TCP DA-LBE

A Meta Congestion Controller for Deadline-Aware Less than Best Effort Delivery in the Linux Operating System

Henning Parratt Tandberg

Thesis submitted for the degree of
Master in Distributed Systems and Networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2020

TCP DA-LBE

A Meta Congestion Controller for Deadline-Aware Less than Best Effort Delivery in the Linux Operating System

Henning Parratt Tandberg

© 2020 Henning Parratt Tandberg

TCP DA-LBE

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Previous efforts on implementing Deadline Aware Less-than Best Effort (DA-LBE) services have provided valuable insight into the topic of DA-LBE with some very promising results. However, lacking from these previous attempts is a stable, long-term solution for providing such services to the users.

This thesis introduces TCP-Dalbe; a meta congestion controller for the Linux Operating System, implemented as a loadable kernel module. Meta congestion controllers collect various statistics from the network, which are used to adapt the underlying congestion controller, altering the aggressiveness of the traffic source. TCP-Dalbe introduces support for adapting TCP-Vegas as the underlying congestion controller, using both Model Based Control and Proportional Integral Differential Control.

Through a series of experiments in an emulated environment, TCP-Dalbe was evaluated in terms of achieving its desired Less-than Best Effort characteristics, what impact the use of fixed point numbers have on the quality of the DA-LBE calculations, and how much function overhead and memory usage the implementation introduces in relation to its underlying congestion controller.

These evaluations demonstrated that the implementation in fact satisfies the criteria for a DA-LBE traffic source, and that the use of fixed point operations introduced some error to the calculations, however, the overall error was still below 1%, which was not enough to have a major impact of the quality of the DA-LBE calculations.

Furthermore, it was demonstrated that the introduced function overhead may be quite large at times, reaching 70% to 80% at times. However, when investigating the overall time spent in each function, these overheads became insignificant, as they were only responsible for less than 1% of the entire time spent in the connection.

Acknowledgments

I would like to express my sincere gratitude to David Hayes, David Ros, and Özgü Alay at Simula Research Laboratories, for the great support and supervision while working on this thesis.

To my friends and family for always being so supportive of me. To Mattis, for the good times we had while working on our theses, and to my brother, Eilif, for the immense help when my mathematical knowledge did not suffice.

Finally, to my better half and best friend, Kathrine, for always being there for me, through thick and thin.

Contents

1 Introduction **1**

1.1 Problem Statement . . . 3
1.2 Research Questions . . . 3
1.3 Previous Work on the Implementation . . . 3
1.4 Structure . . . 4

2 Background **6**

2.1 The Internet . . . 6
2.1.1 Architectural Assumptions . . . 7
2.1.2 Internet Protocol . . . 7
2.1.3 Scalability . . . 9
2.2 Transmission Control Protocol . . . 9
2.3 Congestion Control . . . 10
2.3.1 Loss-Based Congestion Control . . . 10
2.3.2 Delay Based Congestion Control . . . 14
2.3.3 Explicit Congestion Notification . . . 16
2.3.4 Fairness . . . 16
2.4 Less-than Best Effort Delivery . . . 17
2.5 Deadline Aware Less-than Best Effort Delivery . . . 17
2.5.1 Measuring the Price of Congestion . . . 17
2.5.2 Adapting The Weight . . . 18
2.5.3 Meta Congestion Control . . . 21
2.6 The Linux Operating System . . . 22
2.6.1 Versioning . . . 23
2.6.2 Contributing . . . 23
2.6.3 Loadable Kernel Modules . . . 24
2.6.4 TCP/IP Stack in Linux . . . 25
2.6.5 TCP Congestion Control in Linux . . . 26
2.6.6 Floating Point Operations in the Kernel . . . 29
2.7 Summary . . . 29

3 Methodology **31**
3.1 Approach . . . 31
3.1.1 Planning Phase . . . 31
3.1.2 Design Phase . . . 32
3.1.3 Implementation Phase . . . 32
3.1.4 Testing and Experimentation Phase . . . 33
3.2 Collaboration . . . 34
3.3 Tools . . . 34
3.3.1 Common Open Research Emulator . . . 34
3.3.2 Linux Performance Events . . . 35
3.3.3 Function Tracer . . . 35
3.4 Summary . . . 36

4 Design and Implementation **37**
4.1 Requirements for the Implementation . . . 37
4.2 Code Convention . . . 37
4.3 Architectural Decisions . . . 38
4.3.1 Architecture . . . 38
4.3.2 Configuration Possibilities . . . 41
4.3.3 Fixed Point Operations . . . 44
4.4 Changes to the Kernel . . . 47
4.5 Usage . . . 48
4.6 Testing . . . 48
4.6.1 Unit Tests . . . 49
4.7 Debugging . . . 51
4.7.1 Debugging by Logging . . . 51
4.7.2 Debugging Memory Usage . . . 52
4.7.3 Debugging Kernel Panics . . . 54
4.8 Error Handling . . . 54
4.9 Licensing . . . 54
4.10 Shortcomings . . . 55
4.10.1 Available Model Based Controllers . . . 55
4.10.2 Support for Loss-Based Congestion Control . . . 55
4.10.3 Modular Architecture . . . 55
4.10.4 Metadata from Underlying Congestion Controllers . . . 56
4.10.5 Current use of Fixed Point Types . . . 56
4.10.6 Passing Fixed Point Values From User Space to Kernel Space . . . 56
4.11 Summary . . . 56

5 Test Environment **58**
5.1 Requirements . . . 58
5.2 Collaboration . . . 59
5.3 Testing on Virtual Machines . . . 59

5.3.1 Unit Testing with Virtual Machines 59
5.3.2 Debugging with Virtual Machines 60
5.3.3 Drawback of Testing on Virtual Machines 60
5.4 Testing on Hardware 61
5.4.1 Building a Suitable Test Bed 61
5.4.2 Hardware Setup 63
5.4.3 Operating System Setup 64
5.4.4 Defining a Stable Test Environment 65
5.4.5 Network Emulation 67
5.4.6 Verifying the Performance of the Test Environment . . 69
5.5 Experiment Orchestration 69
5.5.1 Experiment Execution 71
5.5.2 Data Collection 71
5.5.3 Data Processing 72
5.6 Summary 72

6 Network Performance Experiments 73
6.1 Requirements 73
6.2 Network Efficiency 74
6.2.1 Setup 74
6.2.2 Expectations 75
6.2.3 Results for Model Based Control for Vegas 76
6.2.4 Fixed Point Precision 81
6.3 Fairness and Completion Times 82
6.3.1 Setup 83
6.3.2 Fairness 83
6.3.3 Completion Times 87
6.4 Summary 94

7 Load and Overhead Experiments 96
7.1 Requirements for Load and Overhead Experiments 96
7.2 Memory Usage 97
7.2.1 Reasoning about Memory Usage 97
7.3 Function Frequency 98
7.3.1 Setup 98
7.3.2 Results 98
7.4 Function Overhead 100
7.4.1 Setup 100
7.4.2 Expectations 103
7.4.3 Results 104
7.5 Summary 109

