of a greedy flow with low priority (0.2) and an application limited flow with a higher priority (1) that is sending based on a video trace. It can be observed that the low-priority flow can grab unused bandwidth as long as there is enough capacity. The bandwidth was not completely utilized in these tests because the simulation time was based on the total duration of the video trace, which was too short for the low priority flow to reach the capacity limit.

We conducted a series of simulations using synthetic background traffic in order to emulate a situation that is typical for the Internet. For this purpose we used
TMIX [13], which is a tool to generate realistic TCP application workload in ns-2. The traffic used in our simulation is taken from 60-minute trace of campus traffic at the University of North Carolina, which is available from the common TCP evaluation suite [14].

We employed a pre-processed version of this traffic which is adapted to provide an approximate load of 50% on a 10 Mbps bottleneck link based on the network topology discussed in previous sections over the course of 300 sec as simulation time. The pre-processing also included the removal of non-stationarity in the background traffic pattern by randomly shuffling different portions of the traffic pattern. The RTT of background TCP flows generated by TMIX fluctuates between the range of 80∼100 ms while the RTT of foreground TFRC flows was statically set to 100 ms,
Coupled Congestion Control for RTP Media

Figure 5.22: Fairness index for 2 RAP flows as the RTT ratio is varied, with and without FSE

Figure 5.23: Fairness index for 3 RAP flows as the RTT ratio is varied, with and without FSE

Figure 5.24: Fairness index for 4 RAP flows as the RTT ratio is varied, with and without FSE

Figure 5.25: Fairness index for 5 RAP flows as the RTT ratio is varied, with and without FSE

Figure 5.26: Fairness index for 4 TFRC flows as the RTT ratio is varied, with and without FSE

Figure 5.27: Fairness index for 5 TFRC flows as the RTT ratio is varied, with and without FSE
Figure 5.28: Application limited flow and greedy flow – with FSE

Figure 5.29: Application limited flow and greedy flow – without FSE

Figure 5.30: High-priority (1) application-limited flow #1 is hardly affected by a low-priority (0.2) greedy flow #2 as long as there is enough capacity for flow #1.

and foreground and background traffic shared the bottleneck queue.

Figure 5.31 shows the goodput values of two TFRC flows with FSE in the presence of background synthetic traffic when the priority of the first flow is set to 1, while the other flows’ priority is varied. As it can be seen from the graph, the goodputs of flows 1 and 2 are very close to the theoretical value that one might expect: for example, when the priority of flow 2 is 0.2, 0.5 and 0.8, the goodput ratio is 0.199 (instead of 0.2), 0.499 (instead of 0.5) and 0.799 (instead of 0.8), respectively. These are surprisingly precise values, seen by the receivers in the presence of synthetic background traffic with various numbers of arriving and departing flows and RTTs at any instance of time.
Coupled Congestion Control for RTP Media

5.5 Conclusions

We have presented the coupled congestion control mechanism that is currently being proposed for WebRTC in the IETF RMCAT group. Simulations with the two congestion control mechanisms RAP and TFRC indicate that, our method not only satisfies the requirements of controllable fairness with prioritization, but, by emulating the behavior of a single flow, also reduces queuing delay and packet loss without significantly affecting throughput. In case of RAP, we even saw these effects combined with better link utilization than with a single flow. The difference in behavior between the two mechanisms highlights the need to evaluate our scheme with each mechanism it is applied to.

We plan to test our method in real life as a next step. The congestion control of RMCAT is currently under development, and will probably be delay based; we therefore need to test our scheme with a delay-based congestion control too. To incorporate WebRTC’s data channel, we will investigate coupling with window-based protocols too; then we can control TCP like E-TCP and EFCM, which will enable us to compare the mechanisms against each other. At this point, it will also be necessary to investigate the effect of coupling different congestion controllers together. The evaluations in this paper were also limited to a scenario where SBD is based on multiplexing, not measurements. With measurement-based SBD, flows between different host pairs can be controlled, which means that the flows will also have different RTTs – another factor that needs to be incorporated in future evaluations.
5.6 Acknowledgments

This work is partially supported by the European Union through the FP7-ICT project RITE under contract number 317700. We would like to thank Dr. David Hayes for his guidance in conducting TMIX tests.

References


Coupled Congestion Control for RTP Media


Chapter 6

Managing Real-Time Media Flows through a Flow State Exchange

Authors. Safiqul Islam, Michael Welzl, David Hayes, and Stein Gjessing

Affiliation. Department of Informatics, University of Oslo, Norway
{safiqui, michawe, davihay, steing}@ifi.uio.no

Publication. IEEE/IFIP Network Operations and Management Symposium (NOMS), Istanbul, Turkey, April, 2016

Abstract. When multiple congestion controlled flows traverse the same network path, their resulting rate is usually an outcome of their competition at the bottleneck. The WebRTC / RTCWeb suite of standards for inter-browser communication is required to allow prioritization. This is addressed by our previously presented mechanism for coupled congestion control, called the Flow State Exchange (FSE). Here, we present our first simulation results using two mechanisms that have been proposed for IETF standardization: Google Congestion Control (GCC) and Network-Assisted Dynamic Adaptation (NADA). These two mechanisms exhibit aspects that allow us to use a simpler “passive” algorithm in our FSE. Passive coupling allows a less time-constrained request-response style of signaling between congestion control mechanisms and the FSE, which enables the FSE to run as a stand-alone management tool.
Managing Real-Time Media Flows through a Flow State Exchange

6.1 Introduction

Despite the fact that video conference applications have been widely used for many years, except for TCP Friendly Rate Control (TFRC) [1], there is no standardized congestion control mechanism available. Therefore, the “RTP Media Congestion Avoidance Techniques” (RMCAT)\(^1\) IETF Working Group has been established to develop standards for RTP-based interactive real-time media, with a focus on helping WebRTC (and the related IETF set of standards, RTCWeb). WebRTC enables interactive real-time communication between web browsers, facilitating a range of applications such as seamless video conferencing, telephony and interactive gaming. These should all be accessible as part of the web surfing experience and not require the installation of additional software. One important RTCWeb requirement for RMCAT standards is the ability to allow the WebRTC application programmer to assign priorities to flows. These priorities control how the available capacity is shared [2].

In a recent work [3] we proposed the Flow State Exchange (FSE); a mechanism that couples congestion control for RMCAT. The FSE couples the congestion controls of competing RMCAT flows with priorities—even between combinations of different congestion control mechanisms. It enables the sender to precisely control prioritized bandwidth sharing, removing self-competition. This improves the overall performance, reducing delay and loss [3].

Having shown in [3] how the FSE performs with the Rate Adaption Protocol (RAP) [4] and TFRC [1], we now investigate its usage for two proposed RMCAT congestion control mechanisms: Network-Assisted Dynamic Adaptation (NADA) [5] and Google Congestion Control (GCC) [6]. Among these congestion control mechanisms, GCC has already been deployed in Google Chrome, Chromium, Firefox, and Opera browsers.

Applying the FSE to different congestion control algorithms requires a small adaption to the FSE algorithm. In this paper, we show how this is done for NADA and GCC, leading us to an interesting observation: because both NADA and GCC update their rates at fixed time intervals – and not as a function of the RTT – we can use an even simpler, “passive” version of our algorithm. Different from our previous “active” version [3], which required immediate callbacks, the passive FSE algorithm can be implemented as a simple request-response type server, with less signaling overhead and relaxed time constraints. This makes it possible to run the FSE as a stand-alone application, potentially turning it from an integral part of an application’s congestion control into a separate tool for managing priorities between flows.

We believe that a standalone FSE tool will have uses beyond the scope of just WebRTC, where support of priorities is a requirement [2]. This belief is supported

\(^1\)http://tools.ietf.org/wg/rmcat/
Background and related work

by the results of a survey in which we asked 139 students and work colleagues whether they had experienced network traffic from different applications interfering with each other on their computers. 96 of them responded “yes, and I found it annoying”, 19 chose “yes but I did not care” and the remaining 24 chose “No: this never happened – or if it did, I did not notice”. Asked if they would use a tool that would be easy to handle and would let them prioritize how applications access the network, 89 participants said yes, 11 said no, and 39 used a free-text field to give a different answer, such as (relatively common) “I already use QoS mechanisms”. While the QoS argument is valid, it requires knowledge of and access to the most common bottleneck link (often the access point), that not all users have.

This paper is organized as follows: Section 6.2 presents background information (NADA and GCC) and related work. Section 6.3 explains our FSE algorithm and how we have changed it for NADA and GCC. In Section 6.4 we show some evaluation results using ns-2 simulations for NADA and simulations using the Chromium browser for GCC, with conclusions in Section 6.5.

6.2 Background and related work

Network-Assisted Dynamic Adaptation (NADA) [5, 7] is a congestion control scheme designed for interactive real-time media applications. In NADA, the receiver combines both implicit (per-packet drops and one way delay measurements) and explicit signals into a composite congestion signal, and periodically reports back to the sender using the Real-Time Transport Control Protocol (RTCP). The sender thus regulates its sending rate upon receipt of the RTCP feedback. It can be seen from the simulation results in [7] and [8] that the algorithm achieves good fairness by maintaining a stable queue when multiple NADA and TCP flows compete.

Google Congestion Control (GCC) [6] is another congestion control algorithm proposed for WebRTC. GCC employs two controllers: (i) a sender side controller, and (ii) a receiver side controller. The sender side controller controls the bandwidth based on packet loss, and the receiver side controller controls the bandwidth based on delay. These two controllers are designed to increase the rate in the absence of congestion. Interestingly, the sender side controller only reacts to losses over 10 percent. In [9, 10], the authors showed that in a previous version, GCC starved when competing with TCP. However, recent changes [11] show that the algorithm now can achieve good fairness while minimizing queuing delay.

There have been a number of earlier end system flow management efforts. The first work to combine flow management used a mechanism that enabled TCP’s Control Block (TCB) to be shared among flows [12]. This shared state is used to initialize new connections. Ensemble TCP (E-TCP) builds upon [12], extending it to let multiple parallel flows immediately benefit from each other [13]. The authors of [14] extend E-TCP’s concepts with their Ensemble Flow Congestion Manager
Managing Real-Time Media Flows through a Flow State Exchange

(EFCM). In E-TCP, the ensemble of $n$ TCP connections is no more aggressive than one TCP connection, however, EFCM allows the ensemble to be as aggressive as $n$ separately controlled TCP connections. In [15], an integrated Congestion Manager (CM) was proposed that replaces each flow’s congestion control mechanism with its own common rate based mechanism. CM was eventually standardized [16], however, it is hard to implement and has never been widely deployed. Our FSE does not provide a new congestion control mechanism, rather it provides a simple protocol interface that facilitates the coupling of flows using their own congestion controllers. This enables it to support an extensible list of transport protocols without the implementation complexity or single protocol dependency that has hampered adoption of the other mechanisms.

6.3 The Flow State Exchange (FSE)

Because prior approaches such as the Congestion Manager did not see wide-spread deployment and have been reputed to be too complex to implement, we have opted for an approach that minimizes changes to existing congestion control mechanisms [17], called the “Flow State Exchange” (FSE).

The FSE couples flows that compete for shared capacity. Since routers or other middle-boxes usually identify flows using a five-tuple of source and destination IP addresses, transport protocol, and the transport protocol’s source and destination port numbers, we use this to identify flows that should be coupled together. In addition, flows that have a different five-tuple, but share a common bottleneck, can also be coupled together. This can be achieved through the use of a measurement based Shared Bottleneck Detection mechanism [18, 19]—however in this paper we simplify the discussion by assuming that all coupled flows have the same five-tuple. This is the case for flows between the same two browsers in WebRTC or for a more heterogeneous set of flows in certain tunnels (e.g. when using a VPN). The FSE can be either in the same sender host or in the same local network. Fig. 6.1 shows the interaction of two media sources S1 and S2 connected to D1 and D2 with the FSE on the sources’ local network.

Applying the FSE to a congestion control algorithm requires a fundamental understanding of the algorithm and then performing some straightforward adaptations. Two major elements govern the dynamic behavior resulting from an FSE: the update frequency and the decision on how rates are increased or decreased.

6.3.1 Active vs. passive updates

The FSE can be implemented in two basic ways: active and passive. In the active version, it initiates communication with flows. In the passive version, it does not initiate communication with flows, but instead maintains the state of coupled flows,
The Flow State Exchange (FSE)

making it available to flows requesting flow state information. Fig. 6.2 illustrates the interaction between flows and the FSE in the active and passive variants.

The passive version (see Fig. 6.2(b)) is easier to implement than the active version. It requires less signalling, with inherently relaxed timing constraints, and does not require tight integration with the congestion control mechanism. This means that a passive FSE could be implemented as a stand-alone tool. However, a passive FSE can create problems with flows that do not update their rate for relatively longer periods – typically because RTTs are significantly different. This is shown in Fig. 6.3, where two TCP flows are coupled using a passive FSE in ns-2 simulations (dumbbell topology, 3 Mbps bottleneck, and a queue length of 13 packets). Because TCP updates its rate as a function of the RTT, large RTT differences also delay the feedback from the FSE. This prevents the FSE from imposing strict fairness (in the test shown in Fig. 6.3, both flows have the same priority).

Consider two flows, flow 1 with a slow update frequency (i.e., a long RTT) and flow 2 with fast update frequency (short RTT). If flow 1 sends at a high rate and flow 2 wants to decrease the rate because it sees congestion, a passive FSE algorithm will make an overall decision for the rate aggregate and assign an appropriate rate to each flow (this is a necessary part of any FSE algorithm that assigns priorities). This means that it will probably record that flow 1 should reduce its rate. However, flow 1 does not incorporate this update for a long time and keeps on sending too fast. Thus, if the rate update of flow 2 is based on an assumption about flow 1, this assumption is wrong and the overall outcome is undesirable leading to the behavior we observe in Fig. 6.3).

This can be solved by choosing the active variant (see Fig. 6.2(a)), where all flows immediately incorporate updates from each other (as in [3]). As our discussion of NADA and GCC will show that the active version is superfluous to their needs.
Managing Real-Time Media Flows through a Flow State Exchange

**Active FSE**

(a) Active FSE from [3]. The FSE initiates updates whenever the FSE state changes.

**Passive FSE**

(b) Passive FSE: flows asynchronously synchronize their state with the FSE whenever they change their sending rate or congestion window.

Figure 6.2: Active and Passive versions of the FSE. CC\_R is the rate received from the flow’s congestion controller. FSE\_R(f) is the rate calculated by the FSE. Variables are explained in Section 6.3.5 and Table 6.1.
Figure 6.3: Jain’s fairness index [20] for two TCP flows with heterogeneous RTTs coupled with the passive FSE. TCP’s fairness deteriorates as the flow RTT ratio decreases due to the lag in adopting the FSE assigned rate.

6.3.2 Increasing and decreasing rates

If the FSE algorithm leaves the increase behavior unchanged, the overall increase behavior is more aggressive when compared to a single flow. This is the behavior adopted in [3, 17]. An FSE algorithm can also force the increase behavior to be exactly like one flow. For example, if the increase behavior is additive as in AIMD, this can be done by only allowing one flow to increase the rate of the aggregate per update interval. This would be a more conservative behavior.

If the FSE algorithm leaves the decrease behavior unchanged, the overall behavior is more aggressive than only a single flow. Consider, for example, 10 AIMD (Additive-Increase, Multiplicative-Decrease) flows sending with a rate of \( x \) bits per second each without using an FSE. The aggregate rate of these flows is \( 10x \). If only one of them experiences congestion and that causes that flow to halve its rate (MD with multiplication factor 0.5), the rate aggregate will only be reduced to \( 9.5x \). In reality, flows often get synchronized, increasing the chance for multiple flows to see congestion during the same round-trip time. If all flows saw congestion at the same time in our example, the aggregate would end up at a rate of \( 5x \). Thus, depending on how synchronized the flows are, the outcome is somewhere between halving the entire aggregate or only halving the rate of a single flow.

The “active conservative” algorithm in [3, 17] forces the decrease behavior to be like one flow. This decision was taken to allow the queue to drain more often because the goal was to achieve lower loss and delay. A single congestion control like TCP also tries to avoid reacting more than once per loss event (RTT), so this behaviour also had to be incorporated into the design.
6.3.3 Application to NADA

NADA has the following relevant properties:

**Building a stable queue**

NADA primarily reacts to delay signals and updates rates gradually when a receiver reports a standing, increasing or decreasing queue; Fig. 6.4 illustrates the delay characteristics of one NADA flow compared with that of a GCC flow.

**Accelerated ramp up**

A NADA sender uses two different modes to update its sending rate: gradual rate update and accelerated ramp up. In accelerated ramp up mode, a NADA flow increases its rate faster when the reported queuing delay is close to zero [5].

**Rate update frequency**

A NADA receiver sends an update to the sender every 100 ms, and the sender updates its sending rate whenever feedback arrives.

*Derived implications for the FSE algorithm:*
The Flow State Exchange (FSE)

Increase There is no obvious reason to change the increase behavior of NADA, so the FSE can leave it unaltered, similar to the version in [3].

Decrease Here, the active conservative algorithm proposed in [3] drains the queue more often than multiple NADA flows normally would do because it proportionally reduces the rates of all flows. While this by itself is not bad, it triggers the accelerated ramp up more often, in turn rendering the flows more aggressive again. Hence we found that leaving the decrease behavior unchanged is a better choice.

Active or passive Because the rate update frequency is fixed, any error introduced by passive updates persists for at most the length of the update interval. Hence, it is limited by a fixed value (as opposed to the RTT, e.g. in TCP), and hence the predictable regular updates make the “active” FSE behavior superfluous.

6.3.4 Application to GCC

The following two properties of GCC are relevant for the FSE:

Builds a stable queue

Since GCC reacts to delay signals, it builds a stable queue similar to NADA. Fig. 6.4 illustrates the delay characteristics of one GCC flow.

Rate update frequency From the most recent specification of GCC in [6], the frequency at which a sender updates its rate (immediately upon receiving feedback from the receiver or based on a timer) is not fully defined. According to [9], the frequency of sending feedback from the receiver is still an open issue. The Google implementation in the version of Chromium that was used for this paper (47.0.2494.0) updates the sending rate every 25 ms and whenever a feedback message arrives; these messages were sent every 50 ms second or if there is rate drop of at least 3%. The latter rarely occurred in our tests, meaning that approximately every other rate update was caused by the 25 ms timer.

*Derived implications for the FSE algorithm:*

Increase There is no obvious reason to change the increase behavior of GCC, so the FSE can leave it unaltered, similar to the version in [3].

Decrease In early tests, we did not see any major issues with the decrease behavior of the active conservative algorithm proposed in [3]. However, given that the mechanism is by itself delay based, the benefits of such a conservative decrease rule
Managing Real-Time Media Flows through a Flow State Exchange

are limited so we opted for leaving the decrease behavior unchanged for the sake of simplicity.

Active or passive Because rate update timers operate independently from the RTT, the frequency at which GCC flows change their rates and hence access the FSE is also RTT-independent. Again, the “active” FSE behavior is probably unnecessary.

6.3.5 A simple passive FSE algorithm for NADA and GCC

The outcome of the active FSE algorithm design in [3] is that flows using the FSE are a bit more aggressive than one flow but less aggressive than multiple: they increase the aggregate rate faster but allow the queue to drain more often. As our discussion has shown, many of the design decisions taken in our active variants are either unnecessary or inappropriate for NADA or GCC, rendering the resulting algorithm significantly simpler.

The passive FSE algorithm is summarized in Algorithm 3; variables are explained in Table 6.1. The FSE contains a list of all flows that have registered with it. Each flow has the following state:
(i) a unique flow number to identify the flow,
(ii) the Flow Group (FG) identifier indicating the group it belongs to (all flows in the same FG are assumed to share the same bottleneck in the network),
(iii) a flow priority P,
(iv) and the calculated rate FSE R.

Each FG contains one global variable S CR which is the sum of the calculated rates of all flows in the same FG. The FSE keeps track of the total rate of all flows and assigns each flow a share that is weighted by the flow’s priority.

Algorithm 3 Passive FSE Rate Control for flow f

Input: \( CC_R(f) \)
Output: \( FSE_R(f) \)

1: \( S_P \leftarrow 0 \)
2: \( S_{CR} \leftarrow S_{CR} + CC_R(f) - FSE_R(f) \)
3: for all flows \( i \) in FG do
4: \( S_P \leftarrow S_P + P(i) \)
5: end for
6: \( FSE_R(f) \leftarrow ((P(f) \times S_{CR}) / S_P) \)
7: send FSE_R(f) to the flow
6.4 Evaluation

We implemented the FSE in ns-2\textsuperscript{2} and the Chromium browser\textsuperscript{3}. The actual behavior of media flows in the browser is codec-dependent and hard to characterize. The IETF RMCAT group has prepared a number of test cases [21] to evaluate the performance of congestion control mechanisms. In accordance with [21], all tests use the dumbbell topology depicted in Fig. 6.5, with a bottleneck queue length chosen to produce a maximum delay of 300 ms. For all figures except Fig. 6.7 and 6.8, the one-way path delay is 50 ms. From Fig. 6.7 to Fig. 6.8 the link capacity is 4 Mbps, and from Fig. 6.9 to Fig. 6.10 the link capacity is 3.5 Mbps.

In this paper, we first show the prioritization results for both NADA and GCC

\textsuperscript{2}http://www.isi.edu/nsnam/ns/
\textsuperscript{3}https://www.chromium.org
Managing Real-Time Media Flows through a Flow State Exchange

flows, then the results using two pertinent test cases\(^4\) from [21] in order to document the efficacy of our proposed solution, and finally test our system’s efficacy when feedback is delayed. The two test cases are:

1. Round-trip time fairness: In this test case, five media sources S1, S2, S3, S4, and S5 are connected to D1, D2, D3, D4, and D5 media sinks, respectively (\(n=5\) in Fig. 6.5). The one way base delays are, 10 ms for S1-D1, 25 ms for S2-D2, 50 ms for S3-D3, 100 ms for S4-D4, and 150 ms for S5-D5, respectively.

2. Media pause and resume: In this test case, three media sources S1, S2, and S3 are connected to D1, D2, and D3, respectively (\(n=3\) in Fig. 6.5). S2 is paused for 20 seconds at around 40 seconds.

Results of tests from [21] are commonly presented in the form of the sending rate and packet transit delay evolution over time. This illustrates the dynamic behavior and we follow the same format in this paper.

6.4.1 Prioritization results

The FSE calculates and assigns rates based on priorities. Fig. 6.6 shows how three FSE-controlled flows change their rates based on the assigned priorities over time. Fig. 6.6(a) and 6.6(b) illustrate the sending rates for three FSE-controlled NADA flows and three FSE-controlled GCC flows, respectively. The three flows started out with a priority of 1 each. At around 50 s, the priorities of streams 2 and 3 were decreased to 0.66 and 0.33, respectively. This means that flows get their assigned proportion of the available capacity without requiring any further changes in the congestion controllers.