8 Conclusion **110**
8.1 Future Work . . . 111
8.1.1 Improvements to the Meta Congestion Controller . . . 111
8.1.2 Improvements to Testing and Experimentation . . . 112

A Architecture and Internals **121**
A.1 Pluggable Congestion Controller Interface . . . 121
A.1.1 Initialize Private Data . . . 121
A.1.2 Cleanup Private Data . . . 121
A.1.3 Calculate New Slow Start Threshold . . . 122
A.1.4 Inform About State Change . . . 122
A.1.5 Calculate New Congestion Window . . . 122
A.1.6 Inform About New Congestion Event . . . 123
A.1.7 Upon Arrival of an ACK . . . 123
A.1.8 Calculate New Window in the Event of Loss . . . 123
A.1.9 Packet Accounting in the Event of an ACK . . . 124
A.1.10 Get Information About the Congestion Controller . . . 125
A.1.11 Set Custom Socket Options . . . 125
A.1.12 Get Custom Socket Options . . . 125
A.2 DALBE Math . . . 126
A.2.1 Multiplication between two unsigned fixed point numbers . . . 126
A.2.2 Multiplication between two signed fixed point numbers 127
A.2.3 Division between two unsigned fixed point numbers . 128
A.2.4 Division between two unsigned fixed point numbers . 129
A.2.5 Support Macros for Fixed Point Operations . . . 129
A.3 Pluggable Congestion Control Architecture . . . 132

Appendices **121**

B Parameters and Socket Options **137**
B.1 Module Parameters . . . 137
B.1.1 Module Parameters . . . 137
B.2 Custom Socket Options . . . 138
B.2.1 Custom Socket Options . . . 138

C Source Code and Raw Data **139**
C.1 Source Code . . . 139
C.1.1 mosaic-students-henning . . . 139
C.1.2 TestBed . . . 139
C.1.3 tcp-dalbe-test . . . 139
C.1.4 tcp-dalbe-analysis . . . 139
C.2 Raw Data . . . 139

D Documentation **140**
D.1 Meta Congestion Controller Documentation 140
D.2 Test Environment Documentation 147
D.3 Unit Tests and Benchmarks Documentation 158

List of Figures

2.1 A diagram of the TCP/IP Stack and how data flows between the different layers. 8

2.2 A simplified model showing two examples of how packets traverse the networking stack in Linux when TCP and IPv4 are used as the transport- and network layer protocols. Sub-figure 2.2a shows the major functions which the data is passed through from leaving the Application layer, before entering the network. Similarly, sub-figure 2.2b shows the major functions which the data pass through before arriving at the Application layer. 27

3.1 A simple model of the work flow used while working on this project. This was a simple adaptation of a typical agile / test driven development work flow, and even though the arrows are pointing in one direction it did not strictly mean that this was the order of that always followed. 32

4.1 A listing of the functions implemented by the meta congestion controller from the Transmission Control Protocol (TCP) pluggable congestion controller interface. 39

4.2 A model of how the meta congestion controller fits into the transport layer of the networking stack in Linux. Each time there is an outgoing or incoming packet, it may trigger one of the meta congestion controller functions. If this is the case, the function may manipulate the TCP socket directly, either before or after invoking the underlying congestion controller. The underlying congestion controller may in turn also manipulate the TCP socket before returning to its caller; the meta congestion controller. 40

4.3 This model illustrates how the meta congestion controller is registered in the kernel as a congestion module through the *dalbe_register* function, how the meta congestion controller is able to fetch a pointer to the underlying congestion controller through the *tcp_ca_find* function call, and how the TCP-DALBE (Dalbe) structure is stored in the hashmap using *hash_add*. 42
4.4 This model illustrates an example of what a call to one of the Dalbe meta congestion controller functions involves, in this case the *dalbe_cong_avoid* function. The first action performed is to fetch the *dalbe struct* from the hash map. If this is successful, the meta congestion module may perform some calculations before invoking the corresponding function of the underlying congestion controller. Upon return from the underlying congestion controller, the meta congestion controller may perform some additional calculations before returning. 43
4.5 A listing showing the difference between a GCC macro based multiplication operation for fixed point numbers, and the same code written with inline functions and an additional line of complexity. 46
4.6 A simple example of a client written in C. The client utilizes the socket option interface (*setsockopt*) to configure the connection for Deadline-Aware Less-than Best Effort (DA-LBE) transfers. 49
4.7 A simple example of a server written in C. The server utilizes the socket option interface (*setsockopt*) to configure the connection for DA-LBE transfers. Note that this is done on the client socket, after the connection with the client has been accepted. 50
4.8 An example of the graphs used for debugging. Each subgraph displays a certain set of metrics from a DA-LBE flow in a network performance related experiment. 53

5.1 An illustration of how the network was set up between the host machine and the virtual machines. 60
5.2 An example of how badly the virtual test bed performed when used to run network performance experiments. This was for a TCP-Vegas (Vegas) based DA-LBE flow using Proportional-Integral-Differential (PID) as the weight policy. 61

5.3 An image of our test bed fully assembled. The large black computer is the router machine. Below the white graphics card are two Network Interface Controller (NIC)s which were used to connect everything together. Mounted on top of the black case are the edge nodes, and below them is the switch used to forward network traffic from the control ports of the edge nodes through the router node. 62
5.4 A figure showing how our hardware test bed was connected. The orange lines show how the edge nodes were connected to the Internet via the router node. 63
5.5 A very simple dumbbell topology consisting of five nodes; four edge nodes, two on each side of the router node. 63
5.6 A more advanced figure of the dumbbell topology, showing how the WEST and EAST networks were set up. The two networks are connected by a 100 Mbps, 30 ms propagation delay link. 67
5.7 A screenshot of CORE GUI in action. This displays how we could easily, visually set up our network typologies. 68
5.8 An advanced model of how the dumbbell topology was set up on the hardware based test bed. All ingress traffic, may it be EAST or WEST bound, passes through an Intermediate Functional Block (IFB). The IFB is responsible for applying the HFSC and PFIFO qdiscs, before passing the traffic on to the outgoing NIC. The outgoing NIC applies a 15 ms delay. . 69
5.9 The figures show the average load and the memory usage on each edge node, taken from one for the network performance experiments in chapter 6. This was use to monitor the edge node during experimentation and testing. 70

6.1 Timeline of the network efficiency experiment showing when the different flows start and stop. 74
6.2 One second averages of the throughput for each flow during the network efficiency experiment. The DA-LBE flow uses Vegas as the underlying congestion controller with Model-Based-Control (MBC) as the weight adjustment policy. 76
6.3 Debug graphs produced for the network efficiency experiment for a Vegas based DA-LBE flow using MBC as the weight policy. 10 seconds increments. 78
6.4 One second averages of the throughput for each flow during the network efficiency experiment. The DA-LBE flow uses Vegas as the underlying congestion controller with PID as the weight adjustment policy. 79