6.4.2 Test case results

Fig. 6.7 and 6.8 show results for the test case “round-trip time fairness” for five RMCAT flows with different round-trip times. The one way propagation delays of flows are 10 ms, 25 ms, 50 ms, 100 ms, and 150 ms, respectively. Fig. 6.7(b) and 6.8(b) show that the FSE helps both NADA and GCC flows to converge more quickly than without the FSE (Fig. 6.7(a) and 6.8(a)).

Fig. 6.9 and 6.10 show results for both NADA and GCC with two continuous and one intermittent RMCAT flows in the “media pause and resume” test case [21]. Fig. 6.9(a) and 6.10(a) show the rate and delay characteristics without the FSE, and Fig. 6.9(b) and 6.10(b) show results with the FSE for NADA and GCC respectively. In this test, all three flows start with the same priority. At around 40 s flow 1 was paused for 20 seconds. Fig. 6.9(b) and 6.10(b) show that the FSE distributes the

\(^4\)Test results of all the test cases of the active version have been presented, see [22, 23].
Evaluation

Figure 6.6: Sending rates of 3 NADA flows and 3 GCC flows as the priorities of flows are varied at around 50 seconds. (Note that markers identify the line and not plotted points)

Figure 6.7: Sending rates and delays of five NADA flows with one way delays of 10 ms, 25 ms, 50 ms, 100 ms, and 150 ms. The FSE not only enforces perfect fairness but also helps the congestion control mechanism to converge quickly. Delay is largely unaffected. (Note that markers identify the line and not plotted points)
Managing Real-Time Media Flows through a Flow State Exchange

Figure 6.8: Sending rates and delays of five GCC flows with one way delays of 10 ms, 25 ms, 50 ms, 100 ms, and 150 ms. The FSE not only enforces perfect fairness but also helps the congestion control mechanism to converge quickly. Delay is largely unaffected. (Note that markers identify the line and not plotted points)

aggregate fairly by enforcing strict fairness. The FSE also improves the delay spike introduced by a NADA flow (see Fig. 6.9) or a GCC flow (see Fig. 6.10) after a long pause.

6.4.3 Delayed feedback tests

In order to see if our mechanism would be robust against operating system disturbance if the FSE would be run as a stand-alone application, we ran tests with 2 GCC flows where we delayed the signal between the congestion controller of flow 1 and the FSE while keeping the delay between the congestion controller of flow 2 and the FSE fixed. The link capacity was 3 Mbps. It can be seen from Fig. 6.11 that the rate allocation is slightly affected when we increase the fixed delay from 50 ms to 100 ms (see Fig. 6.11(a) and 6.11(b) compared to tests without delayed feedback (see Fig. 6.8(b) and 6.10(b)). Realistically, the delay between the FSE and a flow’s congestion controller may fluctuate. To illustrate this behavior, we vary the delay between the congestion controller of flow 1 and the FSE. Delay is selected from a uniformly random distribution where values vary between 1 and 100 ms. Fig. 6.11(c) demonstrates the robustness of the FSE as the flows can achieve a fair rate allocation even with delayed feedback. The same tests with two NADA flows also yielded very good rate allocation, just as in Fig. 6.7(b).

We also carried out tests with delay values beyond the ones shown in Fig. 6.11 (even up to 500 ms), with similar results. Less time-constrained request-response
Evaluation

Figure 6.9: Sending rates and delays of two continuous and one intermittent NADA flows, with and without the FSE. (Note that markers identify the line and not plotted points)

Figure 6.10: Sending rates and delays of two continuous and one intermittent GCC flows, with and without the FSE. (Note that markers identify the line and not plotted points)
Figure 6.11: Two GCC flows coupled via the FSE, where delay between stream 1 and the FSE is varied based on (a) fixed delay of 50 ms (b) fixed delay of 100 ms, and (c) uniformly distributed random delay, between 1 and 100 ms.
style signaling between congestion control mechanisms and the FSE adds to its robustness. This makes it possible to run the FSE as a stand-alone application.

6.5 Conclusions

In this paper, we have evaluated our FSE management mechanism with the two congestion control algorithms NADA and GCC. The evaluations show that the simpler, passive version of FSE works very well with both NADA and GCC flows. Both NADA and GCC update their rates at fixed intervals, and we have argued that this is the reason the passive version of the FSE manages these flows so well.

A passive FSE uses less signaling than an active one. In addition we have shown that a passive FSE works well with relaxed timing constraints vis-à-vis the congestion controllers, making it possible to run the FSE as a stand-alone tool and even on a separate system, i.e. as a server that clients query to obtain the right sending rate. While this could multiply the benefits of the FSE by uniformly controlling several hosts, the FSE always needs to be aware of the common bottleneck, which may complicate the design of such a system. We plan to investigate this in future work. In the RMCAT context, we will also evaluate the passive FSE with the third currently proposed mechanism, SCReAM [24]. We expect similar results because SCReAM also employs a fixed feedback frequency. However, the passive FSE mechanism could be applied to any set of flows that share similar properties, not just real-time media flows. We therefore also plan to investigate whether the passive FSE can be used to control such non-real-time media mechanisms in future work.

6.6 Acknowledgments

This work is partially supported by the European Union through the FP7-ICT project RITE under contract number 317700. The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the overall quality of the paper. The authors would also like to thank Xiaoqing Zhu and Stefan Holmer for helping out with NADA and GCC respectively, and Kristian Hiorth for his advice regarding Chromium.

References


Managing Real-Time Media Flows through a Flow State Exchange


Managing Real-Time Media Flows through a Flow State Exchange
Chapter 7

Start Me Up: Determining and Sharing TCP’s Initial Congestion Window

Authors. Safiqul Islam and Michael Welzl

Affiliation. Department of Informatics, University of Oslo, Norway
{safiquili, michawe}@ifi.uio.no


Abstract. When multiple TCP connections are used between the same host pair, they often share a common bottleneck – especially when they are encapsulated together, e.g. in VPN scenarios. Then, all connections after the first should not have to guess the right initial value for the congestion window, but rather get the appropriate value from other connections. This allows short flows to complete much faster – but it can also lead to large bursts that cause problems on their own. Prior work used timer-based pacing methods to alleviate this problem; we introduce a new algorithm that “paces” packets by instead correctly maintaining the ACK clock, and show its positive impact in combination with a previously presented congestion coupling algorithm.

7.1 Introduction

Finding a suitable initial congestion window (cwnd) has been debated for many years in the IETF. A large Initial Window (IW) can be very beneficial [1], yet prob-
Start Me Up: Determining and Sharing TCP’s Initial Congestion Window

lematic for low bandwidth links, e.g. in developing countries (although RFC 6928 [2] discusses a study involving South America and Africa, this has been criticized for focusing on the flows that used the proposed larger initial window instead of measuring the impact on competing traffic). This is a difficult engineering trade-off because TCP normally assumes no prior knowledge about the path when it applies the IW.

Often, concurrent TCP connections are used between the same source-destination pair. They can share a network bottleneck, in particular when they are in a tunnel, e.g. in case of a VPN. In such cases, it would be possible for newly joining flows to either use a cached connection state or a share of an aggregate from the ongoing transfers instead of “blindly” applying a constant value. RFC 2140 [3] describes these two cases as temporal and ensemble sharing; this paper is concerned with the latter case.

7.2 Background

Each TCP connection maintains states (e.g., local process states, RTT, cwnd, ssthresh) in a data structure called Transport Control Block (TCB). Sharing TCB data across parallel TCP connections can improve transient performance during the initialization phase [3]. Ensemble-TCP (E-TCP) [4] expanded the idea of (ensemble) sharing of TCB data across parallel TCP connections in order to allow active connections to continuously benefit from each other. In E-TCP, the aggregate of an ensemble is no more aggressive than one TCP Reno connection. Ensemble Flow Congestion Management (EFCM) [5] extended E-TCP by allowing the aggregate of an ensemble to be as aggressive as n TCP connections. These two mechanisms both realize a service of joint congestion control, somewhat similar to the Congestion Manager (CM) [6] or when using application stream multiplexing as in e.g. SCTP [7] or QUIC [8] – but such sharing is easier to implement than the former and does not require application involvement as in the latter.

We have recently complemented E-TCP and EFCM with an algorithm that addresses some problems that both of these mechanisms have,\(^1\) as well as a possible encapsulation scheme to ensure that connections traverse a common bottleneck, in an Internet-draft [9]. The coupling algorithm in [9] is inspired by our prior work on coupling for media flows in the context of WebRTC / RMCAT [10, 11].

Sharing TCB data can be particularly beneficial for short flows (e.g., web on/off traffic); short flows joining an aggregate can significantly reduce their completion

\(^1\)TCP congestion control is stateful, but these states are not addressed in the E-TCP and EFCM algorithms. For example, slow start after a retransmission timeout (RTO) should not happen on one flow while ACKs still arrive on another flow. Also, to emulate the backoff of a single flow, TCP’s concept of loss events should be retained for the aggregate, meaning that there should be only one backoff irrespective of the number of losses within the same loss “round”. 

88
time due to acquiring a share of potentially large cwnd from active connections. The crux of such sharing/initialization is that it can create sudden bursts in the network, potentially leading to queue growth and packet loss. E-TCP and EFCM acknowledged this problem, and addressed it using pacing. Most pacing methods (including the ones proposed for E-TCP and EFCM) use timers to clock out packets at regular intervals over an RTT. This is not without problems, both in terms of implementation in the end host as well as (somewhat counter-intuitively) the impact inside the network [12].

The algorithm in [9] does not include any form of pacing, and therefore produces bursts that can lead to the problems described above (potentially diminished by burst limitation mechanisms underneath, e.g. in Linux [13]). We supply a mechanism to avoid sudden bursts in this paper. Different from prior work, this mechanism does not rely on a timer but simply maintains the ACK clock of TCP, thereby minimizing the impact on the dynamics in the network.

7.3 Design

We begin by showing what happens when a new flow gets a share of a large aggregate with our mechanism in [9]; we simulate the behavior of three TCP connections in the ns-2 simulator\(^2\) with a dumbbell topology (bottleneck capacity 10 Mbps, RTT 100 ms, packet size 1500 bytes, and queue length of 1 BDP (83 packets)). Connections 2 and 3 join after 5 and 6 seconds, respectively, and they receive large cwnd values from the aggregate. Fig. 7.1(a) shows a time-sequence plot of connections 2 and 3. The congestion spike due to sudden bursts from connection 2 causes significant packet losses. Appropriate mixing of the two coupled flows did not play out well until the 3rd RTT for connection 2. A small burst is also visible when connection 3 joins.

We propose a simple mechanism to avoid these bursts. Rather than using timers, we make use of the ACKs connection 1 receives to clock packet transmissions of connection 2 over the course of the first RTT when connection 2 joins. Similarly, we make use of the ACKs of connections 1 and 2 to clock packet transmissions of connection 3. In this way, we avoid causing a burst in the network. Fig. 7.1(b) illustrates that using the ack clock from the preexisting connection eliminates the congestion that is shown in Fig. 7.1(a).

When a connection c joins, it turns on the ack-clock feature and calculates the share of the aggregate, cwnd\(_c\). Algorithm 4 illustrates the ack-clock mechanism that is used to distribute the share of the cwnd based on the acknowledgements received from other flows.

\(^2\)We used the TCP-Linux module that allowed us to use TCP code from Linux kernel (3.17.4) in simulations.
Algorithm 4 Ack clocking for new connection c, running as a replacement of any other connection’s cwnd increase. Initially, number_of_acks_c = 0 and clocked_cwnd_c is c’s target cwnd calculated by the algorithm in [9]. N = number of flows.

1: if clocked_cwnd_c != 0 then
2:     return \[\Delta\] alg. ends; other connections
3: \[\Delta\] can increase cwnd again
4: end if
5: if number_of_acks_c % N = 0 then
6:     send a new segment for connection c
7:     clocked_cwnd_c ← clocked_cwnd_c - 1
8: end if
9: number_of_acks_c ← number_of_acks_c + 1

Figure 7.1: Coupling of 3 connections when connections 2 and 3 join after 5 and 6 seconds
Fig. 7.2 demonstrates the reduction of a short flow’s completion times by immediately taking a share of the aggregate. This simulation was repeated 10 times with randomly picked flow start times over the first second for the long flow (25 Mb) and the sixth second for the short flow (200 Kb). We used a dumbbell topology (RTT 100 ms, MTU 1500 bytes, queue length 1 BDP) while varying the capacity from 1 Mbps to 10 Mbps. It can be seen from Fig. 7.2 that there is a significant improvement in the short flow’s completion time using our ack-clock mechanism, and the FCT is reduced by more than 40% for all other bottleneck capacities except 1 and 2 Mbps. The reduced competition also makes the behavior more predictable: the dip at 2 Mbps only exists when flows compete (here, the queue had just enough space for one, but not two flows each sending their Initial Window (IW)).

Because the long flow gets to rapidly increase its cwnd when a short flow terminates, the ack-clock mechanism reduced the FCT of the long flow too, but only by a negligible amount: only 0.66% or less in all cases.

7.4 Conclusion

We have presented an extension of our TCP congestion control coupling algorithm in [9] to maintain ACK clocking for multiple flows as if they were only a single flow.
This allows to let newly starting flows of an aggregate quickly reap the benefit of an already large congestion window, reducing the flow completion times of short flows without incurring disadvantages of timer-based pacing methods.

We have not discussed what happens when another flow joins while this ACK clocking algorithm is active. This requires a slight extension of the algorithm that we will tackle in future work, together with other extensions of the algorithm in [9], e.g. to correctly handle quiescent senders. After a few such updates, we are confident that this algorithm will work significantly better than multiple competing TCPs in all possible cases, such that it would simply be a mistake to leave TCP connections uncoupled in situations where they are already encapsulated together (e.g. VPNs).

7.5 Acknowledgments

This work has received funding from Huawei Technologies Co., Ltd., and the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 644334 (NEAT). The views expressed are solely those of the authors.

References


References


Start Me Up: Determining and Sharing TCP’s Initial Congestion Window
Chapter 8

OpenTCP: Combining Congestion Controls of Parallel TCP Connections

Authors. Safiqul Islam (1), Michael Welzl (1), Stein Gjessing (1), and Jianjie You (2)

Affiliation.
(1) Department of Informatics, University of Oslo, Norway
{safiqui, michawe, steing}@ifi.uio.no
(2) Huawei
youjianjie@huawei.com

Publication. IEEE IMCEC 2016, Xi’an, China, October 2016.

Abstract. Most Internet communication uses the Transmission Control Protocol (TCP) at the transport layer. TCP carries out many tasks, including reliability, flow control and congestion control. Congestion control is a mechanism that is concerned with the path between two hosts, but TCP instances operate for separate communicating processes (as identified by port numbers). This means that multiple TCP connections between the same pair of hosts compete on the network. We propose OpenTCP – a method to combine the congestion controls of multiple TCP connections. We report experiments which show that OpenTCP improves the performance of concurrent TCP connections in terms of packet loss and queuing...
OpenTCP: Combining Congestion Controls of Parallel TCP Connections

delay. OpenTCP also allows to divide the available bandwidth between the flows according to the needs of the applications.

8.1 Introduction

Often, multiple TCP connections are initiated between the same two Internet hosts. For example, when browsing the web, using separate connections for images and other files is quite common (this has only recently been addressed by multi-streaming in HTTP/2, the successor of SPDY [1], but older websites will still require opening several TCP connections for a long time). Other examples include servers that provide multiple services – e.g. the Google Content Distribution Network (CDN) offers YouTube, Google search and many other Google services, and it is then possible that a Google search and a Youtube stream require TCP connections between the same pair of IP addresses.

TCP has traditionally been a “closed” protocol. Its behavior is very strictly based on a large set of standards ([2]), which include the specification of congestion control between two endpoints. Here, an endpoint is attached to a port, i.e. multiple parallel connections between the same IP addresses use multiple endpoints. TCP congestion control begins with a phase called “Slow Start”, where the sending rate (as determined by a variable called the “Congestion Window” (cwnd)) is doubled every Round-Trip Time (RTT) – the time it takes for a packet from the sender to reach the receiver and for the response to return. This phase terminates when, upon exceeding the capacity of the bottleneck, one or more packets are dropped (or congestion-marked [3]). Then, after repairing the loss, “Congestion Avoidance” begins, making the TCP sender increase cwnd by 1 packet per RTT and halve it when incoming acknowledgments (ACKs) inform the sender about packet loss (or marks).

This process repeatedly produces queuing delay whenever the sending rate begins to exceed the bottleneck capacity, and – in the absence of a marking mechanism – results in repeated packet loss as well. Since TCP retransmits lost packets, this also produces even more delay. It is obvious that these problems are magnified when multiple TCP connections individually operate across the same bottleneck. Figure 8.1 shows the cwnd values of 4 TCP flows that compete across the same bottleneck, from a simulation using the ns-2 network simulator with TCP-Linux, a module that allows to run the actual Linux TCP code in simulations (Linux kernel 3.17.4 in our case). The 4 flows competed across a 10 Mbit/s bottleneck in a dumbbell topology with an RTT of 100 ms. The bottleneck queue was a normal FIFO (“DropTail”) queue, with a length of one bandwidth×delay-product (BDP), which was 83 packets (the packet size was 1500 bytes). Background traffic was generated using TMIX traffic [4] from a 60-minute trace of campus traffic at Univ. North Carolina (available from the TCP evaluation suite [5]. RTTs of background flows
varied between 80 and 100 ms. The average link utilization, loss ratio and queue length in this simulation were 68%, 0.78% and 58 packets, respectively.

This behavior results in a form of fairness that has been called “TCP-friendliness”, which has recently been much criticized [6]. In fact, however, when flows originate from the same host and go to the same destination, fairness between these flows should be a system policy – it seems pointless to have to accept a form of fairness that evolves from the competition in the network. We have run a preliminary simulation using the algorithm described in [7] to combine the congestion controls of these TCP flows; the resulting diagram, shown in Figure 8.2, shows only one line for all four flows because they were able to use precisely the same cwnd values. With this algorithm, priorities can be used to divide the total cwnd among the flows exactly as desired (not necessarily equal as in the diagram).

The average link utilization, loss ratio and queue length in this simulation were 66%, 0.13% and 37 packets, respectively. This is a pronounced reduction in packet loss and queuing delay, at the cost of slightly reduced utilization: multiple separate TCP congestion controllers achieve better utilization because together, they more aggressively probe for available capacity than a single or combined congestion controller. However, this is only a side-effect of their behavior and not necessarily ideal: when desired, a single TCP connection can be made to be much more aggressive, and this is in fact done by the experimental CUBIC congestion control mechanism.
OpenTCP: Combining Congestion Controls of Parallel TCP Connections

Figure 8.2: cwnd of 4 OpenTCP flows

[8] (default in Linux) and many others.

In the next section, we will describe related work and explain why it has not become commonly used, leaving the problem unsolved. Then, in Section 8.3, we will describe the requirements and our derived design of OpenTCP. Section 8.4 concludes.

8.2 Related Work

The idea of combining the congestion controls of multiple flows when they traverse the same network bottleneck is not new. To the best of our knowledge, the oldest – and perhaps still best known – proposal along these lines is the Congestion Manager (CM) [9, 10]: here, there is a single congestion control instance in the host that is external to protocols such as TCP, and the flows essentially request their next cwnd or rate value from the CM. This is a good method, but has turned out to be exceedingly hard to implement; moreover, it is even harder to allow the code to make coupling optional (switch it on or off).

The two mechanisms “Ensemble-TCP” (E-TCP) [11] and “Ensemble Flow Congestion Management” (EFCM) [12] share TCP variables such as the cwnd across connections. This approach is close to what we propose for OpenTCP, but it is too simplistic: cwnd and the other variables shared by E-TCP and EFCM are variables
Related Work

<table>
<thead>
<tr>
<th>Reference</th>
<th>Information usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFC2140 [13], case “temporal sharing”</td>
<td>Flow 1 has ended, flow 2 joins. Flow 2 uses the information only at the very beginning.</td>
</tr>
<tr>
<td>RFC2140 [13], case “ensemble sharing”</td>
<td>Flow 1 is ongoing, flow 2 joins. Flow 2 uses the information only at the very beginning.</td>
</tr>
<tr>
<td>E-TCP [11], EFCM [12], CM [9, 10], TiU [7]</td>
<td>Flow 1 is ongoing, flow 2 joins, both flows use the information until they end.</td>
</tr>
</tbody>
</table>

Table 8.1: Overview of work related to sharing congestion information, using two flows as an example

of a stateful algorithm, and not correctly sharing the state can produce errors. Here are two examples:

1. TCP reacts once to any number of packet losses within one “loss event” (typically one RTT). For example, losing one or two packets within the same RTT does not normally change the sender behavior: cwnd is halved once. Simply sharing cwnd across multiple flows produces wrong behavior that, depending on how sharing is implemented, can either become too aggressive or too conservative. This is because the duration of a loss event must be shared too, not just cwnd.

2. TCP should never enter Slow Start when ACKs arrive. If one flow stays in Congestion Avoidance and another flow experiences a timeout, this second flow will enter Slow Start, but it really should not. Again, simply sharing cwnd does not solve this problem: the states of the TCP flows should also be shared.

These issues are addressed by our preliminary sharing algorithm in [7], but this algorithm still has some missing features, as we will discuss in Section 8.3. TCP Control Block (TCB) Sharing [13] resembles E-TCP and EFCM in that it simply shares variables, but it is more limited in that this sharing only affects the beginning of new flows. Table 8.1 provides an overview of all related work that has been mentioned so far.

In addition to the individual problems mentioned with the previously discussed mechanisms, there is a major common issue of all the related works that are summarized in Table 8.1, with the exception of our own earlier preliminary work [7]): while routers should in theory only be concerned with IP addresses, it is the operational reality of the Internet that two packets destined for the same IP address may take a different path if they have different destination TCP port numbers [14]. In this case, coupling their congestion controllers can go very wrong, as two TCP connections between the same hosts may in fact not traverse the same bottleneck.
OpenTCP: Combining Congestion Controls of Parallel TCP Connections

This problem is an obstacle to deployment of mechanisms to combine congestion controls. TiU [7] addresses this problem by encapsulating multiple TCP connections using a single UDP port number pair. The traffic then looks like a single UDP flow, which has some disadvantages (also explained in [7]), including the need for the receiver to understand the encapsulation and undo it.

Multi-Path TCP (MPTCP) also has a form of coupled congestion control [15], which is similar to OpenTCP in that it tries to make sure that multiple flows (in MPTCP’s case, “subflows”) act like a single TCP connection when they traverse the same network bottleneck. There are, however, important differences: MPTCP’s coupling assumes that flows can (actually should) take a different path. When they do use different paths, some of the things that OpenTCP does would become very inappropriate:

- A new connection joining the aggregate can immediately get a share of a potentially quite large cwnd of ongoing transfers. This can significantly reduce the completion time of short flows, but in MPTCP it could be harmful as it would be like using a very large initial window on a new path (TCP’s initial window has recently been increased from 3 to 10 [16] after long debate; 10 has been found to not always be appropriate and clearly, using an arbitrarily large value can be quite problematic).

- OpenTCP should avoid that connections are in Slow Start when ACKs arrive. When a new connection joins a group of connections that are in Congestion Avoidance, it should immediately start in the Congestion Avoidance phase. This is also not the appropriate behavior to start a new connection when it traverses a different path, and therefore MPTCP’s own coupled congestion control maintains initial Slow Start under all circumstances [15].

- OpenTCP allows to assign a share of the aggregate cwnd via a priority. This can also lead to inappropriate behavior when traversing different paths, especially if these priorities can change during the lifetime of connections (which OpenTCP can easily support).

Finally, MPTCP’s subflows use different TCP five-tuples in order to get different paths (cf. [17], which exploits this property even when end hosts have only one interface) – as we will explain in Section 8.3, this is almost the opposite of what OpenTCP is trying to achieve.
8.3 OpenTCP

8.3.1 Requirements

Our analysis of the problem as well as the state of the art lets us derive the following requirements for OpenTCP:

1. **Simple to implement:** for example, the CM [9] has never become widely used, and it is exceedingly hard to implement. We therefore assume that OpenTCP must be much easier to implement.

2. **Correctly share TCP states:** While being simpler than the CM, problems with E-TCP and EFCM of sharing too little state have been identified. OpenTCP must avoid these problems.

3. **Ensure that packets traverse the same bottleneck:** this has only been preliminarily addressed by our own earlier work [7], and it is a problem with all the other related work.

8.3.2 OpenTCP design

Addressing item 1, our previous work has shown that it can be easy to combine congestion controllers of real-time media applications [18, 19]. Because of item 2, sharing the state of the TCP congestion controller is a little more sophisticated. This is shown by our preliminary algorithm in [7]; however, it is clear that gradual extensions of this algorithm will still be much simpler and much easier to implement than the CM.

Item 3 is so far only addressed by [7], which encapsulates multiple TCP connections in UDP, using the same UDP port number pair for all of them. This, however, requires new code on the receiver side and may have other problems (e.g., sometimes UDP traffic is rate-limited in the network). For OpenTCP, we can consider other possibilities:

- The IPv6 flow label [20] identifies packets belonging to the same flow, and allows for easier classification in routers based on a 3-tuple of IP addresses and flow label rather than the common 5-tuple (IP addresses, protocol number, port numbers). If a sender sets the same flow label for the multiple combined connections of OpenTCP, it becomes easy for routers to forward them along the same path and no receiver-side code is needed.

- Generic UDP Encapsulation (GUE) gives us another possibility to multiplex several TCP connections over the same UDP port number pair, but using a method that is already deployed in Linux, meaning that we do not need to change receiver code.
OpenTCP: Combining Congestion Controls of Parallel TCP Connections

- VPNs can use various methods to tunnel all traffic over what looks like a single connection to the network, and hence OpenTCP will work seamlessly for VPNs.

When a new flow joins a long flow that has been operating over a high-capacity network, it may be able to immediately obtain a large cwnd value to start with. If it then immediately makes use of this value, this can produce a burst of back-to-back packets on the wire, which has a high chance of creating transient congestion and loss. E-TCP and EFCM suggest pacing, which is also not without problems (issues related to timer granularity etc); rather, cwnd could be gradually handed over to the new flow, appropriately increasing it whenever an ACK arrives. This method is planned for OpenTCP, and not yet included in the preliminary algorithm in [7]; it maintains TCP’s ACK-clocking without requiring extra timers.

Another issue that has not yet been addressed by any prior work on TCP congestion control coupling is the sharing of cwnd when applications are not fully using their credit. When TCP connections become application-limited, they cannot know about the current congestion state of the network, and this leads to a more conservative recommended behavior in the TCP standard [21]. However, at the same time, a different connection might be fully using cwnd and probing for network capacity – OpenTCP then allows the limited connection to benefit from the other one.

8.3.3 Next steps

The OpenTCP algorithm can be applied to very different congestion control mechanisms and even allows combining a heterogeneous set of congestion controllers with minimal changes. To illustrate this, we first implemented a simple mechanism to combine parallel LEDBAT [22, 23] connections in ns-2 and simulated their behavior using a dumbbell network topology (bottleneck capacity 10 Mbit/s, RTT 100 ms, packet size 1500 bytes). All tests reported in Fig. 8.3 and Fig. 8.4 were carried out 10 times with different randomly picked start times over the first second. It can be seen that OpenTCP reduces the average queuing delay without harming utilization. Our tests also showed that fairness becomes perfect and controllable, e.g. by priorities.

Having established that jointly controlling LEDBAT flows works, we now turn to an evaluation of a combination of TCP and LEDBAT using OpenTCP. LEDBAT is an interesting mechanism for this evaluation because it tries to “step out of the way” of other congestion control mechanisms. Figure 8.5 shows this behavior: TCP attains an average cwnd of around 30 packets, and LEDBAT is pushed aside, getting a cwnd close to 0. While LEDBAT is meant for low-priority background traffic, this behavior is quite extreme and it might be interesting to give it a share of the bandwidth based on a priority instead.
Figure 8.3: Average queue length of LEDBAT connections, with and without the OpenTCP algorithm.

Figure 8.4: Link utilization of LEDBAT connections, with and without the OpenTCP algorithm.
OpenTCP: Combining Congestion Controls of Parallel TCP Connections

Figure 8.5: Closed TCP: LEDBAT gets “pushed aside”

Figure 8.6: OpenTCP: TCP and LEDBAT can be fair if needed
For the test depicted in Fig. 8.6, we decided that LEDBAT traffic is just as important as TCP traffic, giving both connections the same priorities. As we can see, the cwnd becomes almost exactly equal, at around 10 packets per flow. Note that the total of 20 packets is smaller than the previously mentioned TCP average of around 30 in Fig. 8.5. This is because TCP increases its rate until the queue overflows before reacting. LEDBAT, however, notices increasing delay as soon as the queue grows and stops increasing its cwnd. Our algorithm notices this reaction and, when used for both TCP and LEDBAT, ensures that none of the connections make the queue grow. In this way, OpenTCP lets the TCP connection benefit from the good delay reduction behavior of LEDBAT. Indeed, our tests showed a pronounced delay reduction when OpenTCP was used while maintaining approximately equal bottleneck utilization in both cases.

8.4 Conclusion

In this paper we have proposed OpenTCP as a method to coordinate the behaviour of multiple TCP connections between the same endpoints. The behavior of multiple “closed” standard TCP connections is very complex and may not behave according to the needs of the application. OpenTCP puts the application programmer back in control, and makes it easy to implement open and transparent packet scheduling algorithms according to the needs of the users.

The experiments reported in this paper show that, with OpenTCP, it is easy to implement mechanisms that improve the performance of concurrent TCP connections. OpenTCP reduces packet loss and queuing delay. We have also explained that it is possible to divide the available bandwidth between the flows according to the needs of the applications. Hence, in OpenTCP, the application is in full control of the communication resources.

In future work we will show more examples of the use of OpenTCP, both based on different underlying TCP-algorithms and on different needs from the applications. We will conduct real-life experiments to prove that OpenTCP can efficiently control several concurrent end-to-end flows.

8.5 Acknowledgment

This work was funded by a collaborative project between the University of Oslo and Huawei Technologies Co., Ltd.
OpenTCP: Combining Congestion Controls of Parallel TCP Connections

References


References


107
OpenTCP: Combining Congestion Controls of Parallel TCP Connections


Chapter 9

Single-Path TCP Congestion Control Coupling

Authors. Safiqul Islam (1), Michael Welzl (1), Kristian Hiorth (1), David Hayes (2), Øystein Dale (1), Grenville Armitage (3), Stein Gjessing (1)

Affiliation.
(1) Department of Informatics, University of Oslo, Norway
{safiqui, michawe, kristah, steing, oystedal}@ifi.uio.no

(2) Simula Research Laboratory, Oslo, Norway
davidh@simula.no

(3) Swinburne University of Technology, Melbourne, Australia
garmitage@swin.edu.au

Publication. Under submission.

Abstract. We present ctrlTCP, a method to combine the congestion controls of multiple TCP connections between the same pair of hosts. To address the problem that these TCP connections may not share the same path, they can be encapsulated. This is sometimes already done, e.g. for VPNs. For other cases, we propose a method to encapsulate multiple TCP connections within a single UDP port pair without incurring header overhead. Different from the previously proposed Congestion Manager, our method is designed to minimize the changes to the existing TCP code, making it easier to implement. While we are not the first to propose such lighter-weight congestion coupling, prior approaches (E-TCP and EFCM) had
Single-Path TCP Congestion Control Coupling

deficiencies that are eliminated in ctrlTCP. Using ns-2 simulations and an implementation in the FreeBSD kernel, we show that our mechanism reduces both queuing delay and packet loss while enabling precise allocation of the share of the available bandwidth between the connections according to the needs of the applications.

9.1 Introduction

An increasing number of TCP flows overlap in time and share the same endpoints. There can be many reasons: a client-server application may deliberately establish parallel TCP connections to a server in an attempt to expedite content transfers, independent instances of an application may initiate connections to the same server that just happen to overlap in time, and so forth. Unfortunately, as the resulting overlapping TCP connections usually share a common network path, they can suboptimally interact (and indeed, compete) with each other and other flows. Combinations that appear to produce throughput gains will often drive significant additional queuing delays or packet losses, to the detriment of unrelated flows sharing any bottlenecks along the common path. In large part this is due to each overlapping TCP connection’s congestion control state machines acting independently.

Significant solutions to date fall into one of two classes: Merge common application-layer data streams above a single transport layer connection, or couple the transport layer congestion control machinery for connections known to share the same endpoints. Examples of the former include SPDY [1] and HTTP/2 [2], which multiplex multiple web sessions on top of a single TCP connection between client and server. Examples of the latter, often referred to as coupled congestion control, include the Congestion Manager (CM [3]), Ensemble TCP (E-TCP [4]) and Ensemble Flow Congestion Management (EFCM [5]).

However, there are known problems within each class of solutions. Simply multiplexing application flows onto a single TCP connection results in a head-of-line (HoL) blocking, where faster application-layer threads are forced to wait while serialized messages from slower threads are handled at the TCP destination. Solving HoL blocking usually involves entirely different transport protocols, such as QUIC [6] or SCTP [7]. On the other hand, previous coupled congestion control strategies tend not to fully leverage the statefulness of the TCP congestion control algorithm.

Our novel contribution in this paper is two-fold: We introduce ctrlTCP, a refined coupled congestion control strategy that better utilizes a TCP sender’s awareness of network conditions and allows precise intra-flow bandwidth sharing, and we couple this with a light-weight, dynamically configured TCP-in-UDP (TiU) encapsulation scheme that ensures our coupled flows do indeed share all bottlenecks along a single path. TiU is optional, as our coupled congestion control strategy is applicable wherever overlapping TCP flows must follow the same path (such as when routed
Using both ns-2 and FreeBSD implementations we have explored the benefits of our coupled congestion control scheme and TiU encapsulation. Our results demonstrate significantly better (lower) queuing delays and packet loss rates compared to uncoupled TCP connections. We believe our approach is practical and deployable in today’s Internet, offering lower RTTs to all traffic sharing bottlenecks with coupled TCP connections.

The rest of our paper is organized as follows: Section 9.2 introduces related work and describes our improved approach to coupled TCP congestion control. We discuss methods to ensure that coupled flows really traverse the same bottleneck in Section 9.4, and present simulations and experimental results in Section 9.3. The paper concludes in Section 9.5.

9.2 Coupled congestion control algorithm design

Despite a number of previous attempts at coupled congestion control, widespread deployment has proven to be difficult, mainly due to the complexities of the proposed mechanisms. We first briefly outline key prior work as motivation for our proposed mechanism.

9.2.1 Prior Work

To the best of our knowledge, RFC 2140 [8] is the first work to outline a mechanism for coupling TCP connections by sharing TCP Control Block (TCB) in order to better initialize new connections. This idea was expanded by Ensemble TCP (E-TCP) [4] to allow concurrent flows to benefit from each other beyond initialization, working together so that the aggregate is no more aggressive than a single TCP flow. On the other hand, Ensemble Flow Congestion Management (EFCM) [5] allows the aggregate to be collectively as aggressive as the combination of separately controlled TCP connections. The Congestion Manager (CM) [3, 9] takes the concept even further, completely replacing each flow’s congestion controller with its own congestion control mechanism—a rate based controller.

The CM is a major modification to the implementation of congestion control as part of TCP. Our proposal aims to minimize changes to the kernel TCP code, making it much easier to implement as an add-on, closer in spirit to E-TCP and EFCM, but fixing problems that we find with these mechanisms. We adopt a method that we earlier used to couple congestion control for media flows in WebRTC [10, 11]. However, TCP has particular difficulties due to its stateful nature, requiring a significantly different design approach for ctrlTCP.

Congestion control coupling as described here shares some similarities, yet also has important differences, when compared to coupled congestion control for Multi-
Single-Path TCP Congestion Control Coupling

Path TCP (MPTCP), e.g. the mechanisms LIA [12], OLIA [13] and BALIA [14]. Similar to our proposal, E-TCP and the CM, these mechanisms try to behave like one flow across a single bottleneck. However, MPTCP’s coupling assumes that flows should take a different path, and ideally also traverse different bottlenecks. This means that some things that single-path coupling can do would probably be quite inappropriate for MPTCP. For example with single-path coupling:

- A new connection joining the aggregate can immediately get a share of the potentially quite large cwnd of ongoing transfers. This can significantly reduce the completion time of short flows, but in MPTCP, it results in using a very large initial window on a new path.

- Changing states (e.g. avoiding slow start), as our algorithm does, strongly relies on the assumption that there is only a single shared bottleneck. The resulting behaviour is quite wrong in case of multiple bottlenecks.

- A share of the aggregate $cwnd$ can be assigned based on application preference in the case of single-path coupling. These preferences can even change on the fly. Again, if connections would traverse different paths, the outcome on the wire could temporarily be much more aggressive than TCP.

Finally, MPTCP’s subflows also use different tuples in order to be able to use different paths – this is, for example, leveraged in [15] without even using multi-homed end-systems. This is the opposite of what we are trying to achieve with the UDP encapsulation discussed in Section 9.4.

9.2.2 Basic algorithm logic

In ctrlTCP, each TCP session communicates with an entity called a Coupled Congestion Controller (CCC). The CCC couples flows traversing the same path to the same destination. Fig. 9.1 provides an overview of this communication. New TCP sessions first register with the CCC, supplying it with a (i) connection identifier (cid), (ii) Priority (P), (iii) cwnd, and (iv) ssthresh. Where necessary a new coupled group is created and all summation values are initialized; this includes setting common variables such as the sum of all congestion windows, $\text{sum}_c\text{wnd}$. Then the new flow obtains its first cwnd and ssthresh values. Algorithm 5 describes the registering process. Table 9.1 defines the variables used in Algorithms 5 to 7.

Once flows are coupled in a group, they update their status each time they change cwnd. The update message includes cid, cwnd, ssthresh, and the TCP state machine state. The CCC sends a response assigning the updating connection calculated values of cwnd and ssthresh. The update algorithm, summarized in Algorithm 7, emulates the behavior of a single TCP session by choosing one session as the Coordinating Connection (CoCo), initially the first flow. This session dictates
**Coupled congestion control algorithm design**

**Figure 9.1:** Message sequence chart of communication between a TCP session and the Coupled Congestion Control (CCC) entity. *Response is not sent if the session is in Fast Recovery (FR)*
Single-Path TCP Congestion Control Coupling

Algorithm 5 CCC – connection registering

1: **Input:** cid, cwnd, ssthresh, P
2: **Output:** ccc_cwnd(cid), ccc_ssthresh(cid)
3: if first connection in new coupled group then
4:     ccc_cwnd(cid) ← cwnd
5:     sum_P ← 0
6:     sum_cwnd ← 0
7:     sum_ssthresh ← 0
8: end if
9: ccc_P(cid) ← P
10: sum_P ← sum_P + P
11: sum_cwnd ← sum_cwnd + cwnd
12: ccc_cwnd(cid) ← P × sum_cwnd / sum_P
13: ccc_ssthresh(cid) ← ssthresh
14: if sum_ssthresh > 0 then
15:     ccc_ssthresh(cid) ← P × sum_ssthresh / sum_P
16: end if
17: Send ccc_cwnd(cid) and ccc_ssthresh(cid) to cid

the increase / decrease behavior for the aggregate. For simplicity, the algorithm refrains from adjusting cwnd when a connection is in Fast Recovery (FR). As we will explain in the following, we limit the usage of Slow Start (SS), ensuring that the aggregate’s behavior is only dictated by SS when all connections are in the SS phase. A detailed description of Algorithm 7 is provided in the appendix.

When a TCP session terminates (see Algorithm 6) its variables are removed and the summations are recalculated. The common variables also need to be adjusted when a flow leaves, as shown in Algorithm 6. If it is the last connection in the coupled group, the group is removed from the CCC.

Our algorithm shares state variables across multiple TCP connections. This has been done in the past by E-TCP [4] and EFCM [5]; however, these works do not account for the stateful nature of TCP, leading to erroneous behavior if, as with E-TCP and our algorithm, the goal is to emulate the dynamics of a single TCP connection. In the following, we use these two schemes to explain the design rationale underlying our algorithm, and we accordingly contrast our work against them in Section 9.3.
Algorithm 6 CCC – connection leaving

1: **Input:** cid
2: if last connection then
3: Remove coupled group
4: else
5: if cid = CoCo then
6: Coco ← the next connection
7: end if
8: sum_P ← sum_P - ccc_P(cid)
9: Remove ccc_P(cid), ccc_cwnd(cid), ccc_ssthresh(cid)
10: end if

9.2.3 Loss events

Fast recovery behaviour

The algorithm only decides when a flow should enter Fast Recovery (FR); while more elaborate strategies are possible, incorporating the FR phase itself in the algorithm would have made it significantly more complex and did not seem necessary. Thus, once a flow is in FR, we refrain from updating the $cwnd$ and $ssthresh$ values.

TCP operates on loss events (one or more packet losses per RTT), not individual packet losses. When congestion controls are combined, this logic should be preserved. We explain this by using an example of two connections traversing the same bottleneck. A single packet drop from connection 1, two drops from connections 1 and 2 or multiple packet drops from connection 2 only, should all result in the same behavior of the traffic aggregate. Simply sharing TCP variables such as $cwnd$ or $ssthresh$ cannot achieve this.

We simulated the behaviour of two TCP Reno connections in the ns-2 simulator\(^1\) using a dumbbell topology (bottleneck capacity 10 Mbps, RTT 100 ms, packet size 1500 bytes, and queue length of one BDP (83 packets)), in order to compare our mechanism with EFCM coupling\(^2\) and without coupling. Fig. 9.2 captures the behavior of two TCP Reno flows, without coupling, with EFCM, and with our proposed coupled congestion control mechanism, respectively. We artificially induced a packet drop for connection 1 at 25 seconds. Fig. 9.2(a) shows that without coupling, it takes quite some time to converge, and with EFCM (see Fig. 9.2(b)), the aggregate is not halved, and the flows remain aggressive. This can lead to higher queue growth and packet losses.

---

\(^1\)We used the TCP-Linux module that gives the flexibility to use the actual Linux TCP code in simulations. In our case, it was Linux kernel 3.17.4

\(^2\)We implemented EFCM and E-TCP in ns-2 based on the description from [5] and [4].
Single-Path TCP Congestion Control Coupling

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cid</td>
<td>Connection identifier</td>
</tr>
<tr>
<td>cwnd</td>
<td>TCP session congestion window value</td>
</tr>
<tr>
<td>ssthresh</td>
<td>TCP session slow start threshold value</td>
</tr>
<tr>
<td>P</td>
<td>Priority weight of a TCP flow</td>
</tr>
<tr>
<td>ccc_var</td>
<td>Coupled congestion controller calculated version of a TCP session variable</td>
</tr>
<tr>
<td>sum_cwnd</td>
<td>Sum of all the cwnd values in the group</td>
</tr>
<tr>
<td>sum_ssthresh</td>
<td>Sum of all the ssthresh values in the group</td>
</tr>
<tr>
<td>sum_P</td>
<td>Sum of all the priority values in the group</td>
</tr>
<tr>
<td>state</td>
<td>TCP state machine state: Slow Start (SS), or Congestion Avoidance (CA), or Fast Recovery (FR)</td>
</tr>
<tr>
<td>ssbits</td>
<td>Bit array with a bit for each connection in the group. Set if connection state=SS</td>
</tr>
<tr>
<td>CoCo</td>
<td>Coordinating Connection</td>
</tr>
</tbody>
</table>

Table 9.1: Names of variables used in Algorithms 5 to 7

Timeouts and Slow Start

The algorithm operates on the principle that slow start following a timeout should only happen if no packets could be delivered across the same path for the timeout interval. This assumption is broken if some but not all connections experience a timeout. In practice this means that if at least one of the flows is still receiving acknowledgements and has not tried to enter slow start, then it is not appropriate for the coupled group to enter slow start. The variable ssbits is used to facilitate this decision (see Algorithm 7).

E-TCP does not adopt this rationale, instead forcing all flows to slow start if any one has a timeout (see Fig. 9.2(c) where we artificially induced a timeout for connection 1 to simulate this behavior). EFCM has the issue of sharing the initial very large ssthresh value, which potentially moves all existing flows back into Slow Start from Congestion Avoidance when a new flow joins.

With our algorithm, if the timeout of a particular flow is indeed due to lost packets, it will recover the lost packets when it enters Fast Recovery. In the extreme case when a flow has lost its entire cwnd of packets, it can take up to $1 + \text{Dupthresh}^3$ RTTs for the flow to enter fast recovery. The reason for this is that the flow’s duplicate ack counter is not being incrementated for ACKs received on other coupled flows, as would be the case if it was truly one aggregated TCP flow. A possible

---

$^{3}$Number of duplicate ACKs required to trigger fast recovery
solution to this — currently under investigation — is to count ACKs from other coupled flows and send this DupACK count when the affected flow sends its update. This will allow fast recovery to be triggered after just 1 RTT and still maintains the simplicity of implementation and interaction with the coupled flows that our algorithm has.

9.2.4 ACK-clocking

When a new connection joins, it may benefit from sharing the large cwnd of existing flows. This has potential to cause bursts of transmission into the network, unless the packets are paced in some way. Consider the scenario where connection 1 has already achieved a cwnd of 100 packets, connection 2 joins and receives cwnd=50 packets as its share of the capacity. If it sends these without some form of pacing it can cause a significant congestion spike in the network. It is not a problem for connection 1 alone because its packets are paced by arriving ACKs.

Our CCC algorithm uses a simple ack-clocking mechanism to avoid these bursts.
Single-Path TCP Congestion Control Coupling

Figure 9.3: Coupling of 2 flows when flow 2 joins after 5 seconds. Packet sequence plots and cwnd plots are shown with and without the use of ack-clocking mechanism.