6.5 Debug graphs produced for the network efficiency experiment for a Vegas based DA-LBE flow using PID as the weight policy. 10 seconds increments. 80
6.6 The relative error between the actual value of w, computed using *fixed point* operations, and the expected value of w, computed using *floating point* operations. 10 second increments. 82
6.7 Timeline of the fairness and completion times experiment showing when the different flows start and stop. 83
6.8 Fairness indexes for Vegas based DA-LBE flows using MBC as the weight policy. The box expends from the Q1 to Q3; the middle line represents the median (Q2); the whiskers extend to the $1.5 \times (Q3 - Q1)$; and outliers are represented as dots beyond the whiskers. 86
6.9 Fairness indexes for Vegas based DA-LBE flows using PID as the weight policy. The box expends from the Q1 to Q3; the middle line represents the median (Q2); the whiskers extend to the $1.5 \times (Q3 - Q1)$; and outliers are represented as dots beyond the whiskers. 88
6.10 Completion times for Vegas based DA-LBE flows using MBC and PID as the weight policies. The box expends from the Q1 to Q3; the middle line represents the median (Q2); the whiskers extend to the $1.5 \times (Q3 - Q1)$; and outliers are represented as dots beyond the whiskers. 89
6.11 One second averages of the throughput of the earliest completing run for the fairness and completion times experiment for a Vegas based DA-LBE flow using MBC as the weight policy. 90
6.12 One second averages of the throughput of the latest completing run for the fairness and completion times experiment for a Vegas based DA-LBE flow using MBC as the weight policy. . 91
6.13 Debug graphs produced for the latest completion run for the fairness and completion times experiment. A Vegas based DA-LBE flow using MBC as the weight policy. 10 seconds increments. 92
6.14 One second averages of the throughput of the earliest completing run for the fairness and completion times experiment for a Vegas based DA-LBE flow using PID as the weight policy. 93
6.15 Debug graphs produced for the latest completion run for the fairness and completion times experiment. A Vegas based DA-LBE flow using PID as the weight policy. 10 seconds increments. 93
6.16 One second averages of the throughput of the latest completing run for the fairness and completion times experiment for a Vegas based DA-LBE flow using PID as the weight policy. . 94

7.1 A set of figures that illustrate the function overhead introduced by the DA-LBE meta congestion controller, averaged over 50 iterations of an approximately 600 second experiment with one DA-LBE flow competing with a Best Effort (BE) flow. 105
7.2 Histogram showing the samples for the *pkts_acked* function. 108

A.1 Flow chart showing one possible scenario where *dalbe_cwnd_event* may be invoked during the TCP connection due to data being sent. 133
A.2 Flow chart showing another possible scenario where *dalbe_cwnd_event* may be invoked during the TCP connection due to data being sent. 134
A.3 A comprehensive flowchart showing what DA-LBE meta congestion controller functions may be invoked by the reception of an Acknowledgment (ACK). 135
A.4 Flow chart showing a possible scenario where both *dalbe_ssthresh* and *dalbe_cwnd_event* may be invoked during the TCP connection due to an Retransmission Timeout (RTO). 136

List of Tables

5.1 Relevant specifications of edge nodes. 64
5.2 Relevant specifications of router node. 64

6.1 Start and end times for each flow for the network efficiency experiment. The deadline for the DA-LBE flow is set to 1300 s, which means that it should finish close to t = 1700 s. 75
6.2 Start and end times for each flow for the fairness and completion times experiment.The deadline for the DA-LBE flow is set to 600 s, which means that it should finish close to t = 610 s. 83
6.3 Table of the parts used to create fairness indexes. 84
6.4 Table of values related to the fairness indexes for Vegas based DA-LBE flows using MBC as the weight policy. 87
6.5 Table of values related to the fairness indexes for Vegas based DA-LBE flows using PID as the weight policy. 87
6.6 Table of values related to the completion times for Vegas based DA-LBE flows using MBC and PID as the weight policies. 90

7.1 Table of the event counts produced by the function frequency experiment. 99
7.2 Table with the subset of trace events specified for *ftrace*, distinguishing between the most common and uncommon events. The table also shows the code complexity of each function. . 102
7.3 A table with values related to the average function overhead for both common and uncommon trace events. 106

Acronyms

ACK Acknowledgment

AIAD Additive Increase Additive Decrease

AIMD Additive Increase Multiplicative Decrease

API Application Programming Interface

AQM Active Queue Management

BDP Bandwidth Delay Product

BE Best Effort

BSD Berkeley Software Distribution

CC Congestion Control

CORE Common Open Research Emulator

CWND Congestion Window

DA-LBE Deadline-Aware Less-than Best Effort

DupACK Duplicate ACK

ECN Explicit Congestion Notification

FPU Floating Point Unit

FTP File Transfer Protocol

GPL General Purpose Licence

GSO Generic Segmentation Offload

GUI General User Interface

HFSC Hierarchical Fair Service Curve

HTB Hierarchy Token Bucket

IFB Intermediate Functional Block

IP Internet Protocol

LBE Less-than Best Effort

LEDBAT Low Extra Delay Background Transport

LKM Loadable Kernel Module

MBC Model-Based-Control

MPTCP Multipath TCP

MSS Maximum Segment Size

MTU Maximum Transmission Unit

NIC Network Interface Controller

NTP Network Time Protocol

NUM Network Utility Maximization

OS Operating System

OWD One-way Delay

PID Proportional-Integral-Differential

PPS Packets Per Second

PTP Precision Time Protocol

RAM Random Access Memory

RED Random Early Detection

RTO Retransmission Timeout

RTT Round Trip Time

RWND Receiver's Advertised Window

SSD Solid State Drive

SSH Secure Shell

SSTHRESH Slow Start Threshold

TCP Transmission Control Protocol

BIC TCP-BIC

Cubic TCP-Cubic

Dalbe TCP-DALBE

Reno TCP-Reno

Vegas TCP-Vegas

TSO TCP Segmentation Offload

UDP User Datagram Protocol

VM Virtual Machine

Chapter 1

Introduction

The Internet we know today is a large collection of networks of different sizes, connected by single or multiple paths. These networks are a set of nodes serving some kind of purpose for the end users of the Internet e.g., web servers, cloud storage, streaming services, etc. Many of these services are referred to as quality constrained and are highly dependent on a stable network that meets a set of requirements for reliable delivery, requiring low latency and/or high throughput. Such services often share the same network path as non-quality constrained services, that may consume large portions of the available capacity, and which may lead to the service degrading. However, many non-quality constrained services do not need to consume a large amount of the available bandwidth at all times. These services may be able to meet their requirements by only using bandwidth when it is available, allowing quality constrained traffic to be prioritized.