Rather than using timers, we utilize the acknowledgements connection 1 receives to pace the sending of connection 2 over the course of the first RTT. In this way, we avoid causing a congestion spike in the network.

Fig. 9.3 shows the packet sequence diagrams and cwnd plots over time of two coupled-TCP Reno connections, with and without ack-clocking. Without ack-clocking, the congestion spike causes significant packet loss. Our ack clocking algorithm completely eliminates this issue.
9.3 Results

We have implemented our mechanism in the ns-2 simulator and in the FreeBSD 11 kernel. Fig. 9.4 shows the experimental setup for both the ns-2 and emulation experiments. For ns-2 we used TMIX$^4$ produced background traffic in the ns-2 simulator. We used a pre-processed version of TMIX traffic in order to provide an approximate load of 50% on a 10 Mbps bottleneck link. The RTTs of background TCP flows generated by TMIX are in the range of 80 - 100 ms. For the emulation experiments the sender and receiver machines are physical, identical desktop computers (Intel i7-870 2.93GHz CPU, 8GB RAM) equipped with Gigabit Ethernet Network Interface Cards (NICs), running our modified version of FreeBSD-11. They are connected via a third identical machine running Ubuntu Linux 15.04 and the CORE network emulator [18] version 4.7 to form the dumbbell topology in Fig. 9.4 (bottleneck capacity 10 Mbps, RTT 100 ms, MTU 1500 bytes, and queue length 1 BDP (83 packets)). The underlying TCP congestion control mechanism is NewReno.

In order to compare our proposed coupling mechanism with E-TCP and EFCM, we simulated the behavior of two TCP Reno connections by varying the number of timeouts. We achieved this by dropping a series of packets (per provoked timeout) from only one of the flows. Fig. 9.5 illustrates the aggregate throughput of two TCP Reno connections, with EFCM, E-TCP, and $ctrlTCP$, respectively. It can be seen from the graph that the aggregate throughput of two connections coupled with EFCM is more than $ctrlTCP$ and E-TCP because it is more aggressive than one flow. However, the aggregate throughput of two connections coupled with $ctrlTCP$

---

$^4$TMIX traffic used in ns-2 simulations is taken from 60-minute trace of campus traffic at the University of North Carolina [16] traffic in ns-2 simulations, and it can be accessed from the common TCP evaluation suite [17].
is more than E-TCP because our mechanism does not force all flows to enter slow start if one has a timeout. Because this result, together with the findings in Fig. 9.2, shows that ctrlTCP successfully eliminates the deficiencies of EFCM and E-TCP, we do not consider these two algorithms further.

Fig. 9.6 showcases an example of ctrlTCP where two connections are created with priorities 0.75 and 0.25, respectively. Connection 1 starts at t=3 sec and connection 2 starts at t=30 sec. It can be seen from Fig. 9.6 that between t=30 sec and t=85 sec (the duration when two flows coexist), flow 1 gets 3/4 of the aggregate cwnd while flow 2 gets 1/4. It is also observable that the two flows are coupled, as they increase and decrease their cwnds at the same time. Outside of this interval, flow 1 gets all of the cwnd.

In Fig. 9.6, flow 2 is able to immediately use its share of the aggregate cwnd. Fig. 9.7 demonstrates that this can yield a significant improvement in the short flow’s completion time. Here, we varied the capacity from 1 Mbps to 10 Mbps. We repeated this test 10 times with randomly picked flow start times over the first second for the long flow (25 Mb) and the sixth second for the short flow (200 Kb). It can be seen from Fig. 9.7 that ctrlTCP with ack-clocking further improves the short flow’s FCT. Coupling also removes a synchronization effect: in the uncoupled 2 Mbps scenario, the short flow was even faster than in the 3 Mbps scenario because it was the first to send its initial window (IW) into the queue, which did
Figure 9.6: cwnd (in Kbytes) plot of two TCP connections using coupled congestion control with priorities compared to a single TCP flow scenario. The aggregate line depicts the sum of cwnds in two connections scenario. cwnd going down all the way to 0 whenever cwnd is reduced is not related to ctrlTCP but a result of how cwnd is internally updated in FreeBSD.
not have enough space for the IWs of two flows. We also confirm the reduction of FCTs in emulation while varying the capacity to 6, 8, and 10 Mbps, respectively (see Fig. 9.8). In both the simulation and emulation experiments, the impact on the long flow was negligible.

In the following tests, experiments were repeated 10 times with different randomly picked flow start times over the first second. Prerecorded traces of self-similar crosstraffic were injected to occupy 50% of the bottleneck link capacity on average. These were generated using the D-ITG [19] traffic generator, by superposition of 11 on/off streams, with Pareto heavy-tailed on-time distributions ($H = 0.8$) and exponentially distributed off-times ($\mu \in [1, 2]\text{s}$). Packet sizes are normally distributed ($\mu = 1000, \sigma = 200$), with exponentially distributed ($\mu \in [50, 150]\text{pps}$) inter-packet times.

Fig. 9.9 and 9.11 illustrate the average RTT, and loss ratio, with and without coupling, respectively. It can be seen from the graphs that our mechanism reduces both the average RTT and loss without significantly affecting goodput as we varied the number of flows (see Fig. 9.10).

Fig. 9.12 demonstrates that our mechanism can distribute the share of the aggregate cwnd to the TCP connections based on the needs of the applications. Both
Figure 9.8: Flow completion time (FCT) of short flows without ack-clocking (emulation)

the simulation and emulation results confirm that our mechanism calculates and as-
igns the shares as we vary the priority ratio between two TCP Reno connections, close to the ideal behaviour.

In order to see if our mechanism could be run as a stand-alone application to control different senders not running on the same OS instance, we have also implemented our ctrlTCP in the FreeBSD 11 kernel with state shared across the freely available VirtualBox\textsuperscript{5} hypervisor. In this way, it could be used to prevent an unfair sender from obtaining a larger share of the capacity by opening multiple TCP connections. Fig. 9.13 shows Jain’s Fairness Index [20] \( \left( \sum_{i=1}^{N} x_i(t) / N \sum_{i=1}^{N} x_i(t)^2 \right) \) for \( N = 2 \) aggregate flows \( x_1 \) and \( x_2 \), calculated using the traffic that originated from two VMs across a 10 Mbit/s×100 ms bottleneck with and without the coupled congestion control algorithm in Section 9.2.

9.4 Encapsulation

Our algorithm, as well as EFCM, E-TCP and the CM assume that multiple TCP connections sending to the same destination would traverse the same bottleneck. This is not always true – load-balancing mechanisms such as Link Aggregation

\textsuperscript{5}https://www.virtualbox.org
Figure 9.9: Average delay (in milliseconds) as the number of TCP connections is varied, with and without coupled congestion control (emulation)

Figure 9.10: Average goodput as the number of TCP connections is varied, with and without coupled congestion control (emulation)
Figure 9.11: Loss ratio as the number of TCP connections is varied, with and without coupled congestion control (emulation)

Figure 9.12: Throughput ratio as the priorities of two TCP connections are varied
Group (LAG) and Equal-Cost Multi-Path (ECMP) may force them to take different paths [21]. If this leads to the connections seeing different bottlenecks, combining the congestion controllers would incur wrong behavior (as discussed in Section 9.2.1). There are, however, several application scenarios where the single-bottleneck assumption is correct:

- Whenever all traffic between the two hosts is already tunneled, e.g. for VPNs. There are many possible encapsulation schemes for various use cases. For example, Generic UDP Encapsulation (GUE) [22] — a method already deployed in Linux — allows us to multiplex several TCP connections onto a same UDP port number pair. Several encapsulation methods transmit layer-2 frames over an IP network — e.g. VXLAN [23] (over UDP/IP) and NvGRE [24] (over GRE/IP). Because Layer-2 networks should be agnostic to the transport connections running over them, the path should not depend on the TCP port number pair and our algorithm should work. Some care must still be taken: for example, for NvGRE, [24] says: “If ECMP is used, it is RECOMMENDED that the ECMP hash is calculated either using the outer IP frame fields and entire Key field (32 bits) or the inner IP and transport frame fields”. If routers do use the inner transport frame fields (typically, port numbers) for this hashing, we have the same problem even over NvGRE.

- When IPv6 is available, the TCP connections could be assigned the same IPv6 flow label. According to RFC 6437 [25], “The usage of the 3-tuple of the Flow Label, Source Address, and Destination Address fields enables efficient IPv6 flow classification, where only IPv6 main header fields in fixed positions are used” – this would be favorable for TCP congestion control coupling. However, this RFC does not end up making a clear recommendation about either using
Encapsulation

the 3-tuple or 5-tuple (which includes the port numbers) – it seems that both methods are valid. Thus, whether it works to use the flow label as the sole means to put connections on the same path depends on router configuration. When it works, it is an attractive option because it does not require changing the receiver.

Because TCP does not preserve message boundaries and the size of the TCP header can vary depending on the options that are used, it is also no problem to use methods that insert a header in between the TCP header and the UDP packet with a (e.g. SPUD [26]) without exceeding the known MTU limit. When creating a TCP segment, a TCP sender needs to consider the length of this header when calculating the segment size, just like it would consider the length of a TCP option. This assumes that the usage of other headers such as SPUD in-between the UDP header and the TiU header is known to both the sender-side and receiver-side code that processes TiU.

Measurements can infer whether flows traverse the same bottleneck. This even allows to combine flows from the same sender that are destined for different receivers. A recent Shared Bottleneck Detection (SBD) tool [27, 28] was successfully applied to MPTCP in [29]. Such a dynamic decision about shared bottlenecks requires that ctrlTCP can be enabled or disabled, which is easy due to its design but much harder with the Congestion Manager.

Whenever the network configuration is known.

We want to be able to ensure that TCP congestion control coupling can always work, provided that the required code is available at the receiver – and be able to efficiently fall back to the standard behaviour in case it is not. To achieve this, we have devised a method to encapsulate multiple TCP connections using the same UDP port pair. We have implemented this for both the sender and receiver in the FreeBSD kernel, as a simple add-on to the TCP implementation that is controlled via a socket option.

9.4.1 TCP-in-UDP (TiU)

We propose a TCP-in-UDP (TiU) encapsulation method, inspired by [30] and [31], where a UDP header is followed by a slightly altered TCP header. The UDP source ports and destination ports are semantically different than [30] and [31]: multiple TCP connections use the same well known UDP port in our case. To avoid increasing the MSS, we removed redundant fields, such as the checksum, the TCP source and destination ports, and the urgent pointer. However, to identify the connections we replaced reserved bits and urgent flags with a five-bit Connection ID field. The Connection ID is a way to uniquely identify a port number pair of a TCP
connection. TiU encapsulation can combine up to 32 TCP connections with one UDP port number pair. This only affects the on-the-wire behavior – port numbers are otherwise used as normal. The two hosts inform each other about the mapping using SYN options (we have plenty of option space in UDP-encapsulated TCP). To allow other encapsulated content, such as a STUN packet, we changed the order of fields to move the Data Offset field to the beginning of the UDP payload (as in [31]). The altered TCP header for TiU is shown in Fig. 9.14.

The TiU-TCP SYN and SYN/ACK packets are used to initially establish the mapping between the Connection ID and the port numbers of parallel TCP connections. The TiU-TCP SYN and SYN/ACK packets, as shown in Fig. 9.15, look slightly little different than “normal” packets (Fig. 9.14), because it requires four bytes to encapsulate the source port number and destination port number of the TCP connection. To create this header, we swap the position of the original TCP header’s port number fields with the position of the Data Offset / Reserved / Flags / Window fields.

Every TiU SYN or TiU SYN-ACK packet also carries at least the TiU-Setup TCP option, following the format defined in [32]. Fig. 9.16 shows the TiU-setup option field where it has Kind=253, Length=5, a fixed EXID value, and the Connection ID. We used an 8-bit field for Connection ID for easier parsing, however only 5 bits are used to represent 32 connections as shown in Fig. 9.14.

On a SYN packet, the sender uses the Connection ID to notify the recipient of the Connection ID value that it intends to use in future packets to represent the Encapsulated Source Port and Encapsulated Destination Port. On a SYN/ACK packet, the recipient confirms that it supports TiU encapsulation. If the recipient does not support TiU, both hosts fall back to regular non-encapsulated TCP connections.
Encapsulation

<table>
<thead>
<tr>
<th>Source Port</th>
<th>Destination Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>Checksum</td>
</tr>
<tr>
<td>Data Offset</td>
<td>Reserved</td>
</tr>
<tr>
<td>high 4 bits</td>
<td>Window Size</td>
</tr>
<tr>
<td></td>
<td>Sequence Number</td>
</tr>
<tr>
<td></td>
<td>Acknowledgement Number</td>
</tr>
<tr>
<td>Encapsulated Source Port</td>
<td>Encapsulated Destination Port</td>
</tr>
<tr>
<td>Options</td>
<td></td>
</tr>
</tbody>
</table>

Figure 9.15: Complete TCP-in-UDP SYN and SYN/ACK header

<table>
<thead>
<tr>
<th>Kind</th>
<th>Length</th>
<th>ExID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 9.16: TiU setup TCP option

9.4.2 Protocol operation

For every TCP connection, we configure the following two variables through a socket option:

1. TiU ENABLE — a boolean to enable/disable.

2. Priority — a value in the range from 1 to 10. This is being used by the coupled congestion control algorithm to assign an application-specified share of the aggregate’s cwnd to a connection.

Encapsulation and decapsulation

If TiU ENABLE is set, a TCP segment is encapsulated before it is transmitted. During the connection setup phase, TiU SYN and TiU SYN/ACK packets are encapsulated as shown in Fig. 9.15, and the TiU-Setup TCP option, shown in Fig. 9.16 is appended. A TiU sender includes the Connection ID in the TCP SYN packet, which is used for mapping the Encapsulated Source Port and Encapsulated Destination Port. Once the connection is established, TCP segments are encapsulated into the TiU header format (see Fig. 9.14) just before they are transmitted.

Upon receipt of a TiU packet (destined for a well-known port), the reverse mechanism is applied to decapsulate the packet. Both the TiU sender and receiver
remember the mapping of encapsulated port pair. An error message, with a special
value 255, is sent to the TiU sender on TiU SYN/ACK packet if the connection ID
is not free. This informs the sender to stop encapsulating further TCP segments.

Based on the aforementioned implementation details, a TiU sender operates as
follows, assuming a client connects to a server (listening to a well-known port), and
no prior connections have been established between them:

1. Through a socket option, an application requests a TiU operation. The kernel
append the TiU-Setup option to the TiU TCP SYN, which contains Encapsu-
lated Port numbers (source port A) and the connection ID Z.

2. Upon receiving a packet, the server translates the header back to normal
TCP format and gives the resulting TCP packet to regular TCP processing.
It associates future TiU packets with connection ID Z to port A. After that,
the server responds with TiU SYN-ACK.

3. Upon successful TiU handshaking, packets are encapsulated with a TiU header
before sending, and decapsulated upon receiving. Henceforth, both the server
and client use a local identifier to match encapsulated ports from TiU packets,
carrying Connection ID Z.

4. After this, if an application wants to open another connection to the same
destination host, it performs the above steps.

5. Now, upon a successful TiU handshake, the coupled congestion control mech-
anism is activated as there are two or more connections available.

We implemented a fall-back mechanism (see Fig. 9.17) to cater for unsuccessful
TiU handshakes, e.g. connection ID collisions. This allows to immediately switch
to “raw” TCP communication by making TCP SYN and SYN/ACK visible to the
middleboxes: It has been shown in [33] that, since middleboxes keep track of TCP
connections, it may be problematic for connections to fall back to raw TCP packets
without using a raw TCP SYN and SYN/ACK. The fall-back mechanism works as
follows, assuming a client tries to open a connection to a server:

- A client sends a TiU SYN packet followed by a raw TCP packet.
- The server responds with TiU SYN/ACK, and then sends out a raw TCP
  SYN/ACK packet (ideally, in that sequence).
- The server immediately stores the associated encapsulated port numbers and
  Connection ID after receiving a TiU SYN packet and responding with a TiU
  SYN/ACK packet (and raw TCP SYN/ACK packet). Upon successful TiU
  handshakes, it further ignores any additional incoming TCP SYN or TCP
SYN/ACK packets from the same host. Otherwise, it normally processes TCP SYN or TCP SYN/ACK packets.

### 9.5 Conclusions

It is increasingly common for standard TCP connections to overlap in time between the same network endpoints. The result is competition, rather than cooperation, between each connection’s congestion control mechanism, often leading to undesirable spikes in queuing delay and packet loss rates.

Our paper has made two novel contributions. First, we have introduced ctrlTCP, a new coupled congestion control strategy that allows applications to exert precise allocation over the relative bandwidth share offered to coupled flows, with only minimal interfacing to the kernel TCP code. Second, we have designed a new TCP-in-UDP (TiU) encapsulation scheme that establishes on-demand tunnels for up to 32 coupled TCP flows between any two UDP ports on the common endpoints. TiU tunnels ensure our coupled TCP flows always share the same path, regardless of any flow-aware load-balancing (or similar devices) that might otherwise cause individual TCP flows to take different paths between the same endpoints.

We have implemented ctrlTCP and TiU in both ns-2 and the FreeBSD kernel, and used these implementations to demonstrate the utility of our proposal: Our implementation of TiU also demonstrates a fall-back mechanism, allowing regular TCP connection establishment if TiU traffic is unexpectedly blocked or the remote server does not (yet) implement TiU. ctrlTCP yields lower queuing delays and packet loss rates than uncoupled TCP flows, and has negligible impact on aggregate end-to-end performance.
Coupling flows is especially beneficial when many short web-like flows share a common bottleneck as it allows the short flows to quickly obtain a share of the available capacity. However, when multiple bulk transmissions are aggregated, competing as one TCP flow may not achieve an appropriate share of the bottleneck capacity. As future work we plan to investigate dynamically tuning our mechanism as prior work has suggested (cf. MulTCP [34], MulTFRC [35]) to ensure that an appropriate capacity share is obtained across a range of use cases.

In prior work [11], we have successfully applied a method similar to ctrlTCP to various WebRTC RTP congestion controls, using the same minimally-invasive approach of letting each flow do its own rate calculation, and then interacting with a coupled congestion control instance. Early tests indicate that it even may be feasible to couple heterogeneous congestion controls with this homogeneous approach. This could, for example, help avoid known problems that arise when latency-sensitive delay-based mechanisms compete with loss-based mechanisms such as standard TCP. We also consider it feasible to adjust the ctrlTCP algorithm to operate with flows that have different RTTs; this would allow us to couple TCP flows that leave one sender, traverse a common bottleneck (e.g., a home user’s thin uplink) and arrive at different destinations. We intend to investigate these issues in our own future work – but we also expect that our approach creates new opportunities for experimentation (e.g., Fig. 9.13 has shown that senders do not even have to be running in the same OS instance). We therefore hope that this paper inspires others to build upon ctrlTCP.6

9.6 Acknowledgments

This work has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 644334 (NEAT). The views expressed are solely those of the authors.

References


6The source code of ctrlTCP is available at http://safiquli.at.ifi.uio.no/tcp-ccc/
REFERENCES


Single-Path TCP Congestion Control Coupling


**ctrlTCP - CCC Update Algorithm’s Description**

The *update* algorithm (Algorithm 7) uses variables defined in Table 9.1. We can differentiate between two major cases: the connection that invoked the algorithm is the CoCo (lines 15-42) or not (lines 4-14). If the flow currently being updated is in Fast Recovery (FR) and all other flows are in Congestion Avoidance (CA), this flow becomes the CoCo (which will, in turn, lead to a *ssthresh* reduction further down, in line 28). This check for other flows being in CA avoids double reactions: because we assume a common bottleneck and a behavior like one flow, we want to prevent that e.g. three flows would consecutively enter FR and each lead to an aggregate rate reduction. Instead, we allow one reduction only for the full duration of one flow’s FR phase.

If the flow is not the CoCo and it is in CA or Slow Start (SS) state, its *cwnd* and *ssthresh* values are assigned a weighted share of the aggregate (lines 8-9). The *ssbits* are used to keep track of which flows are in SS. For flows that are not the CoCo, they are continuously updated.

If the flow is the CoCo, the algorithm checks whether *cwnd* has increased since the last time the algorithm was invoked (line 17). If so, then it increases the aggregate *cwnd* (*sum_cwnd*) by the amount by which the CoCo has increased its *cwnd*, else it decreases *sum_cwnd* in proportion to the reduction of the flow’s *cwnd* (i.e., for normal TCP, the aggregate *cwnd* gets halved). Then, in lines 22-25, the *cwnd* and *ssthresh* values of the CoCo itself are updated. The values are weighted based on the priority of the flow. The *ssthresh* of the flow group is used if this value has been set. Otherwise, the *ssthresh* value is the one determined by the default CC algorithm.

The *ssbits* entry of the CoCo is only set if the flow experiences a timeout. This is done to avoid reducing the aggregate *ssthresh* sum further in subsequent updates while the CoCo is still in the SS state. If all flows are in SS, the aggregate *ssthresh* sum is reduced, and the *ssbits* are reset. Otherwise, a flow that is not in SS is elected as the new CoCo.

Finally, the *cwnd* and *ssthresh* values computed in the *update* function are applied to the flow in line 44. This is only done for flows not in FR to avoid additional complexity related to handling updated values in the FR phase.
**Algorithm 7** CCC - connection update

1. **Input:** cid, cwnd, ssthresh, state  
2. **Output:** ccc_cwnd(cid), ccc_ssthresh(cid)  
3. ccc_state(cid) ← state  
4. if CoCo ≠ cid then  
5. if state=FR ∧ ccc_state(x) = CA ∀x ≠ cid then  
6. CoCo ← cid  
7. else if state=CA ∨ state=SS then  
8. \[ ccc\_cwnd(cid) ← ccc\_P(cid) \times \text{sum}_cwnd / \text{sum}_P \]  
9. \[ ccc\_ssthresh(cid) ← ccc\_P(cid) \times \text{sum}_ssthresh / \text{sum}_P \]  
10. **end if**  
11. **end if**  
12. if ssbits(cid) = 1 ∧ state ≠ SS then  
13. ssbits(cid) ← 0  
14. **end if**  
15. if CoCo = cid then  
16. if state = CA ∧ ssbits(cid) = 0 then  
17. if cwnd ≥ ccc\_cwnd(cid) then \[ \triangleright \text{increased cwnd} \]  
18. \[ \text{sum}_cwnd ← \text{sum}_cwnd + \text{cwnd-ccc\_cwnd(cid)} \]  
19. else  
20. \[ \text{sum}_cwnd ← \text{sum}_cwnd + \text{cwnd / ccc}_cwnd(cid) \]  
21. **end if**  
22. \[ ccc\_cwnd(cid) ← ccc\_P(cid) \times \text{sum}_cwnd / \text{sum}_P \]  
23. \[ ccc\_ssthresh(cid) ← \text{ssthresh} \]  
24. if sum_ssthresh > 0 then  
25. \[ ccc\_ssthresh(cid) ← ccc\_P(cid) \times \text{sum}_ssthresh / \text{sum}_P \]  
26. **end if**  
27. else if state = FR then  
28. \[ \text{sum}_ssthresh ← \text{sum}_cwnd/2 \]  
29. else if state = SS then  
30. if cid experienced a timeout then  
31. ssbits(cid) ← 1  
32. **end if**  
33. if ssbits(x) = 1 \∀x then  
34. ssbits(x) = 0 \∀x  
35. \[ \text{sum}_cwnd ← \text{sum}_cwnd + \text{cwnd / ccc\_cwnd(cid)} \]  
36. \[ ccc\_cwnd(cid) ← ccc\_P(cid) \times \text{sum}_cwnd / \text{sum}_P \]  
37. \[ \text{sum}_ssthresh ← \text{sum}_cwnd/2 \]  
38. else  
39. CoCo ← first connection where ccc_state≠SS  
40. **end if**  
41. **end if**  
42. **end if**  
43. if state ≠ FR then  
44. Send ccc\_cwnd(cid) and ccc\_ssthresh(cid) to cid  
45. **end if**
Part III

Internet Drafts
Chapter 10

Coupled Congestion Control for RTP Media
Abstract

When multiple congestion controlled RTP sessions traverse the same network bottleneck, combining their controls can improve the total on-the-wire behavior in terms of delay, loss and fairness. This document describes such a method for flows that have the same sender, in a way that is as flexible and simple as possible while minimizing the amount of changes needed to existing RTP applications. It specifies how to apply the method for the NADA congestion control algorithm, and provides suggestions on how to apply it to other congestion control algorithms.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on June 10, 2017.

Copyright Notice

Copyright (c) 2016 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents.
carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction .................................................. 3
2. Definitions .................................................. 3
3. Limitations .................................................... 4
4. Architectural overview ........................................ 4
5. Roles .......................................................... 6
   5.1. SBD .................................................... 6
   5.2. FSE .................................................... 7
   5.3. Flows ................................................... 8
      5.3.1. Example algorithm 1 – Active FSE ................... 8
      5.3.2. Example algorithm 2 – Conservative Active FSE ...... 9
6. Application ..................................................... 10
   6.1. NADA .................................................... 11
   6.2. General recommendations .................................. 11
7. Expected feedback from experiments .......................... 12
8. Acknowledgements .............................................. 12
9. IANA Considerations ........................................... 12
10. Security Considerations ...................................... 12
11. References ................................................... 13
   11.1. Normative References ................................... 13
   11.2. Informative References ................................ 13
Appendix A. Application to GCC .................................. 15
Appendix B. Scheduling ............................................. 15
Appendix C. Example algorithm – Passive FSE .................. 15
   C.1. Example operation (passive) ............................... 18
Appendix D. Change log ............................................ 22
   D.1. draft-welzl-rmcat-coupled-cc ............................. 22
      D.1.1. Changes from -00 to -01 ................................ 22
      D.1.2. Changes from -01 to -02 ................................ 22
      D.1.3. Changes from -02 to -03 ................................ 23
      D.1.4. Changes from -03 to -04 ................................ 23
      D.1.5. Changes from -04 to -05 ................................ 23
   D.2. draft-ietf-rmcat-coupled-cc ............................... 23
      D.2.1. Changes from draft-welzl-rmcat-coupled-cc-05 ........ 23
      D.2.2. Changes from -00 to -01 ................................ 23
      D.2.3. Changes from -01 to -02 ................................ 23
      D.2.4. Changes from -02 to -03 ................................ 24
      D.2.5. Changes from -03 to -04 ................................ 24
      D.2.6. Changes from -04 to -05 ................................ 24
Authors’ Addresses ................................................ 24
1. Introduction

When there is enough data to send, a congestion controller must increase its sending rate until the path’s capacity has been reached; depending on the controller, sometimes the rate is increased further, until packets are ECN-marked or dropped. This process inevitably creates undesirable queuing delay when multiple congestion controlled connections traverse the same network bottleneck.

The Congestion Manager (CM) [RFC3124] couples flows by providing a single congestion controller. It is hard to implement because it requires an additional congestion controller and removes all per-connection congestion control functionality, which is quite a significant change to existing RTP based applications. This document presents a method to combine the behavior of congestion control mechanisms that is easier to implement than the Congestion Manager [RFC3124] and also requires less significant changes to existing RTP based applications. It attempts to roughly approximate the CM behavior by sharing information between existing congestion controllers. It is able to honor user-specified priorities, which is required by rtcweb [RFC7478].

2. Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

Available Bandwidth:
The available bandwidth is the nominal link capacity minus the amount of traffic that traversed the link during a certain time interval, divided by that time interval.

Bottleneck:
The first link with the smallest available bandwidth along the path between a sender and receiver.

Flow:
A flow is the entity that congestion control is operating on. It could, for example, be a transport layer connection, an RTP session, or a subsession that is multiplexed onto a single RTP session together with other subsessions.

Flow Group Identifier (FGI):
A unique identifier for each subset of flows that is limited by a common bottleneck.
Flow State Exchange (FSE):
   The entity that maintains information that is exchanged between flows.

Flow Group (FG):
   A group of flows having the same FGI.

Shared Bottleneck Detection (SBD):
   The entity that determines which flows traverse the same bottleneck in the network, or the process of doing so.

3. Limitations

   Sender-side only:
   Coupled congestion control as described here only operates inside a single host on the sender side. This is because, irrespective of where the major decisions for congestion control are taken, the sender of a flow needs to eventually decide on the transmission rate. Additionally, the necessary information about how much data an application can currently send on a flow is often only available at the sender side, making the sender an obvious choice for placement of the elements and mechanisms described here.

   Shared bottlenecks do not change quickly:
   As per the definition above, a bottleneck depends on cross traffic, and since such traffic can heavily fluctuate, bottlenecks can change at a high frequency (e.g., there can be oscillation between two or more links). This means that, when flows are partially routed along different paths, they may quickly change between sharing and not sharing a bottleneck. For simplicity, here it is assumed that a shared bottleneck is valid for a time interval that is significantly longer than the interval at which congestion controllers operate. Note that, for the only SBD mechanism defined in this document (multiplexing on the same five-tuple), the notion of a shared bottleneck stays correct even in the presence of fast traffic fluctuations: since all flows that are assumed to share a bottleneck are routed in the same way, if the bottleneck changes, it will still be shared.

4. Architectural overview

   Figure 1 shows the elements of the architecture for coupled congestion control: the Flow State Exchange (FSE), Shared Bottleneck Detection (SBD) and Flows. The FSE is a storage element that can be
implemented in two ways: active and passive. In the active version, it initiates communication with flows and SBD. However, in the passive version, it does not actively initiate communication with flows and SBD; its only active role is internal state maintenance (e.g., an implementation could use soft state to remove a flow's data after long periods of inactivity). Every time a flow's congestion control mechanism would normally update its sending rate, the flow instead updates information in the FSE and performs a query on the FSE, leading to a sending rate that can be different from what the congestion controller originally determined. Using information about/from the currently active flows, SBD updates the FSE with the correct Flow State Identifiers (FSIs). This document describes both active and passive versions, however the passive version is put into the appendix as it is extremely experimental. Figure 2 shows the interaction between flows and the FSE, using the variable names defined in Section 5.2.

Figure 1: Coupled congestion control architecture

```
Flow#1(cc) --------------- FSE --------------- Flow#2(cc)
#1 JOIN ----register--> REGISTER
REGISTER <--register-- JOIN #1
#2 CC_R ----UPDATE-----> UPDATE (in)
#3 NEW RATE <---FSE_R------- UPDATE (out) --FSE_R----> #3 NEW RATE
```

Figure 2: Flow-FSE interaction

Since everything shown in Figure 1 is assumed to operate on a single host (the sender) only, this document only describes aspects that
have an influence on the resulting on-the-wire behavior. It does, for instance, not define how many bits must be used to represent FSIs, or in which way the entities communicate. Implementations can take various forms: for instance, all the elements in the figure could be implemented within a single application, thereby operating on flows generated by that application only. Another alternative could be to implement both the FSE and SBD together in a separate process which different applications communicate with via some form of Inter-Process Communication (IPC). Such an implementation would extend the scope to flows generated by multiple applications. The FSE and SBD could also be included in the Operating System kernel.

5. Roles

This section gives an overview of the roles of the elements of coupled congestion control, and provides an example of how coupled congestion control can operate.

5.1. SBD

SBD uses knowledge about the flows to determine which flows belong in the same Flow Group (FG), and assigns FGIs accordingly. This knowledge can be derived in three basic ways:

1. From multiplexing: it can be based on the simple assumption that packets sharing the same five-tuple (IP source and destination address, protocol, and transport layer port number pair) and having the same Differentiated Services Code Point (DSCP) in the IP header are typically treated in the same way along the path. The latter method is the only one specified in this document: SBD MAY consider all flows that use the same five-tuple and DSCP to belong to the same FG. This classification applies to certain tunnels, or RTP flows that are multiplexed over one transport (cf. [transport-multiplex]). Such multiplexing is also a recommended usage of RTP in rtcweb [rtcweb-rtp-usage].

2. Via configuration: e.g. by assuming that a common wireless uplink is also a shared bottleneck.

3. From measurements: e.g. by considering correlations among measured delay and loss as an indication of a shared bottleneck.

The methods above have some essential trade-offs: e.g., multiplexing is a completely reliable measure, however it is limited in scope to two end points (i.e., it cannot be applied to couple congestion controllers of one sender talking to multiple receivers). A measurement-based SBD mechanism is described in [I-D.ietf-rmcat-sbd].
Measurements can never be 100% reliable, in particular because they are based on the past but applying coupled congestion control means to make an assumption about the future; it is therefore recommended to implement cautionary measures, e.g. by disabling coupled congestion control if enabling it causes a significant increase in delay and/or packet loss. Measurements also take time, which entails a certain delay for turning on coupling (refer to [I-D.ietf-rmcat-sbd] for details). Using system configuration to decide about shared bottlenecks can be more efficient (faster to obtain) than using measurements, but it relies on assumptions about the network environment.

5.2. FSE

The FSE contains a list of all flows that have registered with it. For each flow, it stores the following:

- a unique flow number to identify the flow
- the FGI of the FG that it belongs to (based on the definitions in this document, a flow has only one bottleneck, and can therefore be in only one FG)
- a priority $P$, which here is assumed to be represented as a floating point number in the range from 0.1 (unimportant) to 1 (very important).
- The rate used by the flow in bits per second, $FSE_R$.

Note that the priority does not need to be a floating point value and its value range does not matter for this algorithm: the algorithm works with a flow’s priority portion of the sum of all priority values. Priorities can therefore be mapped to the "very-low", "low", "medium" or "high" priority levels described in [I-D.ietf-rtcweb-transports] using the values 1, 2, 4 and 8, respectively.

In the FSE, each FG contains one static variable $S\_CR$ which is the sum of the calculated rates of all flows in the same FG. This value is used to calculate the sending rate.

The information listed here is enough to implement the sample flow algorithm given below. FSE implementations could easily be extended to store, e.g., a flow’s current sending rate for statistics gathering or future potential optimizations.
5.3. Flows

Flows register themselves with SBD and FSE when they start, deregister from the FSE when they stop, and carry out an UPDATE function call every time their congestion controller calculates a new sending rate. Via UPDATE, they provide the newly calculated rate and optionally (if the algorithm supports it) the desired rate. The desired rate is less than the calculated rate in case of application-limited flows; otherwise, it is the same as the calculated rate.

Below, two example algorithms are described. While other algorithms could be used instead, the same algorithm must be applied to all flows. Names of variables used in the algorithms are explained below.

- **CC_R** - The rate received from a flow’s congestion controller when it calls UPDATE.

- **FSE_R** - The rate calculated by the FSE for a flow.

- **S_CR** - The sum of the calculated rates of all flows in the same FG; this value is used to calculate the sending rate.

- **FG** - A group of flows having the same FGI, and hence sharing the same bottleneck.

- **P** - The priority of a flow which is received from the flow’s congestion controller; the FSE uses this variable for calculating FSE R.

- **S_P** - The sum of all the priorities.

5.3.1. Example algorithm 1 - Active FSE

This algorithm was designed to be the simplest possible method to assign rates according to the priorities of flows. Simulations results in [fse] indicate that it does however not significantly reduce queuing delay and packet loss.

(1) When a flow f starts, it registers itself with SBD and the FSE. FSE_R is initialized with the congestion controller’s initial rate. SBD will assign the correct FGI. When a flow is assigned an FGI, it adds its FSE_R to S_CR.

(2) When a flow f stops or pauses, its entry is removed from the list.
(3) Every time the congestion controller of the flow \( f \) determines a new sending rate \( CC_R \), the flow calls \( UPDATE \), which carries out the tasks listed below to derive the new sending rates for all the flows in the FG. A flow's \( UPDATE \) function uses a local (i.e. per-flow) temporary variable \( S_P \), which is the sum of all the priorities.

(a) It updates \( S_CR \).

\[
S_CR = S_CR + CC_R - FSE_R(f)
\]

(b) It calculates the sum of all the priorities, \( S_P \).

\[
S_P = 0
\]

\[
for \ all \ flows \ i \ in \ FG \ do \\
\quad S_P = S_P + P(i)
\end{enumerate}

(c) It calculates the sending rates for all the flows in an FG and distributes them.

\[
for \ all \ flows \ i \ in \ FG \ do \\
\quad FSE_R(i) = (P(i) \times S_CR) / S_P \\
\quad send \ FSE_R(i) \ to \ the \ flow \ i
\end{enumerate}

5.3.2. Example algorithm 2 - Conservative Active FSE

This algorithm extends algorithm 1 to conservatively emulate the behavior of a single flow by proportionally reducing the aggregate rate on congestion. Simulations results in \([fse]\) indicate that it can significantly reduce queuing delay and packet loss.

(1) When a flow \( f \) starts, it registers itself with SBD and the FSE. FSE \( R \) is initialized with the congestion controller's initial rate. SBD will assign the correct FGI. When a flow is assigned an FGI, it adds its FSE \( R \) to \( S_CR \).

(2) When a flow \( f \) stops or pauses, its entry is removed from the list.

(3) Every time the congestion controller of the flow \( f \) determines a new sending rate \( CC_R \), the flow calls \( UPDATE \), which carries out the tasks listed below to derive the new sending rates for all the flows in the FG. A flow's \( UPDATE \) function uses a local
(i.e. per-flow) temporary variable $S_P$, which is the sum of all the priorities, and a local variable $DELTA$, which is used to calculate the difference between $CC_R$ and the previously stored $FSE_R$. To prevent flows from either ignoring congestion or overreacting, a timer keeps them from changing their rates immediately after the common rate reduction that follows a congestion event. This timer is set to 2 RTTs of the flow that experienced congestion because it is assumed that a congestion event can persist for up to one RTT of that flow, with another RTT added to compensate for fluctuations in the measured RTT value.

(a) It updates $S_{CR}$ based on $DELTA$.

```
if Timer has expired or not set then
    DELTA = $CC_R - FSE_R(f)$
    if $DELTA < 0$ then  // Reduce $S_{CR}$ proportionally
        $S_{CR} = S_{CR} * CC_R / FSE_R(f)$
        Set Timer for 2 RTTs
    else
        $S_{CR} = S_{CR} + DELTA$
    end if
end if
```

(b) It calculates the sum of all the priorities, $S_P$.

```
$S_P = 0$
for all flows $i$ in FG do
    $S_P = S_P + P(i)$
end for
```

(c) It calculates the sending rates for all the flows in an FG and distributes them.

```
for all flows $i$ in FG do
    $FSE_R(i) = (P(i) * S_{CR}) / S_P$
    send $FSE_R(i)$ to the flow $i$
end for
```

6. Application

This section specifies how the FSE can be applied to specific congestion control mechanisms and makes general recommendations that facilitate applying the FSE to future congestion controls.
6.1. NADA

Network-Assisted Dynamic Adaptation (NADA) [I-D.ietf-rmcat-nada] is a congestion control scheme for rtcweb. It calculates a reference rate $r_{ref}$ upon receiving an acknowledgment, and then, based on the reference rate, it calculates a video target rate $r_{vin}$ and a sending rate for the flows, $r_{send}$.

When applying the FSE to NADA, the UPDATE function call described in Section 5.3 gives the FSE NADA’s reference rate $r_{ref}$. The recommended algorithm for NADA is the Active FSE in Section 5.3.1.

In step 3 (c), when the FSE_R(i) is "sent" to the flow i, this means updating $r_{ref}(r_{vin}$ and $r_{send}$) of flow i with the value of FSE_R(i).

6.2. General recommendations

This section provides general advice for applying the FSE to congestion control mechanisms.

Receiver-side calculations:
When receiver-side calculations make assumptions about the rate of the sender, the calculations need to be synchronized or the receiver needs to be updated accordingly. This applies to TFRC [RFC5348], for example, where simulations showed somewhat less favorable results when using the FSE without a receiver-side change [fse].

Stateful algorithms:
When a congestion control algorithm is stateful (e.g., TCP, with Slow Start, Congestion Avoidance and Fast Recovery), these states should be carefully considered such that the overall state of the aggregate flow is correct. This may require sharing more information in the UPDATE call.

Rate jumps:
The FSE-based coupling algorithms can let a flow quickly increase its rate to its fair share, e.g. when a new flow joins or after a quiescent period. In case of window-based congestion controls, this may produce a burst which should be mitigated in some way. An example of how this could be done without using a timer is presented in [anrw2016], using TCP as an example.
7. Expected feedback from experiments

The algorithm described in this memo has so far been evaluated using simulations covering all the tests for more than one flow from [I-D.ietf-rmcat-eval-test] (see [IETF-93], [IETF-94]). Experiments should confirm these results using at least the NADA congestion control algorithm with real-life code (e.g., browsers communicating over an emulated network covering the conditions in [I-D.ietf-rmcat-eval-test]. The tests with real-life code should be repeated afterwards in real network environments and monitored. Experiments should investigate cases where the media coder’s output rate is below the rate that is calculated by the coupling algorithm (FSE_R in algorithms 1 and 2, section 5.3). Implementers and testers are invited to document their findings in an Internet draft.

8. Acknowledgements

This document has benefitted from discussions with and feedback from Andreas Petlund, Anna Brunstrom, David Hayes, David Ros (who also gave the FSE its name), Ingemar Johansson, Karen Nielsen, Kristian Hiorth, Mirja Kuehlewind, Martin Stiemerling, Varun Singh, Xiaoqing Zhu, and Zaheduzzaman Sarker. The authors would like to especially thank Xiaoqing Zhu and Stefan Holmer for helping with NADA and GCC.

This work was partially funded by the European Community under its Seventh Framework Programme through the Reducing Internet Transport Latency (RITE) project (ICT-317700).

9. IANA Considerations

This memo includes no request to IANA.

10. Security Considerations

In scenarios where the architecture described in this document is applied across applications, various cheating possibilities arise: e.g., supporting wrong values for the calculated rate, the desired rate, or the priority of a flow. In the worst case, such cheating could either prevent other flows from sending or make them send at a rate that is unreasonably large. The end result would be unfair behavior at the network bottleneck, akin to what could be achieved with any UDP based application. Hence, since this is no worse than UDP in general, there seems to be no significant harm in using this in the absence of UDP rate limiters.
In the case of a single-user system, it should also be in the interest of any application programmer to give the user the best possible experience by using reasonable flow priorities or even letting the user choose them. In a multi-user system, this interest may not be given, and one could imagine the worst case of an "arms race" situation, where applications end up setting their priorities to the maximum value. If all applications do this, the end result is a fair allocation in which the priority mechanism is implicitly eliminated, and no major harm is done.

11. References

11.1. Normative References

[I-D.ietf-rmcat-nada]


11.2. Informative References

[I-D.ietf-rmcat-eval-test]

[I-D.ietf-rmcat-gcc]


Appendix A. Application to GCC

Google Congestion Control (GCC) [I-D.ietf-rmcat-gcc] is another congestion control scheme for RTP flows that is under development. GCC is not yet finalised, but at the time of this writing, the rate control of GCC employs two parts: controlling the bandwidth estimate based on delay, and controlling the bandwidth estimate based on loss. Both are designed to estimate the available bandwidth, $A_{\text{hat}}$.

When applying the FSE to GCC, the UPDATE function call described in Section 5.3 gives the FSE GCC’s estimate of available bandwidth $A_{\text{hat}}$. The recommended algorithm for GCC is the Active FSE in Section 5.3.1. In step 3 (c), when the FSE $R(i)$ is "sent" to the flow $i$, this means updating $A_{\text{hat}}$ of flow $i$ with the value of FSE $R(i)$.

Appendix B. Scheduling

When connections originate from the same host, it would be possible to use only one single sender-side congestion controller which determines the overall allowed sending rate, and then use a local scheduler to assign a proportion of this rate to each RTP session. This way, priorities could also be implemented as a function of the scheduler. The Congestion Manager (CM) [RFC3124] also uses such a scheduling function.

Appendix C. Example algorithm – Passive FSE

Active algorithms calculate the rates for all the flows in the FG and actively distribute them. In a passive algorithm, UPDATE returns a rate that should be used instead of the rate that the congestion controller has determined. This can make a passive algorithm easier to implement; however, when round-trip times of flows are unequal, shorter-RTT flows may (depending on the congestion control algorithm) update and react to the overall FSE state more often than longer-RTT flows, which can produce unwanted side effects. This problem is more
significant when the congestion control convergence depends on the RTT. While the passive algorithm works better for congestion controls with RTT-independent convergence, it can still produce oscillations on short time scales. The algorithm described below is therefore considered as highly experimental. Results of a simplified passive FSE algorithm with both NADA and GCC can be found in [fse-noms].

This passive version of the FSE stores the following information in addition to the variables described in Section 5.2:

- The desired rate \( DR \). This can be smaller than the calculated rate if the application feeding into the flow has less data to send than the congestion controller would allow. In case of a bulk transfer, \( DR \) must be set to \( CC_R \) received from the flow’s congestion module.

The passive version of the FSE contains one static variable per FG called TLO (Total Leftover Rate -- used to let a flow ‘take’ bandwidth from application-limited or terminated flows) which is initialized to 0. For the passive version, \( S_CR \) is limited to increase or decrease as conservatively as a flow’s congestion controller decides in order to prohibit sudden rate jumps.

1. When a flow \( f \) starts, it registers itself with SBD and the FSE. \( FSE_R \) and \( DR \) are initialized with the congestion controller’s initial rate. SBD will assign the correct FGI. When a flow is assigned an FGI, it adds its \( FSE_R \) to \( S_CR \).