Less-than Best Effort (LBE) services, or so called *scavenger services*, have been introduced with this in mind. The goal of such a service is to use only the available network capacity and let other BE traffic finish without disruption. LBE has been in use for some time by providers like BitTorrent [62], Apple [42], and Microsoft [21], which all utilize Low Extra Delay Background Transport (LEDBAT) congestion control [66]. However LBE has one major drawback; there is no notion of completion time. This means that in a worst case scenario when there are many BE traffic sources on the same path as the LBE traffic source, the LBE traffic source may be too passive and possibly suffer from starvation.

A framework to solve this problem was introduced by D. Hayes et al. called Deadline-Aware Less-than Best Effort (DA-LBE) [35]. They showed that DA-LBE provides a valuable transport service for bulk data transfers such as backups. Allowing transfers to be completed by a soft deadline while keeping disruption of other traffic to a minimum.

Since the DA-LBE framework was introduced, there has been some work on implementing this for use in practice. A library for developing

DA-LBE transport services was introduced in a master thesis by H. Wallenburg [74] which relied on modified kernel by L. Storbukås [68]. The modified kernel provided mechanisms for gathering statistics and adjusting congestion signals for an arbitrary congestion controller. The library provided an Application Programming Interface (API) similar to that of Berkley sockets [50, *socket(2)*] and a daemon which performed the necessary meta congestion control operations on the underlying congestion control algorithm on the socket, asynchronously. These two implementations together allowed application developers a means to create, and experiment, with their own meta congestion controllers, and possibly providing a LBE or DA-LBE service to their applications.

In addition to this, these two implementations provided some very valuable insight into the development of the DA-LBE framework in practice. However, as as a long term solution for providing DA-LBE transport services to a system, these efforts may not be considered as acceptable. For one, the kernel modifications were implemented in such as way that getting them accepted into the Linux operating system would be very difficult [68], and though the library looked promising with some very interesting results on the topic of DA-LBE it was highly dependant on the non-submitted kernel modifications, making it hard to provide a maintainable, stable, long-term solution.

This thesis presents TCP-DALBE (Dalbe): a meta congestion controller for DA-LBE transport services implemented as a Linux kernel module [49]. The implementation makes an attempt to follow the standard way of implementing congestion control algorithms in the Linux kernel, as well as utilize the already existing socket API exposed to the users with as little modification as possible.

In addition to describing how the meta congestion controller was designed and implemented, I present how a test environment was built and configured specifically for testing the meta congestion controller. I conduct a series of experiments to show the meta congestion controller performs in an emulated network environment and analyze the additional memory usage and function overhead introduced by such an implementation. And, lastly discuss what this thesis contributes to within the field of DA-LBE, experiences and challenges that I faced along the way, and my suggestions for future work.

1.1 Problem Statement

Previous efforts on implementing DA-LBE services have provided valuable insight into the topic of DA-LBE with some very promising results. However, lacking from these previous attempts is a stable, long-term solution for providing such services to the users.

In this thesis I aim to produce as a stable, long-term solution for DA-LBE transport services, by creating an implementation for the Linux operating system.

1.2 Research Questions

To guide me in the correct direction of solving this problem I have defined a set of research questions which I intend to answer throughout this thesis.

RQ1: How should a meta congestion controller for DA-LBE transport services be designed and implemented for it to be considered a stable, long-term, solution suitable for the Linux operating system?

RQ2: How can a test environment be built and configured to be suitable for thorough testing and experimentation of the DA-LBE transport services?

RQ3: How should a DA-LBE transport service be tested and evaluated to verify its correctness and performance?

RQ4: How can the additional memory usage and computation load be evaluated for a DA-LBE meta congestion controller module?

1.3 Previous Work on the Implementation

This was initially a summer project started by D. Hayes and E. Band at Simula Research Laboratory[1] at Fornebu in Oslo, in 2018. The implementation had some of the functionality required for a DA-LBE transport service to work, however, it was lacking many important features, a good structure, and thorough testing. For this reason it is important to recognize, and acknowledge, the good work that was already put into this implementation, as I was fortunate enough to be able to picked up where the previous developers left off and use their implementation as the basis for what later evolved into Dalbe.

[1] https://www.simula.no/

1.4 Structure

This section provides a brief description of the main structure for each chapter in this thesis.

Chapter 2 - Background This chapter provides an introduction to the background material which is considered to be the essential building blocks for understanding the topics discussed in this thesis. Building up from the most basic on computer networking, to a more in depth look at some of the protocols used in the Internet, as well as an introduction to the Linux operating system with some implementation specific examples relevant for this thesis.

Chapter 3 - Methodology This chapter presents the methodology used in this thesis. The approach used for planning, designing, implementing and testing the DA-LBE transport services is explained in detail. In addition to this, some important tools and methods are presented, which have frequently been utilized for this work.

Chapter 4 - Design and Implementation This chapter dives into the details of designing and implementing the DA-LBE transport service. Some important design decisions are taken into consideration which define how the architecture, configurations options and arithmetic operations are designed and implemented. A short introduction of how the implementation may be used is presented in a server/client example, as well as a solution for how the DA-LBE transport service can be tested and debugged. Lastly, there is a discussion on the shortcomings of the implementation.

Chapter 5 - Test Environment This chapter presents how the virtual and hardware based test bed, used in this thesis, were built and configured. Details about the hardware components, the operating systems, and their configurations are discussed, as well as how the network topologies were emulated for our experiments. Lastly, a description of the software suite is given and how it is used for test orchestration, data collection and data parsing.

Chapter 6 - Network Performance Experiments This chapter presents a series of experiments which target the network performance, fixed point operations, network friendliness, and completion times of the DA-LBE transport service. Together with an analysis of the results produced from these experiments.

Chapter 7 - Load and Overhead Experiments This chapter presents a series of experiments which target internal memory usage, function frequency, and function overhead introduced by the DA-LBE transport service. Together with an analysis of the results produced from these experiments.

Chapter 8 - Conclusion The final chapter concludes the work by comparing the major findings to the problem statement for this thesis, in addition to a discussion on future work on the topic of DA-LBE transport services.

Chapter 2

Background

In this chapter I give a short introduction to the background material which the work in this thesis is based on. Building up from basic knowledge about computer networking, to a more in-depth description of some of the key protocols and frameworks. Provided in addition to this, is an introduction to the Linux Operating System (OS), with some implementation specifics related to its networking sub-system.

2.1 The Internet

Throughout the years we have grown used to always being connected. Wherever we are in the world, there is almost always a solution for connecting to the huge collection of networks we call "The Internet". Devices of all kinds are connected to the Internet. May it be a server farm hosted by some large firm, an individual person with their smart phone, or even everyday devices such as refrigerators and vacuum cleaners. In the 21st century it is so common to be connected, that we simply take the Internet for granted. We just expect it to always work.