2. When a flow \( f \) stops or pauses, it sets its \( DR \) to 0 and sets \( P \) to -1.

3. Every time the congestion controller of the flow \( f \) determines a new sending rate \( CC_R \), assuming the flow’s new desired rate \( new_DR \) to be "infinity" in case of a bulk data transfer with an unknown maximum rate, the flow calls UPDATE, which carries out the tasks listed below to derive the flow’s new sending rate, \( Rate \). A flow’s UPDATE function uses a few local (i.e. per-flow) temporary variables, which are all initialized to 0: DELTA, \( new_S_CR \) and \( S_P \).

   a. For all the flows in its FG (including itself), it calculates the sum of all the calculated rates, \( new_S_CR \). Then it calculates the difference between \( FSE_R(f) \) and \( CC_R \), DELTA.
for all flows i in FG do
    new_S_CR = new_S_CR + FSE_R(i)
end for
DELTA = CC_R - FSE_R(f)

(b) It updates S_CR, FSE_R(f) and DR(f).

FSE_R(f) = CC_R
if DELTA > 0 then // the flow’s rate has increased
    S_CR = S_CR + DELTA
else if DELTA < 0 then
    S_CR = new_S_CR + DELTA
end if
DR(f) = min(new_DR,FSE_R(f))

(c) It calculates the leftover rate TLO, removes the terminated flows from the FSE and calculates the sum of all the priorities, S_P.

for all flows i in FG do
    if P(i)<0 then
        delete flow
    else
        S_P = S_P + P(i)
    end if
end for
if DR(f) < FSE_R(f) then
    TLO = TLO + (P(f)/S_P) * S_CR - DR(f))
end if

(d) It calculates the sending rate, Rate.

Rate = min(new_DR, (P(f)*S_CR)/S_P + TLO)
if Rate != new_DR and TLO > 0 then
    TLO = 0 // f has ‘taken’ TLO
end if

(e) It updates DR(f) and FSE_R(f) with Rate.

if Rate > DR(f) then
    DR(f) = Rate
end if
FSE_R(f) = Rate
The goals of the flow algorithm are to achieve prioritization, improve network utilization in the face of application-limited flows, and impose limits on the increase behavior such that the negative impact of multiple flows trying to increase their rate together is minimized. It does that by assigning a flow a sending rate that may not be what the flow's congestion controller expected. It therefore builds on the assumption that no significant inefficiencies arise from temporary application-limited behavior or from quickly jumping to a rate that is higher than the congestion controller intended. How problematic these issues really are depends on the controllers in use and requires careful per-controller experimentation. The coupled congestion control mechanism described here also does not require all controllers to be equal; effects of heterogeneous controllers, or homogeneous controllers being in different states, are also subject to experimentation.

This algorithm gives all the leftover rate of application-limited flows to the first flow that updates its sending rate, provided that this flow needs it all (otherwise, its own leftover rate can be taken by the next flow that updates its rate). Other policies could be applied, e.g. to divide the leftover rate of a flow equally among all other flows in the FGI.

C.1. Example operation (passive)

In order to illustrate the operation of the passive coupled congestion control algorithm, this section presents a toy example of two flows that use it. Let us assume that both flows traverse a common 10 Mbit/s bottleneck and use a simplistic congestion controller that starts out with 1 Mbit/s, increases its rate by 1 Mbit/s in the absence of congestion and decreases it by 2 Mbit/s in the presence of congestion. For simplicity, flows are assumed to always operate in a round-robin fashion. Rate numbers below without units are assumed to be in Mbit/s. For illustration purposes, the actual sending rate is also shown for every flow in FSE diagrams even though it is not really stored in the FSE.

Flow #1 begins. It is a bulk data transfer and considers itself to have top priority. This is the FSE after the flow algorithm’s step 1:

<table>
<thead>
<tr>
<th>#</th>
<th>FGI</th>
<th>P</th>
<th>FSE_R</th>
<th>DR</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

S_CR = 1, TLO = 0
Its congestion controller gradually increases its rate. Eventually, at some point, the FSE should look like this:

<table>
<thead>
<tr>
<th>#</th>
<th>FGI</th>
<th>P</th>
<th>FSE_R</th>
<th>DR</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

S_CR = 10, TLO = 0

Now another flow joins. It is also a bulk data transfer, and has a lower priority (0.5):

<table>
<thead>
<tr>
<th>#</th>
<th>FGI</th>
<th>P</th>
<th>FSE_R</th>
<th>DR</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.5</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

S_CR = 11, TLO = 0

Now assume that the first flow updates its rate to 8, because the total sending rate of 11 exceeds the total capacity. Let us take a closer look at what happens in step 3 of the flow algorithm.

CC_R = 8. new_DR = infinity.
3 a) new_S_CR = 11; DELTA = 8 - 10 = -2.
3 b) FSE_R(f) = 8. DELTA is negative, hence S_CR = 9; DR(f) = 8.
3 c) S_P = 1.5.
3 d) new sending rate = min(infinity, 1/1.5 * 9 + 0) = 6.
3 e) FSE_R(f) = 6.

The resulting FSE looks as follows:

<table>
<thead>
<tr>
<th>#</th>
<th>FGI</th>
<th>P</th>
<th>FSE_R</th>
<th>DR</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.5</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

S_CR = 9, TLO = 0
The effect is that flow #1 is sending with 6 Mbit/s instead of the 8 Mbit/s that the congestion controller derived. Let us now assume that flow #2 updates its rate. Its congestion controller detects that the network is not fully saturated (the actual total sending rate is 6+1=7) and increases its rate.

CC_R = 2. new_DR = infinity.

3 a) new_S_CR = 7; DELTA = 2 - 1 = 1.
3 b) FSE_R(f) = 2. DELTA is positive, hence S_CR = 9 + 1 = 10;
   DR(f) = 2.
3 c) S_P = 1.5.
3 d) new sending rate = min(infinity, 0.5/1.5 * 10 + 0) = 3.33.
3 e) DR(f) = FSE_R(f) = 3.33.

The resulting FSE looks as follows:

<table>
<thead>
<tr>
<th>#</th>
<th>FGI</th>
<th>P</th>
<th>FSE_R</th>
<th>DR</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.5</td>
<td>3.33</td>
<td>3.33</td>
<td>3.33</td>
</tr>
</tbody>
</table>

S_CR = 10, TLO = 0

The effect is that flow #2 is now sending with 3.33 Mbit/s, which is close to half of the rate of flow #1 and leads to a total utilization of 6(#1) + 3.33(#2) = 9.33 Mbit/s. Flow #2’s congestion controller has increased its rate faster than the controller actually expected. Now, flow #1 updates its rate. Its congestion controller detects that the network is not fully saturated and increases its rate. Additionally, the application feeding into flow #1 limits the flow’s sending rate to at most 2 Mbit/s.
CC_R=7. new_DR=2.
3 a) new_S_CR = 9.33; DELTA = 1.
3 b) FSE_R(f) = 7, DELTA is positive, hence S_CR = 10 + 1 = 11;
    DR = min(2, 7) = 2.
3 c) S_P = 1.5; DR(f) < FSE_R(f), hence TLO = 1/1.5 * 11 - 2 = 5.33.
3 d) new sending rate = min(2, 1/1.5 * 11 + 5.33) = 2.
3 e) FSE_R(f) = 2.

The resulting FSE looks as follows:

<table>
<thead>
<tr>
<th></th>
<th>#</th>
<th>FGI</th>
<th>P</th>
<th>FSE_R</th>
<th>DR</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.5</td>
<td>3.33</td>
<td>3.33</td>
<td></td>
<td>3.33</td>
</tr>
</tbody>
</table>

S_CR = 11, TLO = 5.33

Now, the total rate of the two flows is 2 + 3.33 = 5.33 Mbit/s, i.e. the network is significantly underutilized due to the limitation of flow #1. Flow #2 updates its rate. Its congestion controller detects that the network is not fully saturated and increases its rate.

CC_R=4.33. new_DR = infinity.
3 a) new_S_CR = 5.33; DELTA = 1.
3 b) FSE_R(f) = 4.33. DELTA is positive, hence S_CR = 12;
    DR(f) = 4.33.
3 c) S_P = 1.5.
3 d) new sending rate: min(infinity, 0.5/1.5 * 12 + 5.33 ) = 9.33.
3 e) FSE_R(f) = 9.33, DR(f) = 9.33.

The resulting FSE looks as follows:

<table>
<thead>
<tr>
<th></th>
<th>#</th>
<th>FGI</th>
<th>P</th>
<th>FSE_R</th>
<th>DR</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.5</td>
<td>9.33</td>
<td>9.33</td>
<td>9.33</td>
<td>9.33</td>
</tr>
</tbody>
</table>

S_CR = 12, TLO = 0

Now, the total rate of the two flows is 2 + 9.33 = 11.33 Mbit/s.
Finally, flow #1 terminates. It sets P to -1 and DR to 0. Let us
assume that it terminated late enough for flow #2 to still experience the network in a congested state, i.e. flow #2 decreases its rate in the next iteration.

CC_R = 7.33. new_DR = infinity.
3 a) new_S_CR = 11.33; DELTA = -2.
3 b) FSE_R(f) = 7.33. DELTA is negative, hence S_CR = 9.33;
   DR(f) = 7.33.
3 c) Flow 1 has P = -1, hence it is deleted from the FSE.
   S_P = 0.5.
3 d) new sending rate: min(infinity, 0.5/0.5*9.33 + 0) = 9.33.
3 e) FSE_R(f) = DR(f) = 9.33.

The resulting FSE looks as follows:

<table>
<thead>
<tr>
<th>#</th>
<th>FGI</th>
<th>P</th>
<th>FSE_R</th>
<th>DR</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0.5</td>
<td>9.33</td>
<td>9.33</td>
<td>9.33</td>
</tr>
</tbody>
</table>

S_CR = 9.33, TLO = 0

Appendix D. Change log

D.1. draft-welzl-rmcat-coupled-cc

D.1.1. Changes from -00 to -01
   o Added change log.
   o Updated the example algorithm and its operation.

D.1.2. Changes from -01 to -02
   o Included an active version of the algorithm which is simpler.
   o Replaced "greedy flow" with "bulk data transfer" and "non-greedy" with "application-limited".
   o Updated new_CR to CC_R, and CR to FSE_R for better understanding.
D.1.3. Changes from -02 to -03
  
  o  Included an active conservative version of the algorithm which reduces queue growth and packet loss; added a reference to a technical report that shows these benefits with simulations.

  o  Moved the passive variant of the algorithm to appendix.

D.1.4. Changes from -03 to -04
  
  o  Extended SBD section.

  o  Added a note about window-based controllers.

D.1.5. Changes from -04 to -05
  
  o  Added a section about applying the FSE to specific congestion control algorithms, with a subsection specifying its use with NADA.

D.2. draft-ietf-rmcat-coupled-cc

D.2.1. Changes from draft-welzl-rmcat-coupled-cc-05
  
  o  Moved scheduling section to the appendix.

D.2.2. Changes from -00 to -01
  
  o  Included how to apply the algorithm to GCC.

  o  Updated variable names of NADA to be in line with the latest version.

  o  Added a reference to [I-D.ietf-rtcweb-transports] to make a connection to the prioritization text there.

D.2.3. Changes from -01 to -02
  
  o  Minor changes.

  o  Moved references of NADA and GCC from informative to normative.

  o  Added a reference for the passive variant of the algorithm.
D.2.4. Changes from -02 to -03

- Minor changes.
- Added a section about expected feedback from experiments.

D.2.5. Changes from -03 to -04

- Described the names of variables used in the algorithms.
- Added a diagram to illustrate the interaction between flows and the FSE.
- Added text on the trade-off of using the configuration based approach.
- Minor changes to enhance the readability.

D.2.6. Changes from -04 to -05

- Changed several occurrences of "NADA and GCC" to "NADA", including the abstract.
- Moved the application to GCC to an appendix, and made the GCC reference informative.
- Provided a few more general recommendations on applying the coupling algorithm.

Authors’ Addresses

Safiqul Islam
University of Oslo
PO Box 1080 Blindern
Oslo, N-0316
Norway

Phone: +47 22 84 08 37
Email: safiquli@ifi.uio.no
Michael Welzl
University of Oslo
PO Box 1080 Blindern
Oslo, N-0316
Norway

Phone: +47 22 85 24 20
Email: michawe@ifi.uio.no

Stein Gjessing
University of Oslo
PO Box 1080 Blindern
Oslo, N-0316
Norway

Phone: +47 22 85 24 44
Email: steing@ifi.uio.no
Chapter 11

TCP-CCC: Single-Path TCP Congestion Control Coupling
TCP-CCC: single-path TCP congestion control coupling
draft-welzl-tcp-ccc-00

Abstract

This document specifies a method, TCP-CCC, to combine the congestion controls of multiple TCP connections between the same pair of hosts. This can have several performance benefits, and it makes it possible to precisely assign a share of the congestion window to the connections based on priorities. This document also addresses the problem that TCP connections between the same pair of hosts may not share the same path. We discuss methods to detect if, or enforce that connections traverse a common bottleneck.

Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on May 4, 2017.
1. Introduction

When multiple TCP connections between the same host pair compete on the same bottleneck, they often incur more delay and losses than a single TCP connection. Moreover, it is often not possible to precisely divide the available capacity among the connections. To address this problem, this document presents TCP-CCC, a method to combine the congestion controls of multiple TCP connections between the same pair of hosts. This can have several performance benefits:

- Reduced average loss and queuing delay (because the competition between the encapsulated TCP connections is avoided)
- Assign a precise capacity share based on a priority.
o Even in the absence of prioritization, better fairness between the TCP connections.

o No need for new connections to slow start up to a reasonable cwnd value that ongoing connections already have: a connection can immediately be assigned its share of the aggregate’s total cwnd. This can significantly reduce the completion time of short connections.

All of these benefits only play out when there are more than one TCP connections. Some of the benefits in the list above are more significant when some transfers are short. This makes the usage of TCP-CCC especially attractive in situations where some transfers are short.

We discuss methods to determine if connections traverse the same bottleneck as well as methods to ensure this. To this end, we propose a light-weight, dynamically configured TCP-in-UDP (TiU) encapsulation scheme. TiU is optional, as our coupled congestion control strategy is applicable wherever overlapping TCP flows must follow the same path (such as when routed over a VPN tunnel).

2. Coupled Congestion Control

For each TCP connection \( c \), the algorithm described below receives cwnd and ssthresh as input and stores the following information:

- The Connection ID.
- A priority \( P(c) \) -- e.g., an integer value in the range from 1 (unimportant) to 10 (very important).
- The previously used cwnd used by the connection \( c \), \( ccc\_cwnd(c) \).
- The previously used ssthresh used by the connection \( c \), \( ccc\_ssthresh(c) \).

Three global variables \( \text{sum\_cwnd} \), \( \text{sum\_ssthresh} \) and \( \text{sum\_p} \) are used to represent the sum of all the \( ccc\_cwnd \) values, \( ccc\_ssthresh \) values and priorities of all TCP connections, respectively. \( \text{sum\_cwnd} \) and \( \text{sum\_ssthresh} \) are used to update the cwnd and ssthresh values for all connections.

This algorithm emulates the behavior of a single TCP connection by choosing one connection as the connection that dictates the increase / decrease behavior for the aggregate. We call it the "Coordinating Connection" (CoCo). The algorithm was designed to be as simple as possible. Below, abbreviations are used to refer to the phases of
TCP congestion control as defined in [RFC5681]: SS refers to Slow Start, CA refers to Congestion Avoidance and FR refers to Fast Recovery.

For simplicity, this algorithm refrains from changing cwnd when a connection is in FR. SS should not happen as long as ACKs arrive. Hence, the algorithm ensures that the aggregate’s behavior is only dictated by SS when all connections are in the SS phase. We use a bit array, ssbits, with a bit for each connection in the group. We set the bit if the connection state is SS due to an RTO.

1. When a connection c starts, it adds its priority \( P(c) \) to \( \text{sum}_p \). If it is the very first connection, it sets \( \text{sum}_c\text{wnd} \) to its own cwnd. After that, the connection’s globally known cwnd and ssthresh values (\( \text{ccc}_\text{cwnd}(c) \) and \( \text{ccc}_\text{ssthresh}(c) \)) are updated, and the connection updates its own cwnd and ssthresh values to be equal to \( \text{ccc}_\text{cwnd}(c) \) and \( \text{ccc}_\text{ssthresh}(c) \).

\[
\begin{align*}
\text{ccc}_P(c) &= P \\
\text{sum}_P &= \text{sum}_P + P \\
\text{sum}_\text{cwnd} &= \text{sum}_\text{cwnd} + \text{cwnd} \\
\text{ccc}_\text{cwnd}(c) &= \frac{\text{sum}_\text{cwnd}}{\text{sum}_P} \\
\text{ccc}_\text{ssthresh}(c) &= \text{ssthresh} \quad \text{if sum_ssthresh} > 0 \\
&\quad \text{end if} \\
&\quad \text{ccc}_\text{ssthresh}(c) = \frac{\text{sum}_\text{ssthresh}}{\text{sum}_P} \\
&\quad \text{end if}
\end{align*}
\]

// Update c’s own cwnd and ssthresh for immediate use:
Send \( \text{ccc}_\text{cwnd}(c) \) and \( \text{ccc}_\text{ssthresh}(c) \) to c

2. When a connection c stops, its entry is removed. \( \text{sum}_p \) is recalculated.

\[
\text{if c = CoCo then} \\
\text{Coco = the next connection} \\
\text{end if}
\]

\[
\text{sum}_p = \text{sum}_p - \text{ccc}_P(c) \\
\text{Remove \( \text{ccc}_P(c), \text{ccc}_\text{cwnd}(c), \text{ccc}_\text{ssthresh}(c) \)}
\]

3. Every time the congestion controller of a connection c calculates a new cwnd, the connection calls UPDATE, which carries out the tasks listed below to derive the new cwnd and ssthresh values. Whenever the CoCo calls UPDATE, \( \text{sum}_\text{cwnd} \) and \( \text{sum}_\text{ssthresh} \) are additionally updated to reflect the current sum of all stored \( \text{ccc}_\text{cwnd} \) and \( \text{ccc}_\text{ssthresh} \) values. Initially,
there is only one connection and this connection automatically becomes the CoCo. It updates sum_cwnd to its own cwnd and sets sum_ssthresh to 0.

(4) WHEN a non-CoCo connection c CALLS UPDATE......

if(all of the connections including CoCo are in CA but c is in FR) c becomes the new CoCo.
else
  if(c is in CA or SS)
    c’s cwnd is assigned its previously stored ccc_cwnd value.

(5) WHEN c(chooseCoCo) CALLS UPDATE......

if CoCo == c then
  if state == CA and ssbits(c) == 0 then
    if cwnd >= ccc_cwnd(c) then // increased cwnd
      sum_cwnd = sum_cwnd + cwnd - ccc_cwnd(c)
    else
      sum_cwnd = sum_cwnd * cwnd / ccc_cwnd(c)
    end if
  ccc_cwnd(c) = ccc_P(c) * sum_cwnd / sum_p
  ccc_ssthresh(c) = ssthresh
  if sum_ssthresh > 0 then
    ccc_ssthresh(c) = ccc_P(c) * sum_ssthresh/sum_p
  end if
else if state == FR then
  sum_ssthresh = sum_cwnd/2
else if state == SS then
  if c experienced a timeout then
    ssbits(c) = 1
  end if
  if ssbits(x) == 1 for all x then
    ssbits(x) = 0 // for all x
    sum_cwnd = sum_cwnd * cwnd / ccc_cwnd(c)
    ccc_cwnd(c) = ccc_P(c) * sum_cwnd / sum_p
    sum_ssthresh = sum_cwnd/2
  else
    CoCo = first connection where ccc_state == SS
  end if
end if
end if

(6) After that, if the ccc_state(c) is not equal to FR
if state != FR then
    Send ccc_cwnd(c) and ccc_ssthresh(c) to c
end if

When a flow gets a large share of the aggregate immediately after joining, it can potentially create a burst in the network. We propose a mechanism [anrw2016] to clock the packet transmission out by using the ack-clock of TCP. Our algorithm achieves a form of "pacing", but it does not rely on any timers.

When a connection c joins, it turns on the ack-clock feature and calculates the share of the aggregate, clocked_cwnd c. Below, we illustrate the ack-clock mechanism that is used to distribute the share of the cwnd based on the acknowledgements received from other flows.

if clocked_cwnd(c) <= 0 then
    return // alg. ends; other connections can increase cwnd again
end if
if number_of_acks c % N = 0 then
    send a new segment for connection c
    clocked_cwnd(c)= clocked_cwnd(c) - 1
end if
number_of_acks(c) = number_of_acks(c) + 1

3. Ensuring a Common Bottleneck

Our algorithm, as well as EFCM [EFCM], E-TCP [EFCM] and the CM [RFC3124] assume that multiple TCP connections between the same host pair traverse the same bottleneck. This is not always true: load-balancing mechanisms such as Link Aggregation Group (LAG) and Equal-Cost Multi-Path (ECMP) may force them to take different paths [RFC7424]. If this leads to the connections seeing different bottlenecks, combining the congestion controllers would incur wrong behavior. There are, however, several application scenarios where the single-bottleneck assumption is correct.

Sometimes, the network configuration is known, and it is known that mechanisms such as ECMP and LAG do not operate on the bottleneck or are simply not in use. Alternatively, measurements can infer whether flows traverse the same bottleneck [I-D.ietf-rmcat-sbd]. When IPv6 is available, the TCP connections could be assigned the same IPv6 flow label. According to [RFC6437], "The usage of the 3-tuple of the Flow Label, Source Address, and Destination Address fields enables efficient IPv6 flow classification, where only IPv6 main header
fields in fixed positions are used" - this would be favorable for TCP congestion control coupling. However, this [RFC6437] does not make a clear recommendation about either using the 3-tuple or 5-tuple (which includes the port numbers) - both methods are valid. Thus, whether it works to use the flow label as the sole means to put connections on the same path depends on router configuration. When it works, it is an attractive option because it does not require changing the receiver.

Finally, encapsulating packets with a header that ensures a common path is another possibility to make connections traverse the same bottleneck. We will discuss encapsulation in the next section.

3.1. Encapsulation

3.1.1. TCP in UDP

3.1.1.1. Introduction

We want to be able to ensure that TCP congestion control coupling can always work, provided that the required code is available at the receiver - and be able to efficiently fall back to the standard behaviour in case it is not. To achieve this, we present a method, TCP-in-UDP (TiU), to encapsulate multiple TCP connections using the same UDP port pair.

TCP-in-UDP (TiU) is based on [Che13]. It differs from it in that:

- Other than [Che13], TiU encapsulates multiple TCP connections using the same UDP port number pair. TCP port numbers are preserved; a single well-known UDP port is used for TiU. If TiU is implemented in the kernel, this allows using normal TCP sockets, where enabling the usage of TiU could be done via a socket option, for example.

- The header format is slightly different to allow representing a TCP connection with a few bits that are encoded across the original TCP header’s "Reserved" field and the URG (Urgent) flag to encode a Connection ID. With this encoding, similar to the encapsulation in [Che13], the total TiU header size does not exceed the original TCP header size.

- A (TiU-encapsulated) TCP SYN uses a newly defined TCP option to establish the mapping between a Connection ID and the original TCP port number pair.