This notion of simplicity has spread to the people who are constantly expanding it with applications and services,
namely, the software developers. For a software developer to create an *e.g.* Web application, the developer need little to no knowledge of the underlying complexity of the Internet. When the developer makes a request using a common web protocol such as HTTP, (s)he may just expect the data to effortlessly arrive for use in their application. When in fact the data has been through several protocols, and may even have been exposed to delays and even losses along the network path.

It may seem strange that a developer can be able to do so much with the network, yet have so little knowledge of it. However, this is an architectural decision made by the early developers of the Internet and is referred to as "separation of concerns".

2.1.1 Architectural Assumptions

The complexity of the Internet is divided into several layers forming a stack. This stack (see 2.1) consists primarily of five layers, starting from the top; *application layer*, *transport layer*, *network layer*, *link layer* and *physical layer* [7, 8].

A system (*e.g.* a desktop computer) typically has to implement a protocol for the four first layers. However in modern system support for multiple protocols may be implemented at each layer for the opportunity to choose given the type of service. The last layer, the *physical layer*, is usually handled by the hardware *e.g.* NICs.

The application layer sits at the top of the protocol stack, and is usually responsible for providing some service to the end user. Some protocols in this layer may also act as support protocols providing some common system function [7]. Two common protocols for this layer are File Transfer Protocol (FTP) and Secure Shell (SSH).

The transport layer is responsible for end-to-end communications. Transport layer protocols vary in complexity depending on the requirement of the end-to-end delivery. Some common protocols for this layer are User Datagram Protocol (UDP), and TCP.

The network layer is responsible for forwarding and routing data through the network, and is considered connection less. This layer applies addressing to the data being transmitted, but may also apply quality constraints and encryption to the data. Commonly there are only two protocols available for this layer, Internet Protocol (IP) version 4 and 6.

The link layer provides direct communication between two connected nodes. It is responsible for detecting (and possibly fixing) errors in the transmitted data, maintain a connection to the directly-connected nodes, and in some cases flow control. Most commonly used at the link layer is the range of IEEE802 protocols, such as WiFi and Ethernet.

The physical layer interfaces the physical medium directly. It is responsible for converting the data, which has traversed down the stack, from digital bits to a physical signals. Such signals may be radio waves with WiFi, electrical signals over coax, light using fiber, etc.

2.1.2 Internet Protocol

The Internet Protocol (IP) is designed for packet-switched communication between networks and within networks. It is used for transmitting data, as

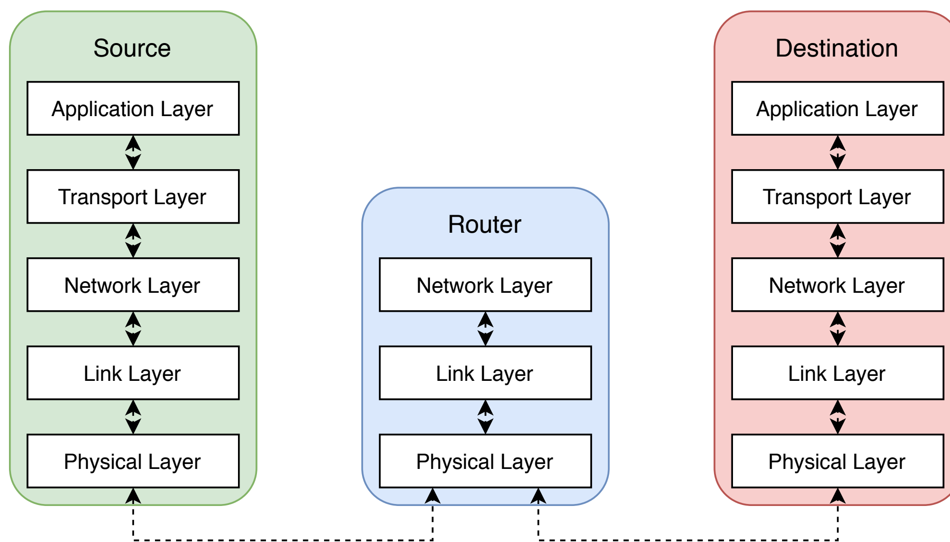


Figure 2.1: A diagram of the TCP/IP Stack and how data flows between the different layers.

datagrams, between a source and a destination. By adding the notion of an address to the datagrams, it allows for intermediate nodes throughout the network to forward it to the correct destination.

Routing

Routing in the Internet is performed by intermediate nodes along the network path, and are referred to as routers. Routers are responsible for forwarding the datagrams in the direction of the destination. Routers implement routing protocols which often rely on a form for Dijkstra's shortest path algorithm for determining the shortest routs to the destination. Example of such routing algorithms are Open Shortest Path First [56] and Routing Information Protocol [36].

To handle incoming bursts of traffic, routers are equipped with buffers which allow for intermediate storage of packets while routing table look-ups are performed. These buffers have to be sufficient in size with respect to the capacity of the network link which the router is attached to. If the buffers are too small the routers will drop to many packets, and if the buffers are too large this will lead to whats referred to as *bufferbloat*.

Active Queue Management

As increasing the queue length does not solve the problem of handling large burst of traffic by it self, other methods have to be taken into consideration. A mechanism introduced by Floyd and Jacobsen named Random Early

Detection (RED) [25] attempts to solve this problem by making a decision based on the average queue length to drop a randomly selected packet. What RED tries to achieve is to allow the queues to be relatively small while being able to compensate for bursts of traffic, and also providing the traffic effects that protocols *e.g.* TCP rely on to regulate their transmission rates. RED is one of many mechanisms within the field of Active Queue Management (AQM).

One of the major challenges for AQM is the choice of packet to drop [6].

2.1.3 Scalability

Usually, in the context of computer networks, scalability refers to the networks ability to scale with the amount of traffic flows in the network. I.e. if the network scales well, it is expected to work equally not matter the load or circumstances.

This is perhaps the most important aspect of the Internet as it is always expected to perform well independent of connected devices.

End-to-end Argument

The end-to-end argument, described by Saltzer *et al.* [63], is one of the key elements of the scalability of the Internet [12]. Simply interpreted the end-to-end argument states that the complexity should be moved "out of the network" at the end hosts, and that the network should be kept as simple as possible. This interpretation is a little more restrictive than it should be, as some complexity will in most cases have to be within the network, such as *e.g.* routing algorithms and AQM. It comes down to where or not complexity needs to be located within the network. If it does not, it should be moved out.

2.2 Transmission Control Protocol

TCP is designed for providing highly reliable host-to-host communication on a packet switched network [58]. The protocol is designed to fit in the transport layer (see fig) as little to no reliability are to be expected from the layers beneath. Thus, TCP only a assumes it will be able to obtain a simple, unreliable service from the layers beneath. In the other direction, TCP faces the application layer. It provides a simple interface which consists of a set of system calls which resemble those found in the operating system for manipulating files *e.g. read / write* [50, *read(2) and write(2)*], but with more suiting names such as *recv / send* [50, *recv(2) and send(2)*].