TiU inherits all the benefits of [Che13] and a preceding similar proposal, [Den08]. It enables TCP-CCC coupled congestion control,
and it adds the potential disadvantage of not being able to benefit from ECMP. In short, the benefits and features of TiU that are already explained in detail in [Che13] and [Den08] are:

- To establish direct communication between two devices that are both behind NAT gateways, Interactive Connectivity Establishment (ICE) [RFC5245] is used to create the necessary mappings in both NAT gateways, and ICE can have higher success rates using UDP [RFC5128].

- TCP options, as required for Multipath TCP [RFC6824], for example, are expected to work more reliably because middleboxes will be less able to interfere with them.

- Because the packet format allows the first octet to be in the range 0x0-0x3 (as is the case for a STUN [RFC5389] packet, where the most significant two bits are always zero), the UDP port number pair used by TiU can be used to exchange STUN packets with a STUN server that is unaware of TiU.

- Following the method described in [Che13] and [Den08], other transport protocols than TCP (e.g., SCTP) could be UDP-encapsulated in a similar fashion. With TiU, the same outer UDP port number pair could be used for different encapsulated protocols at the same time.

[Che13] also lists a disadvantage of UDP-encapsulating TCP packets: because NAT gateways typically use shorter timeouts for UDP port mappings than they do for TCP port mappings, long-lived UDP-encapsulated TCP connections will need to send more frequent keepalive packets than native TCP connections. TiU inherits this problem too, although using a single five-tuple for multiple TCP connections alleviates it by reducing the chance of experiencing long periods of silence.

3.1.1.2. Specification

TiU uses a header that is very similar to the header format in [Den08] and [Che13], where it is explained in greater detail. It consists of a UDP header that is followed by a slightly altered TCP header. The UDP source and destination ports are semantically different from [Den08] and [Che13]: TiU uses a single well-known UDP port, and multiple TCP connections use the same UDP port number pair. The encapsulated TCP header is changed to fit into a UDP packet without increasing the MSS; this is achieved by removing the TCP source and destination ports, the Urgent Pointer and the (now unnecessary) TCP checksum. Moreover, the order of fields is changed to move the Data Offset field to the beginning of the UDP payload.
This allows using it to identify other encapsulated content such as a STUN packet: for TCP, the Data Offset must be at least 5, i.e. the most-significant four bits of the first octet of the UDP payload are in the range 0x5-0xF, whereas this is not the case for other protocols (e.g., STUN requires these bits to be 0). The altered TCP header for TiU is shown below:

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Length             |           Checksum            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data | Conn  |C|E|C|A|P|R|S|F|                               |
| Offset|  ID   |W|C|I|C|S|S|Y|I|            Window             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      (Optional) Options                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 1: Encapsulated TCP-in-UDP Header Format (the first 8 bytes are the UDP header)

Different from [Den08] and [Che13], the least-significant four bits of the first octet and a bit that replaces the URG bit in the next octet together form a five-bit "Connection ID" (Conn ID). TiU maintains the port numbers of the TCP connections that it encapsulates; the Connection ID is a way to encode the port number information with a few unused header bits. It uniquely identifies a port number pair of a TCP connection that is encapsulated with TiU. Using these five bits, TiU can combine up to 32 TCP connections with one UDP port number pair.

The TiU-TCP SYN and SYN/ACK packets look slightly different, because they need to establish the mapping between the Connection ID and the port numbers that are used by TiU-encapsulated TCP connections:
The Encapsulated Source Port and Encapsulated Destination Port are the port numbers of the TCP connection. To create this header, an implementation can simply swap the position of the original TCP header’s port number fields with the position of the Data Offset / Reserved / Flags / Window fields.

Every TiU SYN or TiU SYN-ACK packet also carries at least the TiU-Setup TCP option. This option contains a Connection ID number. On a SYN packet, it is the Connection ID that the sender intends to use in future packets to represent the Encapsulated Source Port and Encapsulated Destination Port. On a SYN/ACK packet, it confirms that such usage is accepted by the recipient of the SYN. A special value of 255 is used to signify an error, upon which TiU will no longer be used (i.e., the next packet is expected to be a non-encapsulated TCP packet). The TiU-Setup TCP option is defined as follows:

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Kind |    Length     |     ExID                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Connection ID |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3: TiU Setup TCP Option
The option follows the format for Experimental TCP Options defined in [RFC6994]. It has Kind=253, Length=5, an ExID that is with value TBD (see Section 6) and the Connection ID. The Connection ID is an 8-bit field for easier parsing, but only values 0-31 are valid Connection IDs (because the Connection ID in non-SYN or SYN/ACK TiU packets is only 5 bit long).

3.1.1.3. Protocol Operation and Implementation Notes

There can be several ways to implement TCP-in-UDP. The following gives an overview of how a TiU implementation can operate. This description matches the implementation described in Section 5.

A goal of TiU is to achieve congestion control coupling with a simple implementation that minimizes changes to existing code. It is thus recommendable to implement TiU in the kernel, as a change to the existing kernel TCP code.

- Encapsulation and decapsulation: this is code that should, in the simplest case, operate just before a TCP segment is transmitted. Based on e.g. a socket option that enables/disables TiU, the TCP segment is changed into the TiU header format (Figure 1). In case it is a TCP SYN or TCP SYN/ACK packet, the header format is defined as in Figure 2, and the TiU-Setup TCP option is appended. This packet is then transmitted. For decapsulation, the reverse mechanism applies, upon reception of a UDP packet that uses destination port XXX (TBD, see Section 6). Both hosts keep a list of encapsulated TCP port numbers and their corresponding Connection IDs. In case a SYN packet requests using a Connection ID that is already reserved, an error (Connection ID value 255 in the TiU Setup TCP option) must be signified to the other end in a TiU-encapsulated TCP SYN/ACK, and encapsulation must be disabled on all further TCP packets. Similarly, when receiving a TiU SYN/ACK with an error, a TCP sender must stop encapsulating TCP packets.

The TCP port number space usage on the host is left unchanged: the original code can reserve TCP ports as it always did. Except for the TiU encapsulation compressing the port numbers into a Connection ID field, TCP ports should be used similar to normal TCP operation. A TCP port that is in use by a TiU-encapsulated TCP connection must therefore not be made available to non-encapsulated TCP connections, and vice versa.

For each TCP connection, two variables must be configured: 1) TiU-ENABLE, which is a boolean, deciding whether to use TiU or not, and 2) Priority, which is a value, e.g. from 1 to 10, that is used by the coupled congestion control algorithm to assign an appropriate share.
of the total cwnd to the connection. Priority values are local and their range does not matter for this algorithm: the algorithm works with a flow's priority portion of the sum of all priority values. The configuration of the two per-connection variables can be implemented in various ways, e.g. through an API option.

With these code changes in place, TiU can operate as follows, assuming no previous TiU connections have been made between a specific host pair and a client tries to connect to a server:

- An application uses an API option to request TiU operation. The kernel then sends out a TiU TCP SYN that contains a TiU-Setup TCP option. This packet header contains the encapsulated TCP port numbers (source port A and destination port B) and the Connection ID X.

- The server listens on UDP port XXX (TBD, see Section 6). Upon receiving a packet on this port, it knows that it is a TiU packet and decodes it, handing the resulting TCP packet over to "normal" TCP processing. The TiU-Setup TCP option allows the server to associate future TiU packets containing Connection ID X with ports A and B. The server sends its response as a TiU SYN-ACK.

- TCP operates as normal from here on, but packets are TiU-encapsulated before sending them out and decapsulated upon reception, using Connection ID X. Both hosts associate TiU packets carrying Connection ID X with a local identifier that matches ports A and B, just like they would associate non-encapsulated TCP packets with the same local identifier when seeing ports A and B in the TCP header.

- If an application on either side of the TiU connection wants to connect to a destination host on the other side and requests TiU operation, the kernel sends out another TiU TCP SYN, this time containing a different TCP source port number and either the same or a different destination port number (C and D), and a TiU-Setup TCP option with Connection ID Y. From now on, packets carrying Connection ID Y will be associated with ports C and D on both hosts. Otherwise, TiU operation continues as described above.

- Now, because there are two or more connections available between the same host pair, coupled congestion control begins to operate for all outgoing TiU packets (see Section 2 for details). This is a local operation, applying the priority values that were configured to use for the TiU-encapsulated TCP connections.

Unless it is known that UDP packets with destination port number XXX (TBD, see Section 6) can be used without problems on the path between
two communicating hosts, it is advisable for TiU implementations to
contain methods to fall back to non-encapsulated ("raw") TCP
communication. Such fall-back must be supported for the case of
Connection ID collisions anyway. Middleboxes have been known to
track TCP connections [Honda11], and falling back to communication
with raw TCP packets without ever using a raw TCP SYN - SYN/ACK
handshake may lead to problems with such devices. The following
method is recommended to efficiently fall back to raw TCP
communication:

- After sending out a TiU SYN packet, additionally send a raw TCP
  SYN packet.

- After sending out a TiU SYN/ACK packet, additionally send a raw
  TCP SYN/ACK packet.

- Upon receiving a TiU SYN packet, after responding with a TiU SYN/
  ACK packet and raw TCP SYN/ACK packet, immediately store the
  encapsulated port numbers and Connection ID. As long as a TiU
  connection is ongoing, ignore any additional incoming TCP SYN or
  TCP SYN/ACK packets from the same host that carry port numbers
  matching the stored encapsulated port numbers. Otherwise, process
  TCP SYN or TCP SYN/ACK packets as normal.

This method ensures that the TCP SYN / SYN/ACK handshake is visible
to middleboxes and allows to immediately switch back to raw TCP
communication in case of failures. If implemented on both sides as
described above and no TiU SYN or TiU SYN/ACK packet arrives, yet a
TCP SYN or TCP SYN/ACK packet does, this can only mean that the other
host does not support TiU, a UDP packet was dropped, or the UDP and
TCP packets were reordered in transit. Reordering in the host (e.g.,
a server responding to a TCP SYN before it responds to a TiU SYN) can
be a problem for similar methods (e.g. [RFC6555]), but it can be
eliminated by prescribing the processing order as above.

Because TCP does not preserve message boundaries and the size of the
TCP header can vary depending on the options that are used, it is
also no problem to precede the TCP header in the UDP packet with a
different header (e.g. PLUS or SPUD [I-D.hildebrand-spud-prototype])
without exceeding the known MTU limit. When creating a TCP segment,
a TCP sender needs to consider the length of this header when
calculating the segment size, just like it would consider the length
of a TCP option. For this to work, the usage of other headers such
as PLUS or SPUD in-between the UDP header and the TiU header must
therefore be known to both the sender-side and receiver-side code
that processes TiU.
3.1.1.4. Usage Considerations

TiU cannot work with applications that require the Urgent pointer (which is not recommended for use by new applications anyway [RFC6093], but should be consider if TiU is implemented in a way that allows it to be applied onto existing applications; telnet is a well-known example of an application that uses this functionality). It can also be used as a method to experimentally test new TCP functionality in the presence of middleboxes that would otherwise create problems (as some have been known to do [Honda11]).

Reasons to use TiU include the benefits of [Che13] and [Den08] that were discussed in Section 1. TiU has the disadvantage of disabling ECMP for the TCP connections that it encapsulates. This can reduce the capacity usage of these TCP connections. It has the advantage of being able to apply TCP-CCC coupled congestion control, which can provide precise congestion window assignment based on a priority.

3.1.2. Other Methods

There are many possible encapsulation schemes for various use cases. For example, Generic UDP Encapsulation (GUE) [I-D.draft-ietf-nvo3-gue] allows us to multiplex several TCP connections onto a same UDP port number pair. Several encapsulation methods transmit layer-2 frames over an IP network - e.g. VXLAN [RFC7348] (over UDP/IP) and NVGRE [RFC7637] (over GRE/IP). Because Layer-2 networks should be agnostic to the transport connections running over them, the path should not depend on the TCP port number pair and our algorithm should work. Some care must still be taken: for example, for NVGRE, [RFC7637] says: "If ECMP is used, it is RECOMMENDED that the ECMP hash is calculated either using the outer IP frame fields and entire Key field (32 bits) or the inner IP and transport frame fields". If routers do use the inner transport frame fields (typically, port numbers) for this hashing, we have the same problem even over NVGRE.

4. Related Work

The TCPMUX mechanism in [RFC1078] multiplexes TCP connections under the same outer transport port number; it does however not preserve the port numbers of the original TCP connections, and no method to couple congestion controls is described in [RFC1078].

Congestion control coupling follows the style of RTP application congestion control coupling in [I-D.ietf-rmcat-coupled-cc] which is designed to be easy to implement, and to minimize the number of changes that need to be made to the underlying congestion control mechanisms. This method was shown to yield several benefits in...
TCP-CCC requires slightly deeper changes to TCP’s congestion control, making it harder to implement than [I-D.ietf-rmcat-coupled-cc], but it is still a much smaller code change than the Congestion Manager [RFC3124].

Combining congestion controls as TCP-CCC does it has some similarities with Ensemble Sharing in [RFC2140], which however only concerns initial values of variables used by new connections and does not share the congestion window (cwnd). The cwnd variable is shared across ongoing connections in [ETCP] and [EFCM], and the mechanism described in Section 2 resembles the mechanisms in these works, but neither [ETCP] nor [EFCM] address the problem of ECMP.

Coupled congestion control has also been specified for Multipath TCP [RFC6356]. MPTCP’s coupled congestion control combines the congestion controls of subflows that may traverse different paths, whereas we propose congestion control coupling for flows sharing a single-path. TCP-CCC builds on the assumption that all its encapsulated TCP connections traverse the same path. This makes the two methods for coupled congestion control very different, even though they both aim at emulating the behavior of a single TCP connection in the case where all flows traverse the same network bottleneck. For example, a new flow obtaining a a larger-than-IW share of the aggregate cwnd would be inappropriate for an MPTCP subflow.

5. Implementation Status

We have implemented TCP-CCC and TiU encapsulation for both the sender and receiver in the FreeBSD kernel, as a simple add-on to the TCP implementation that is controlled via a socket option.

6. IANA Considerations

This document specifies a new TCP option that uses the shared experimental options format [RFC6994]. No value has yet been assigned for ExID.

This document requires a well-known UDP port (referred to as port XXX in this document). Due to the highly experimental nature of TiU, this document is being shared with the community to solicit comments before requesting such a port number.

7. Security Considerations

TBD
8. Acknowledgements

This work has received funding from Huawei Technologies Co., Ltd., and the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 644334 (NEAT). The views expressed are solely those of the author(s).

9. References

9.1. Normative References


9.2. Informative References


Authors’ Addresses

Michael Welzl
University of Oslo
PO Box 1080 Blindern
Oslo N-0316
Norway

Email: michawe@ifi.uio.no

Safiqul Islam
University of Oslo
PO Box 1080 Blindern
Oslo N-0316
Norway

Phone: +47 22 84 08 37
Email: safiquli@ifi.uio.no

Kristian Hiorth
University of Oslo
PO Box 1080 Blindern
Oslo N-0316
Norway

Email: kristahi@ifi.uio.no
Jianjie You
Huawei
101 Software Avenue, Yuhua District
Nanjing 210012
China

Email: youjianjie@huawei.com
Chapter 12

TCP Control Block
Interdependence
TCP Control Block Interdependence
draft-touch-tcpm-02bis-02.txt

TCPM WG                                                       J. Touch
Internet Draft                                              USC/ISI
Intended status: Informational                           M. Welzl
Expires: July 2017                                     S. Islam
University of Oslo                                    J. You
                        Huawei
                January 12, 2017

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html
Abstract

This memo describes interdependent TCP control blocks, where part of the TCP state is shared among similar concurrent or consecutive connections. TCP state includes a combination of parameters, such as connection state, current round-trip time estimates, congestion control information, and process information. Most of this state is maintained on a per-connection basis in the TCP Control Block (TCB), but implementations can (and do) share certain TCB information across connections to the same host. Such sharing is intended to improve overall transient transport performance, while maintaining backward-compatibility with existing implementations. The sharing described herein is limited to only the TCB initialization and so has no effect on the long-term behavior of TCP after a connection has been established.

Table of Contents

1. Introduction................................................... 3
2. Conventions used in this document.............................. 3
3. Terminology.................................................... 4
4. The TCP Control Block (TCB).................................... 4
5. TCB Interdependence............................................ 5
6. An Example of Temporal Sharing................................ 5
7. An Example of Ensemble Sharing................................ 8
8. Compatibility Issues............................................ 10
9. Implications.................................................... 12
10. Implementation Observations.................................... 14
11. Security Considerations....................................... 15
12. IANA Considerations........................................... 16
13. References.................................................... 17
   13.1. Normative References.................................... 17
1. Introduction

TCP is a connection-oriented reliable transport protocol layered over IP [RFC793]. Each TCP connection maintains state, usually in a data structure called the TCP Control Block (TCB). The TCB contains information about the connection state, its associated local process, and feedback parameters about the connection’s transmission properties. As originally specified and usually implemented, most TCB information is maintained on a per-connection basis. Some implementations can (and now do) share certain TCB information across connections to the same host. Such sharing is intended to lead to better overall transient performance, especially for numerous short-lived and simultaneous connections, as often used in the World-Wide Web [Be94],[Br02].

This document discusses TCB state sharing that affects only the TCB initialization, and so has no effect on the long-term behavior of TCP after a connection has been established. Path information shared across SYN destination port numbers assumes that TCP segments having the same host-pair experience the same path properties, irrespective of TCP port numbers. The observations about TCB sharing in this document apply similarly to any protocol with congestion state, including SCTP [RFC4960] and DCCP [RFC4340], as well as for individual subflows in Multipath TCP [RFC6824].

2. Conventions used in this document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

In this document, these words will appear with that interpretation only when in ALL CAPS. Lower case uses of these words are not to be interpreted as carrying significance described in RFC 2119.

In this document, the characters ">>" preceding an indented line(s) indicates a statement using the key words listed above. This convention aids reviewers in quickly identifying or finding the portions of this RFC covered by these keywords.
3. Terminology

Host - a source or sink of TCP segments associated with a single IP address

Host-pair - a pair of hosts and their corresponding IP addresses

Path - an Internet path between the IP addresses of two hosts

4. The TCP Control Block (TCB)

A TCB describes the data associated with each connection, i.e., with each association of a pair of applications across the network. The TCB contains at least the following information [RFC793]:

- Local process state
  - pointers to send and receive buffers
  - pointers to retransmission queue and current segment
  - pointers to Internet Protocol (IP) PCB

- Per-connection shared state
  - macro-state
    - connection state
    - timers
    - flags
    - local and remote host numbers and ports
  - TCP option state
  - micro-state
    - send and receive window state (size*, current number)
    - round-trip time and variance
    - cong. window size (snd_cwnd)*
    - cong. window size threshold (ssthresh)*
    - max window size seen*
    - sendMSS#
    - MMS_S#
    - MMS_R#
    - PMTU#
    - round-trip time and variance#

The per-connection information is shown as split into macro-state and micro-state, terminology borrowed from [Co91]. Macro-state describes the finite state machine; we include the endpoint numbers and components (timers, flags) used to help maintain that state. Macro-state describes the protocol for establishing and maintaining shared state about the connection. Micro-state describes the protocol after a connection has been established, to maintain the reliability and congestion control of the data transferred in the connection.
We further distinguish two other classes of shared micro-state that are associated more with host-pairs than with application pairs. One class is clearly host-pair dependent (#, e.g., MSS, MMS, PMTU, RTT), and the other is host-pair dependent in its aggregate (*, e.g., congestion window information, current window sizes, etc.).

5. TCB Interdependence

There are two cases of TCB interdependence. Temporal sharing occurs when the TCB of an earlier (now CLOSED) connection to a host is used to initialize some parameters of a new connection to that same host, i.e., in sequence. Ensemble sharing occurs when a currently active connection to a host is used to initialize another (concurrent) connection to that host.

6. An Example of Temporal Sharing

The TCB data cache is accessed in two ways: it is read to initialize new TCBs and written when more current per-host state is available. New TCBs are initialized using context from past connections as follows:

<table>
<thead>
<tr>
<th>Safe?</th>
<th>Cached TCB</th>
<th>New TCB</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>old_MMS_S</td>
<td>old_MMS_S or not cached</td>
</tr>
<tr>
<td>yes</td>
<td>old_MMS_R</td>
<td>old_MMS_R or not cached</td>
</tr>
<tr>
<td>yes</td>
<td>old_sendMSS</td>
<td>old_sendMSS</td>
</tr>
<tr>
<td>yes</td>
<td>old_PMTU</td>
<td>old_PMTU</td>
</tr>
<tr>
<td>TBD</td>
<td>old_RTT</td>
<td>old_RTT</td>
</tr>
<tr>
<td>TBD</td>
<td>old_RTTvar</td>
<td>old_RTTvar</td>
</tr>
<tr>
<td>varies</td>
<td>old_option</td>
<td>(option specific)</td>
</tr>
<tr>
<td>TBD</td>
<td>old_ssthresh</td>
<td>old_ssthresh</td>
</tr>
<tr>
<td>TBD</td>
<td>old_snd_cwnd</td>
<td>old_snd_cwnd</td>
</tr>
</tbody>
</table>

Table entries indicate which are considered to be safe to share temporally. The other entries are discussed in section 8.
Most cached TCB values are updated when a connection closes. The exceptions are MMS_R and MMS_S, which are reported by IP [RFC1122], PMTU which is updated after Path MTU Discovery [RFC1191][RFC1981][RFC4821], and sendMSS, which is updated if the MSS option is received in the TCP SYN header.

Sharing sendMSS information affects only data in the SYN of the next connection, because sendMSS information is typically included in most TCP SYN segments. Caching PMTU can accelerate the efficiency of PMTUD, but can also result in black-holing until corrected if in error. Caching MMS_R and MMS_S may be of little direct value as they are reported by the local IP stack anyway.

[TBD - complete this section with details for TFO and other options whose state may, must, or must not be shared] The way in which other TCP option state can be shared depends on the details of that option. E.g., TFO state includes the TCP Fast Open Cookie [RFC7413] or, in case TFO fails, a negative TCP Fast Open response (from [RFC 7413]: "The client MUST cache negative responses from the server in order to avoid potential connection failures. Negative responses include the server not acknowledging the data in the SYN, ICMP error messages, and (most importantly) no response (SYN-ACK) from the server at all, i.e., connection timeout."). TFOinfo is cached when a connection is established.

Other TCP option state might not be as readily cached. E.g., TCP-AO [RFC5925] success or failure between a host pair for a single SYN destination port might be usefully cached. TCP-AO success or failure to other SYN destination ports on that host pair is never useful to cache because TCP-AO security parameters can vary per service.

The table below gives an overview of option-specific information that is considered safe to share.