Reliability is achieved by implementing a set of functionality and operations such as *connections, sequence numbering, and checksums, flow control*. In addition to the address introduced by IP [57], TCP introduces a port

number which allows for identifying process and establishing a host-to-host communication between two processes. Before any transfer of data can be made, the connection must be established between the two hosts which ensures that both ends are ready for sending and receiving data.

To handle segments that are lost, duplicated or out-of-order TCP adds a sequence number to each segment. The receiving end can than use these numbers to keep track of what has been received, and inform the sender if anything went wrong. The receiver informs the sender by answering segments with and ACK which includes the segment number, which corresponds to the last byte received in-order. Informing the sender of what has been successfully received at the other end.

The one's compliment of the one's compliment sum of all 16 bit words in the TCP-header and data is used as a checksum to ensure that segments are delivered without any errors. In the case of the checksum being incorrect, the segment may have been corrupted and thus has to be corrected.

Flow control allows the receiver to govern the amount of data transmitted by the sender. The connected hosts make use of a window which define the maximum allowed range of sequence numbers that have to be ACKed before further transmission.

In addition to being highly reliable, TCP is also responsible for ensuring that the network path utilized between the two hosts is not congested. This is referred to as TCP Congestion Control.

2.3 Congestion Control

When the load on the network grows beyond what it can handle, the network gets congested. The queues on the routers in the network start to fill up faster than what they can forward the traffic. At some point the queues will reach their limit, and the routers have to start dropping packets based on their queue management scheme to avoid total congestion collapse. This also applies to the Internet. As we know from the *End-to-End argument* the complexity lays at the edge of the network [63]. It is up to the transport layer to adjust the amount data which are pushed out on the network at one time. In TCP this is done with a Congestion Window (CWND) which does just that; it defines the maximum amount of bytes (or packets) that a connection can have on the network at any time. This window is adjusted periodically by some Congestion Control (CC) algorithm which detects the amount of congestion on the network. In general these kinds of CCs detect congestion based on some kind of congestion event e.g. loss or delay.

2.3.1 Loss-Based Congestion Control

RFC5681 [4] defines four algorithms, first introduced by V. Jacobsen [43], for congestion control; *Slow Start*, *Congestion Avoidance*, *Fast Retransmit*

and *Fast Recovery*. These algorithms provide a set of rules for how the TCP sender should react in the presence of a congestion signal. Note that the congestion signal in which these algorithms react to is loss. However other signals *e.g.* Explicit Congestion Notification (ECN) could also be used as a trigger for these algorithms [4].

To start the *self clocking* mechanism of TCP, slow start was introduced. Slow start rapidly increases the flow of data in transit until it reaches somewhere near its fair share of the network capacity [43]. Once the equilibrium has been reached, the congestion avoidance takes over and attempts to maintain a steady flow of data on the network without causing congestion. Fast re-transmit and Fast recovery are algorithms are designed to increase the performance of TCP by elegantly handling a loss segment.

All these algorithms have in common that they are dependent on three important variables to work; (1) Congestion Window (CWND), (2) Receiver's Advertised Window (RWND), and (3) Slow Start Threshold (SSTHRESH).

Congestion Window is maintained by the sender to limit the amount of data that can be transmitted before an ACKed is received. Initially the CWND may be set to the size of one segment, but generally a larger size of four to ten segments is preferred to increase the performance of short lived connections or connections with large Round Trip Time (RTT)s [3, 13].

For each ACK received the CWND is traditionally increased by one Maximum Segment Size (MSS), however it is advised to increase by CWND += min(N, MSS), where N is the amount of unacknowleged bytes at the time of receiving the ACK. Which provides robustness against misbehaving receiver [2].

Receiver's Advertised Window is maintained by the receiver to limit the amount of outstanding data. As the name suggests, the receiver may advertise the maximum allowed CWND.

Slow Start Threshold is used to determine whether the slow start or congestion avoidance should be used to increase the CWND. In the case where CWND < SSTHRESH, slow start should be used, and congestion avoidance should be used when CWND > SSTHRESH. The initial value of SSTHRESH should be set arbitrary high or to the largest known RWND. However, by setting the SSTHRESH arbitrary high this allows the network conditions to dictate the sending rate [4].

Slow Start

is the mechanism used to reach the rate at which the ack-clock is *self-clocking*. The term *self-clocking* refers to the rate at which the ACKs are received at the

sender. This rate will be the maximum rate at which the sender can transmit over the network. For each received ACK the CWND is increased by one segment, or the amount of bytes ACKed. As long as no congestion event is experienced by the sender the CWND will double in size of each RTT, causing an exponential growth. However, at some point the limit of growth for the CWND will be reached. This can be due to a congestion event (*e.g.* loss or ECN), the RWND limit has been reached, or that the SSTHRESH has been reached. When this occurs the SSTHRESH is set to half the size of the CWND, the CWND is reset to some new initial value, and the congestion avoidance state is entered.

Congestion Avoidance

follows an Additive Increase Multiplicative Decrease (AIMD) growth model. Instead of increasing the CWND by one whole MSS for each received ACK, the CWND is commonly increased by $max(1, \frac{\text{MSS}\times\text{MSS}}{\text{cwnd}})$ for each received ACK. This means that the CWND increases by roughly one MSS for each RTT. Generally integer arithmetic is used in TCP implementations for calculating the window increment. Therefore the result may yield 0 and should be rounded up to 1. When the sender experiences a loss due to a time out the CWND is reduced by a multiplicative factor. If the new window size is less than the SSTHRESH the sender switches back to the *slow start* algorithm in the search for a new threshold as the network conditions may have changed. Else it continues with additive increase from the new window size.

The idea of the congestion avoidance is to keep the flow of data (throughput) near the capacity of the network for as long as possible without causing congestion.

Fast Retransmit

Whenever the sender receives a Duplicate ACK (DupACK) this can be caused by many many different reasons. It can be the result of a dropped segment, re-ordering in the network, or by replication of an ACK or data by the network. Anyhow this should be handled immediately by the sender. On the arrival of 3 consecutive DupACKs the fast retransmit algorithm interprets this as if the segment has been lost. It handles this by re-transmitting the segment(s) that seem to have been in hopes of fixing the issue before RTO expires.

Fast Recovery

The fast recovery algorithm handles the transmission of data after the fast re-transmit has sent what seemed to be missing. The reason for this is that the receiver can only produce DupACKs in the period when it is waiting

for the lost segment. However, the receiver will only send DupACKs if it receives a new segment, which most likely are the segments that were still in flight following the loss. Fast recovery uses this knowledge to its advantage by making artificial increments to CWND as "normal" until the lost segment has been received and the receiver stops sending DupACKs.

TCP Reno and TCP NewReno

The four algorithms presented in the previous sections are the basis for TCP-Reno (Reno). However, in later years some adjustments have been done to improve Reno which is referred to as NewReno. The latest being presented in RFC6582 [39]. NewReno specifies mostly changes to the Fast re-transmit and the Fast Recovery of Reno.

Though other algorithms for congestion avoidance are generally preferred over Reno and NewReno, many of these algorithms borrow some functionality for Reno/ NewReno. In this thesis we use the definition "Reno" as a common denotation for Reno and NewReno where it is worth mentioning.

Reno is not the default congestion controller in the current Linux kernel, however it is always available to fall back on as it s merged in the TCP implementation of Linux [73, *include/net/tcp.h*].

TCP Cubic

As Reno grows its window by roughly one segment each RTT, it may have problems utilizing the full capacity offered by the network. This is especially so if the network is categorized as a "long fat network", which are recognizable by having a high bandwidth and long RTT giving them a large Bandwidth Delay Product (BDP). In this case, if the length of the TCP flow is shorter than the time in which the window is able to grow to the full size of the network capacity, Reno will show quite bad performance.

TCP-Cubic (Cubic) is an improvement of TCP-BIC (BIC) which is designed to work reliably for both long- and short-RTT networks with a high bandwidth, achieving good [33, 60]. In BIC a binary search algorithm is used to grow the window to a midpoint between the last window size where the TCP witnessed a packet loss, and the last window size where TCP witnessed no loss for one whole RTT [75].

Cubic improves upon BIC by changing the binary search with a cubic growth function which greatly simplifies the algorithm. When a congestion event occurs, the CWND quickly grows to the previous known equilibrium point, following a concave growth curve. Around equilibrium point the growth function "flattens" out allowing it to stay around this point for some while before it starts probing for more capacity. When probing for capacity

if follows a convex curve, which allows it to quickly find the next max point. The key feature of Cubic is that the growth function is only dependent on the real time between two consecutive congestion events, thus making it independent of RTTs [33]. The authors also show that Cubic achieves good RTT-Fairness when multiple Cubic flows compete in the same bottleneck, as the flows have approximately the same window size independent of their RTTs [33].

Cubic became the default congestion controller of the Linux operating system in version 2.6.18 [33] and still is in the latest version.

2.3.2 Delay Based Congestion Control

Though loss-based congestion control it maybe more widely used [1], it has one drawback being that it has to increase its CWND until it generates a loss. This can be avoided by enabling ECN, which again only will notify the sender that it is right about to overflow the network, at which it may already be to late. A better solution would be to somehow monitor the congestion building up on the network, and back off before it gets fully congested.

This is exactly what delay-based congestion control attempts to do. Generally speaking, delay-based congestion controllers try to model the network congestion by using the *propagation delay* of data transmitted on the network as a congestion signal. As the queues along the network start to fill up, the *propagation delay* will start to increase. This can the be used as an indication that congestion is building up on the network, and that the sender should act less aggressively, possibly avoiding the loss of packets.

Generally speaking, a delay-based congestion signal may be one of the following; (1) One-way Delay (OWD), or RTT.

Using Delay as Congestion Signals

OWD is the time a segment uses from the sender to the receiver (or vice versa). It is up to the receiver to record the time at which the segment is received, and report back to the sender by attaching the information to *e.g.* an ACK. This excludes the re-transmission time from the calculation making OWD a more precise calculation, suitable as an indication for network issues and load for one direction. However OWD, comes with a price. For it to work it is imposed that both the sender and receiver have support for TCP timestamps [45]. In addition to this, the system clocks on both ends will never be completely synchronized, making OWD more of an estimation than an absolute value.

[1] In recent years Google has started to increase the used of BBR which is based RTT estimation. BBR is commonly used by the video sharing platform YouTube [44], however Cubic is still the most commonly used.

RTT is also an estimation of the time a segment is in flight. The estimation is calculated at the sender side by recording the time at which the segment enters the network, and recording the time at which the ACK for the corresponding segment is received. In practice this simplifies the calculation. It does not need the participation from both sides nor does it need any additional protocol support to work.

Using RTT as congestion signal, however, does come with some potential issues. One potential issue is the accuracy of an RTT estimation. As there is no guarantee that both directions are equally congested, the estimation may be skewed. In the worst case this could lead to the sender over estimating the amount of congestion in the network. This again may lead too the sender being to passive, and missing out on available bandwidth.

TCP Vegas

One of the earlier, delay-based, approaches is Vegas [9]. It tries to maximize the throughput while keeping the amount of loss as low as possible. To achieve this it uses an estimate of the amount of data that is queued in the network by regularly checking the difference between the current measured RTT and the BaseRTT, and tries to keep this estimate between two thresholds $\alpha < \beta$. The BaseRTT is the lowest perceived RTT since the connection started, while α and β are the boundaries for how much data the sender can have on the network at one time.

During the congestion avoidance algorithm of Vegas make use of Additive Increase Additive Decrease (AIAD) to regulate the congestion window on a per ACK basis. This is done by measuring the *expected rate*, derived from the BaseRTT and the current CWND. Followed by measuring the *actual rate*, which is derived from the current RTT and the current CWND. The queuing delay is computed as the *diff* between the *expected rate* and the *acutual rate* times the BaseRTT. When $diff < \alpha$, then CWND is increased by one. When $diff > \beta$, then CWND is decreased by one. Otherwise, when $\alpha < diff < \beta$, the CWND remains unchanged.

The use of RTT as a congestion signal allows Vegas to more timely react to network congestion, and the authors claim that Vegas is able to achieve between 40% and 70% more throughput than Reno [9]. However later studies have shown that Vegas performs much worse than that of loss-based congestion control when used in a heterogeneous environment with drop tail queues and large buffers [34].

Vegas is available as loadable kernel module in the latest version of Linux [73, *include/net/tcp_vegas.h*].

2.3.3 Explicit Congestion Notification

So far the method for detecting congestion in the network has been a result of implicit feedback along the network path, *e.g.* loss and queuing delay. Implicit because it is not a direct message from the intermediate nodes.

Relying only on implicit feedback assumes that the end nodes are able to make good, precise, assumptions about the network's performance. This however, is difficult as unpredictable behaviour may arise such as bufferbloat and "early comer advantage/disadvantage".

However, a solution for explicit notification from the network is available for IP, referred to as ECN [59]. ECN allows intermediate nodes along the network path, such as routers, to notify the end nodes that congestion is about to occur. This is achieved by marking incoming packets that exceeds some congestion threshold with a mark telling the destination node that congestion is about to occur. The destination node marks its ACK for the packet marked with the congestion mark, which tells the sender to back off before it causes the router to fully congest.