<table>
<thead>
<tr>
<th>TEMPORAL SHARING - Option info</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cached</td>
</tr>
<tr>
<td>---------------------------</td>
</tr>
<tr>
<td>old_TFO_Cookie</td>
</tr>
<tr>
<td>old_TFO_Failure</td>
</tr>
</tbody>
</table>
### TEMPORAL SHARING - Cache Updates

<table>
<thead>
<tr>
<th>Safe?</th>
<th>Cached TCB</th>
<th>Current TCB</th>
<th>when?</th>
<th>New Cached TCB</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>old_MMS_S</td>
<td>curr_MMS_S</td>
<td>OPEN</td>
<td>curr_MMS_S</td>
</tr>
<tr>
<td>yes</td>
<td>old_MMS_R</td>
<td>curr_MMS_R</td>
<td>OPEN</td>
<td>curr_MMS_R</td>
</tr>
<tr>
<td>yes</td>
<td>old_sendMSS</td>
<td>curr_sendMSS</td>
<td>MSSopt</td>
<td>curr_sendMSS</td>
</tr>
<tr>
<td>yes</td>
<td>old_PMTU</td>
<td>curr_PMTU</td>
<td>PMTUD</td>
<td>curr_PMTU</td>
</tr>
<tr>
<td>TBD</td>
<td>old_RTT</td>
<td>curr_RTT</td>
<td>CLOSE</td>
<td>merge(curr, old)</td>
</tr>
<tr>
<td>TBD</td>
<td>old_RTTvar</td>
<td>curr_RTTvar</td>
<td>CLOSE</td>
<td>merge(curr, old)</td>
</tr>
<tr>
<td>varies</td>
<td>old_option</td>
<td>curr_option</td>
<td>ESTAB</td>
<td>(depends on option)</td>
</tr>
<tr>
<td>TBD</td>
<td>old_ssthresh</td>
<td>curr_ssthresh</td>
<td>CLOSE</td>
<td>merge(curr, old)</td>
</tr>
<tr>
<td>TBD</td>
<td>old_snd_cwnd</td>
<td>curr_snd_cwnd</td>
<td>CLOSE</td>
<td>merge(curr, old)</td>
</tr>
</tbody>
</table>

Caching PMTU and sendMSS is trivial; reported values are cached, and the most recent values are used. The cache is updated when the MSS option is received in a SYN or after PMTUD (i.e., when an ICMPv4 Fragmentation Needed [RFC1191] or ICMPv6 Packet Too Big message is received [RFC1981] or the equivalent is inferred, e.g., as from PLPMTUD [RFC4821]), respectively, so the cache always has the most recent values from any connection. For sendMSS, the cache is consulted only at connection establishment and not otherwise updated, which means that MSS options do not affect current connections. The default sendMSS is never saved; only reported MSS values update the cache, so an explicit override is required to reduce the sendMSS. There is no particular benefit to caching MMS_S and MMS_R as these are reported by the local IP stack.

TCP options are copied or merged depending on the details of each option. E.g., TFO state is updated when a connection is established and read before establishing a new connection.

RTT values are updated by a more complicated mechanism [RFC1644][Ja86]. Dynamic RTT estimation requires a sequence of RTT measurements. As a result, the cached RTT (and its variance) is an average of its previous value with the contents of the currently active TCB for that host, when a TCB is closed. RTT values are updated only when a connection is closed. The method for merging old and current values needs to attempt to reduce the transient for new
connections. [THESE MERGE FUNCTIONS NEED TO BE SPECIFIED, considering e.g. [DM16] - TBD].

The updates for RTT, RTTvar and ssthresh rely on existing information, i.e., old values. Should no such values exist, the current values are cached instead.

**TEMPORAL SHARING - Option info Updates**

<table>
<thead>
<tr>
<th>Cached</th>
<th>Current</th>
<th>when?</th>
<th>New Cached</th>
</tr>
</thead>
<tbody>
<tr>
<td>old_TFO_Cookie</td>
<td>old_TFO_Cookie</td>
<td>ESTAB</td>
<td>old_TFO_Cookie</td>
</tr>
<tr>
<td>old_TFO_Failure</td>
<td>old_TFO_Failure</td>
<td>ESTAB</td>
<td>old_TFO_Failure</td>
</tr>
</tbody>
</table>

7. An Example of Ensemble Sharing

Sharing cached TCB data across concurrent connections requires attention to the aggregate nature of some of the shared state. For example, although MSS and RTT values can be shared by copying, it may not be appropriate to copy congestion window or ssthresh information (see section 8 for a discussion of congestion window or ssthresh sharing).

**ENSEMBLE SHARING - TCB Initialization**

<table>
<thead>
<tr>
<th>Safe?</th>
<th>Cached TCB</th>
<th>New TCB</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>old_MMS_S</td>
<td>old_MMS_S</td>
</tr>
<tr>
<td>yes</td>
<td>old_MMS_R</td>
<td>old_MMS_R</td>
</tr>
<tr>
<td>yes</td>
<td>old_sendMSS</td>
<td>old_sendMSS</td>
</tr>
<tr>
<td>yes</td>
<td>old_PMTU</td>
<td>old_PMTU</td>
</tr>
<tr>
<td>TBD</td>
<td>old_RTT</td>
<td>old_RTT</td>
</tr>
<tr>
<td>TBD</td>
<td>old_RTTvar</td>
<td>old_RTTvar</td>
</tr>
<tr>
<td>TBD</td>
<td>old_option</td>
<td>(option-specific)</td>
</tr>
</tbody>
</table>

Table entries indicate which are considered to be safe to share across an ensemble. The other entries are discussed in section 8.
The table below gives an overview of option-specific information that is considered safe to share.

### ENSEMBLE SHARING - Option info

<table>
<thead>
<tr>
<th>Cached</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>old_TFO_Cookie</td>
<td>old_TFO_Cookie</td>
</tr>
<tr>
<td>old_TFO_Failure</td>
<td>old_TFO_Failure</td>
</tr>
</tbody>
</table>

### ENSEMBLE SHARING - Cache Updates

<table>
<thead>
<tr>
<th>Safe?</th>
<th>Cached TCB</th>
<th>Current TCB</th>
<th>when?</th>
<th>New Cached TCB</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>old_MMS_S</td>
<td>curr_MMS_S</td>
<td>OPEN</td>
<td>curr_MMS_S</td>
</tr>
<tr>
<td>yes</td>
<td>old_MMS_R</td>
<td>curr_MMS_R</td>
<td>OPEN</td>
<td>curr_MMS_R</td>
</tr>
<tr>
<td>yes</td>
<td>old_sendMSS</td>
<td>curr_sendMSS</td>
<td>MSSopt</td>
<td>curr_sendMSS</td>
</tr>
<tr>
<td>yes</td>
<td>old_PMTU</td>
<td>curr_PMTU</td>
<td>PMTUD</td>
<td>curr_PMTU</td>
</tr>
<tr>
<td></td>
<td>TBA</td>
<td>curr_RTTvar</td>
<td>update</td>
<td>rtt_update(old,cur)</td>
</tr>
<tr>
<td>varies</td>
<td>old_option</td>
<td>curr_option</td>
<td>(depends)</td>
<td>(option specific)</td>
</tr>
</tbody>
</table>

For ensemble sharing, TCB information should be cached as early as possible, sometimes before a connection is closed. Otherwise, opening multiple concurrent connections may not result in TCB data sharing if no connection closes before others open. The amount of work involved in updating the aggregate average should be minimized, but the resulting value should be equivalent to having all values measured within a single connection. The function "rtt_update" in the ensemble sharing table indicates this operation, which occurs whenever the RTT would have been updated in the individual TCP connection. As a result, the cache contains the shared RTT variables, which no longer need to reside in the TCB [Ja86].

Congestion window size and ssthresh aggregation are more complicated in the concurrent case. When there is an ensemble of connections, we...
need to decide how that ensemble would have shared these variables, in order to derive initial values for new TCBs.

<table>
<thead>
<tr>
<th>Cached</th>
<th>Current</th>
<th>when?</th>
<th>New Cached</th>
</tr>
</thead>
<tbody>
<tr>
<td>old_TFO_Cookie</td>
<td>old_TFO_Cookie</td>
<td>ESTAB</td>
<td>old_TFO_Cookie</td>
</tr>
<tr>
<td>old_TFO_Failure</td>
<td>old_TFO_Failure</td>
<td>ESTAB</td>
<td>old_TFO_Failure</td>
</tr>
</tbody>
</table>

Any assumption of this sharing can be incorrect, including this one, because identical endpoint address pairs may not share network paths. In current implementations, new congestion windows are set at an initial value of 4-10 segments [RFC3390][RFC6928], so that the sum of the current windows is increased for any new connection. This can have detrimental consequences where several connections share a highly congested link.

There are several ways to initialize the congestion window in a new TCB among an ensemble of current connections to a host, as shown below. Current TCP implementations initialize it to four segments as standard [rfc3390] and 10 segments experimentally [RFC6928] and T/TCP hinted that it should be initialized to the old window size [RFC1644]. In the former cases, the assumption is that new connections should behave as conservatively as possible. In the latter T/TCP case, no accommodation is made for concurrent aggregate behavior.

In either case, the sum of window sizes can increase, rather than remain constant. A different approach is to give each pending connection its "fair share" of the available congestion window, and let the connections balance from there. The assumption we make here is that new connections are implicit requests for an equal share of available link bandwidth, which should be granted at the expense of current connections. [TBD - a new method for safe congestion sharing will be described]

8. Compatibility Issues

For the congestion and current window information, the initial values computed by TCB interdependence may not be consistent with the long-term aggregate behavior of a set of concurrent connections between the same endpoints. Under conventional TCP congestion control, if a single existing connection has converged to a congestion window of 40 segments, two newly joining concurrent
connections assume initial windows of 10 segments [RFC6928], and the current connection’s window doesn’t decrease to accommodate this additional load and connections can mutually interfere. One example of this is seen on low-bandwidth, high-delay links, where concurrent connections supporting Web traffic can collide because their initial windows were too large, even when set at one segment.

[TBD - this paragraph needs to be revised based on new recommendations] Under TCB interdependence, all three connections could change to use a congestion window of 12 (rounded down to an even number from 13.33, i.e., 40/3). This would include both increasing the initial window of the new connections (vs. current recommendations [RFC6928]) and decreasing the congestion window of the current connection (from 40 down to 12). This gives the new connections a larger initial window than allowed by [RFC6928], but maintains the aggregate. Depending on whether the previous connections were in steady-state, this can result in more bursty behavior, e.g., when previous connections are idle and new connections commence with a large amount of available data to transmit. Additionally, reducing the congestion window of an existing connection needs to account for the number of packets that are already in flight.

Because this proposal attempts to anticipate the aggregate steady-state values of TCB state among a group or over time, it should avoid the transient effects of new connections. In addition, because it considers the ensemble and temporal properties of those aggregates, it should also prevent the transients of short-lived or multiple concurrent connections from adversely affecting the overall network performance. There have been ongoing analysis and experiments to validate these assumptions. For example, [Ph12] recommends to only cache ssthresh for temporal sharing when flows are long. Sharing ssthresh between short flows can deteriorate the overall performance of individual connections[Ph12, Nd16], although this may benefit overall network performance. [TBD - the details of this issue need to be summarized and clarified herein].

[TBD - placeholder for corresponding RTT discussion]

Due to mechanisms like ECMP and LAG [RFC7424], TCP connections sharing the same host-pair may not always share the same path. This does not matter for host-specific information such as RWIN and TCP option state, such as TFOinfo. When TCB information is shared across different SYN destination ports, path-related information can be incorrect; however, the impact of this error is potentially diminished if (as discussed here) TCB sharing affects only the transient event of a connection start or if TCB information is
shared only within connections to the same SYN destination port. In case of Temporal Sharing, TCB information could also become invalid over time. Because this is similar to the case when a connection becomes idle, mechanisms that address idle TCP connections (e.g., [RFC7661]) could also be applied to TCB cache management.

There may be additional considerations to the way in which TCB interdependence rebalances congestion feedback among the current connections, e.g., it may be appropriate to consider the impact of a connection being in Fast Recovery [RFC5861] or some other similar unusual feedback state, e.g., as inhibiting or affecting the calculations described herein.

TCP is sometimes used in situations where packets of the same host-pair always take the same path. Because ECMP and LAG examine TCP port numbers, they may not be supported when TCP segments are encapsulated, encrypted, or altered – for example, some Virtual Private Networks (VPNs) are known to use proprietary UDP encapsulation methods. Similarly, they cannot operate when the TCP header is encrypted, e.g., when using IPsec ESP. TCB interdependence among the entire set sharing the same endpoint IP addresses should work without problems under these circumstances. Moreover, measures to increase the probability that connections use the same path could be applied: e.g., the connections could be given the same IPv6 flow label. TCB interdependence can also be extended to sets of host IP address pairs that share the same network path conditions, such as when a group of addresses is on the same LAN (see Section 9).

It can be wrong to share TCB information between TCP connections on the same host as identified by the IP address if an IP address is assigned to a new host (e.g., IP address spinning, as is used by ISPs to inhibit running servers). It can be wrong if Network Address (and Port) Translation (NA(P)T) [RFC2663] or any other IP sharing mechanism is used. Such mechanisms are less likely to be used with IPv6. Other methods to identify a host could also be considered to make correct TCB sharing more likely. Moreover, some TCB information is about dominant path properties rather than the specific host. IP addresses may differ, yet the relevant part of the path may be the same.

9. Implications

There are several implications to incorporating TCB interdependence in TCP implementations. First, it may reduce the need for application-layer multiplexing for performance enhancement [RFC7231]. Protocols like HTTP/2 [RFC7540] avoid connection reestablishment costs by serializing or multiplexing a set of per-
host connections across a single TCP connection. This avoids TCP’s per-connection OPEN handshake and also avoids recomputing MSS, RTT, and congestion windows. By avoiding the so-called, “slow-start restart,” performance can be optimized. TCB interdependence can provide the "slow-start restart avoidance" of multiplexing, without requiring a multiplexing mechanism at the application layer.

TCB interdependence pushes some of the TCP implementation from the traditional transport layer (in the ISO model), to the network layer. This acknowledges that some state is in fact per-host-pair or can be per-path as indicated solely by that host-pair. Transport protocols typically manage per-application-pair associations (per stream), and network protocols manage per-host-pair and path associations (routing). Round-trip time, MSS, and congestion information could be more appropriately handled in a network-layer fashion, aggregated among concurrent connections, and shared across connection instances [RFC3124].

An earlier version of RTT sharing suggested implementing RTT state at the IP layer, rather than at the TCP layer [Ja86]. Our observations are for sharing state among TCP connections, which avoids some of the difficulties in an IP-layer solution. One such problem is determining the associated prior outgoing packet for an incoming packet, to infer RTT from the exchange. Because RTTs are still determined inside the TCP layer, this is simpler than at the IP layer. This is a case where information should be computed at the transport layer, but could be shared at the network layer.

Per-host-pair associations are not the limit of these techniques. It is possible that TCBs could be similarly shared between hosts on a subnet or within a cluster, because the predominant path can be subnet-subnet, rather than host-host. Additionally, TCB interdependence can be applied to any protocol with congestion state, including SCTP [RFC4960] and DCCP [RFC4340], as well as for individual subflows in Multipath TCP [RFC6824].

There may be other information that can be shared between concurrent connections. For example, knowing that another connection has just tried to expand its window size and failed, a connection may not attempt to do the same for some period. The idea is that existing TCP implementations infer the behavior of all competing connections, including those within the same host or subnet. One possible optimization is to make that implicit feedback explicit, via extended information associated with the endpoint IP address and its TCP implementation, rather than per-connection state in the TCB.
Like its initial version in 1997, this document’s approach to TCB interdependence focuses on sharing a set of TCBs by updating the TCB state to reduce the impact of transients when connections begin or end. Other mechanisms have since been proposed to continuously share information between all ongoing communication (including connectionless protocols), updating the congestion state during any congestion-related event (e.g., timeout, loss confirmation, etc.) [RFC3124]. By dealing exclusively with transients, TCB interdependence is more likely to exhibit the same behavior as unmodified, independent TCP connections.

10. Implementation Observations

The observation that some TCB state is host-pair specific rather than application-pair dependent is not new and is a common engineering decision in layered protocol implementations. A discussion of sharing RTT information among protocols layered over IP, including UDP and TCP, occurred in [Ja86]. Although now deprecated, T/TCP was the first to propose using caches in order to maintain TCB states (see Appendix A for more information).

The table below describes the current implementation status for some TCB information in Linux kernel version 4.6, FreeBSD 10 and Windows (as of October 2016). In the table, "shared" only refers to temporal sharing.
TCB data                      Status
-----------------------------------------------------------
old MMS_S                     Not shared
old MMS_R                     Not shared
old_sendMSS                   Cached and shared in Linux (MSS)
old PMTU                      Cached and shared in FreeBSD and Windows (PMTU)
old_RTT                      Cached and shared in FreeBSD and Linux
old_RTTvar                    Cached and shared in FreeBSD
old TFOinfo                   Cached and shared in Linux and Windows
old_snd_cwnd                  Not shared
old_ssthresh                  Cached and shared in FreeBSD and Linux:
                                FreeBSD: arithmetic
                                mean of ssthresh and previous value if
                                a previous value exists;
                                Linux: depending on state,
                                max(cwnd/2, ssthresh) in most cases

11. Security Considerations

These suggested implementation enhancements do not have additional ramifications for explicit attacks. These enhancements may be susceptible to denial-of-service attacks if not otherwise secured. For example, an application can open a connection and set its window size to zero, denying service to any other subsequent connection between those hosts.

TCB sharing may be susceptible to denial-of-service attacks, wherever the TCB is shared, between connections in a single host, or between hosts if TCB sharing is implemented within a subnet (see Implications section). Some shared TCB parameters are used only to create new TCBs, others are shared among the TCBs of ongoing connections. New connections can join the ongoing set, e.g., to optimize send window size among a set of connections to the same host.

Attacks on parameters used only for initialization affect only the transient performance of a TCP connection. For short connections, the performance ramification can approach that of a denial-of-
service attack. E.g., if an application changes its TCB to have a false and small window size, subsequent connections would experience performance degradation until their window grew appropriately.

The solution is to limit the effect of compromised TCB values. TCBs are compromised when they are modified directly by an application or transmitted between hosts via unauthenticated means (e.g., by using a dirty flag). TCBs that are not compromised by application modification do not have any unique security ramifications. Note that the proposed parameters for TCB sharing are not currently modifiable by an application.

All shared TCBs MUST be validated against default minimum parameters before used for new connections. This validation would not impact performance, because it occurs only at TCB initialization. This limits the effect of attacks on new connections to reducing the benefit of TCB sharing, resulting in the current default TCP performance. For ongoing connections, the effect of incoming packets on shared information should be both limited and validated against constraints before use. This is a beneficial precaution for existing TCP implementations as well.

TCBs modified by an application SHOULD NOT be shared, unless the new connection sharing the compromised information has been given explicit permission to use such information by the connection API. No mechanism for that indication currently exists, but it could be supported by an augmented API. This sharing restriction SHOULD be implemented in both the host and the subnet. Sharing on a subnet SHOULD utilize authentication to prevent undetected tampering of shared TCB parameters. These restrictions limit the security impact of modified TCBs both for connection initialization and for ongoing connections.

Finally, shared values MUST be limited to performance factors only. Other information, such as TCP sequence numbers, when shared, are already known to compromise security.

12. IANA Considerations

There are no IANA implications or requests in this document.

This section should be removed upon final publication as an RFC.
13. References

13.1. Normative References


13.2. Informative References


[Ja86] Jacobson, V., (mail to public list "tcp-ip", no archive found), 1986.


14. Acknowledgments

The authors would like to thank for Praveen Balasubramanian for information regarding TCB sharing in Windows, and Yuchung Cheng, Lars Eggert, Ilpo Jarvinen and Michael Scharf for comments on earlier versions of the draft. This work has received funding from a collaborative research project between the University of Oslo and Huawei Technologies Co., Ltd., and is partly supported by USC/ISI’s Postel Center.

This document was prepared using 2-Word-v2.0.template.dot.

15. Change log

from -01 to -02:

- Stated that our OS implementation overview table only covers temporal sharing.

- Correctly reflected sharing of old_RTT in Linux in the implementation overview table.

- Marked entries that are considered safe to share with an asterisk (suggestion was to split the table)

- Discussed correct host identification: NATs may make IP addresses the wrong input, could e.g. use HTTP cookie.

- Included MMS_S and MMS_R from RFC1122; fixed the use of MSS and MTU
- Added information about option sharing, listed options in the appendix

Authors’ Addresses

Joe Touch
USC/ISI
4676 Admiralty Way
Marina del Rey, CA 90292-6695
USA
Phone: +1 (310) 448-9151
Email: touch@isi.edu

Michael Welzl
University of Oslo
PO Box 1080 Blindern
Oslo  N-0316
Norway
Phone: +47 22 85 24 20
Email: michawe@ifi.uio.no

Safiqul Islam
University of Oslo
PO Box 1080 Blindern
Oslo  N-0316
Norway

Phone: +47 22 84 08 37
Email: safiquli@ifi.uio.no
16. Appendix A: TCB sharing history

T/TCP proposed using caches to maintain TCB information across instances (temporal sharing), e.g., smoothed RTT, RTT variance, congestion avoidance threshold, and MSS [RFC1644]. These values were in addition to connection counts used by T/TCP to accelerate data delivery prior to the full three-way handshake during an OPEN. The goal was to aggregate TCB components where they reflect one association - that of the host-pair, rather than artificially separating those components by connection.

At least one T/TCP implementation saved the MSS and aggregated the RTT parameters across multiple connections, but omitted caching the congestion window information [Br94], as originally specified in [RFC1379]. Some T/TCP implementations immediately updated MSS when the TCP MSS header option was received [Br94], although this was not addressed specifically in the concepts or functional specification [RFC1379][RFC1644]. In later T/TCP implementations, RTT values were updated only after a CLOSE, which does not benefit concurrent sessions.

Temporal sharing of cached TCB data was originally implemented in the SunOS 4.1.3 T/TCP extensions [Br94] and the FreeBSD port of same [FreeBSD]. As mentioned before, only the MSS and RTT parameters were cached, as originally specified in [RFC1379]. Later discussion of T/TCP suggested including congestion control parameters in this cache [RFC1644].

17. Appendix B: Options

In addition to the options that can be cached and shared, this memo also lists all options for which state should *not* be kept. This list is meant to avoid work duplication and should be removed upon publication.
Obsolete (MUST NOT keep state):

ECHO
ECHO REPLY
PO Conn permitted
PO service profile
CC
CC.NEW
CC.ECHO
Alt CS req
Alt CS data

No state to keep:

EOL
NOP
WS
SACK
TS
MD5
TCP-AO
EXP1
EXP2

MUST NOT keep state:
Skeeter (DH exchange – might be obsolete, though)

Bubba (DH exchange – might really be obsolete, though)

Trailer CS

SCPS capabilities

S-NACK

Records boundaries

Corruption experienced

SNAP

TCP Compression

Quickstart response

UTO

MPTCP (can we cache when this fails?)

TFO success

MAY keep state:

MSS

TFO failure (so we don’t try again, since it’s optional)

MUST keep state:

TFP cookie (if TFO succeeded in the past)