2.3.4 Fairness

A question related to the performance of a congestion controller is it fairness. Generally interpreted as how much of the networks available resources should be delegate to each traffic flow. The difficult of defining fairness increases when the users of the network do not share the same amount of resources. This is generally the main concern when it comes to the filed of Qos, however, in terms of TCP congestion control it generally comes down to how a common bottleneck can be equally shared between the competing traffic flows.

Jain's Fairness Index

One way to ensure that a bottleneck is being shared equally is by applying Jain's Fairness Index [46]:

"If a system allocates resources to n contending users, such that the ith user receives an allocation x_i, then the fairness index $f(x)$ is defined as:"

$$f(x) = \frac{(\sum_{i=1}^{n} x_i)^2}{\sum_{i=1}^{n} x_i^2}$$

This results in a good measure for faines as $f(x) = 1$ if all allocations x_i are perfectly equal, but quickly becomes less than one if they are not. And thus making it suitable for determining the fairness between competing congestion controllers on a shared bottleneck.

TCP Friendliness

A common definition of fairness with respect to TCP is TCP friendliness. It comes from the assumption that most of the traffic in the Internet comes from TCP. Braden *et al.* use the term TCP-compatible instead of TCP-friendly, and define TCP friendliness as follows [6]:

"A TCP-compatible flow is responsive to congestion notification, and in steady-state it uses no more bandwidth than a conformant TCP running under comparable conditions (drop rate, RTT, MTU, etc.)"

2.4 Less-than Best Effort Delivery

Less-than Best Effort (LBE) transport services can be loosely defined as transport services that result in a smaller impact on bandwidth and delay when sharing a bottleneck with BE transport services. While standard TCP traffic sources make an attempt to share the bottleneck evenly amongst each other, a LBE traffic source tries to avoid any disruption of competing traffic. LBE tries to stay in the background only to utilize available network capacity when there are no (or little) competing traffic, leading to it being referred to as a "scavenger protocol". Scavenger protocols can be seen in use by companies such as BitTorrent LEDBAT [66], Apple [42], and in recent years by Microsoft [21].

2.5 Deadline Aware Less-than Best Effort Delivery

Deadline-aware less-than best effort (Deadline-Aware Less-than Best Effort (DA-LBE)) is a Less-than Best Effort (LBE) service with a notion of timeliness [35]. It strives to keep the disruption of other Best Effort (BE) traffic to a minimum as well as delivering it's data within a *soft deadline*. Hence the term *deadline aware*. This makes it possible for a DA-LBE traffic source to linger in the background, letting other traffic sources utilize most of the available bandwidth. As the deadline closes in, the DA-LBE traffic source adjusts its aggressiveness to gradually compete more and more for the available bandwidth. If the deadline has been reached and the DA-LBE traffic source is not yet finished it will compete with the other traffic sources as if it was a BE traffic source.

2.5.1 Measuring the Price of Congestion

In Network Utility Maximization (NUM) the network congestion control problem is framed as an optimization problem. The objective is determine the appropriate send rate x_s for the traffic source s to maximize the utilization of bandwidth subject to the link capacity constraints. This problem can be solved using the Lagrangian dual, where the Lagrange multiplier q_s is

considered to be the congestion signal or price of each link l in the network path.

Hayes *et al.* used NUM to show that an Less-than Best Effort (LBE) services is modeled as traffic source s that inflates its measured network price $q^{(s)}$, by some weight $w^{(s)} \in [w_{min}, w_{max}]$. These limits determine the degree of Less-than Best Effort (LBE), or the LBEness of the traffic source. By adjusting the weight $w^{(s)}$, the aggressiveness of the send rate of the traffic source is also adjusted. The closer $w^{(s)}$ is to w_{min}, the closer it is to its lowest Less-than Best Effort (LBE) rate. Similarly, the other way, the closer $w^{(s)}$ is to w_{max}, the closer it is to a Best Effort (BE) rate. $w_{max} = 1$, so that the Less-than Best Effort (LBE) traffic source will be no more aggressive than a normal Best Effort (BE) traffic source [35].

2.5.2 Adapting The Weight

The weight $w^{(s)}$ of the traffic source is periodically changed throughout the lifetime of the connection with respect to the congestion price $q^{(s)}$ to reach $\hat{q}$ which is the congestion price for the sending rate required to deliver the remaining data within the *soft deadline*. The weight is determined by the target rate ζ, which defines the lowest send rate which the source needs to reach the *soft deadline* t_D, after the nth interval of duration T_w.

$$\zeta(t_n, t_D) = \frac{\text{data remaining}}{t_D - t_n} \tag{2.1}$$

Two methods of adapting w have been proposed:
a Proportional-Integral-Differential (PID) controller which base their control of the error ϵ between the current state and the target state, and a Model-Based-Control (MBC) which relies on having a good model for the protocol send rate with respect to the network price that would achieve the desired Less-than Best Effort (LBE) bit rate.

Model Based Control

In Model-Based-Control (MBC) the goal is to have a good model of what congestion price $\hat{q}_n$ is desired to achieve the desired Less-than Best Effort (LBE) send rate. An error ϵ_{n-1} is used to determine the relative difference between the target send rate and the actual send rate $\bar{x}$ of the preceding interval $(t_{n-1}, t_n]$.

$$\epsilon_{n-1} = \frac{\zeta(t_{n-1}, t_D) - \bar{x}(t_{n-1}, t_n)}{\bar{x}(t_{n-1}, t_n)} \tag{2.2}$$

The new weight is based on the price that will achive the expected send rate $\hat{q}$ with respect to the actual send rate of the previous interval q_{n-1}. This

is corrected by the error ϵ_{n-1} and results in a new weight w_n between w_{min} and w_{max}.

$$w_n = \left[\frac{q_{n-1}}{\hat{q}_n} (1 + \epsilon_{n-1}) \right]_{w_{min}}^{w_{max}} \tag{2.3}$$

To assure that the weight does not grow too much at each interval, an increase limit l_w set for the weight. There is no limit on the decrease, as sudden drops in aggressiveness are crucial for the controller to back off and minimize disruption of competing traffic.

$$\hat{w}_n = \begin{cases} w_{n-1} + l_w w_n & \text{if } (w_n - w_{n-1}) > l_w w_n \\ w_n & \text{otherwise.} \end{cases} \tag{2.4}$$

Note that the key to a well performing Model-Based-Control (MBC) requires a good model of the congestion controller.

Model-Based Control for TCP Vegas

In practice Vegas not only reacts to delay, but also loss. In fact the implementation of Vegas in the Linux kernel inherits the functions used by Reno when a loss event occurs to reduce the congestion window. This has to be taken into account when adjusting the weight. However, mixing CCs that react to different congestion prices makes it more difficult for Dalbe to ensure its LBE-ness. To solve this Hayes *et al.* suggest that the price of congestion has to be mapped to probability of congestion indication, $\mathbb{P}[\text{cong_ind}]$. This is then weighted by the relative effect the receipt of each congestion indication, W [35]. For Vegas we get the following:

$$\phi = \frac{W^{(\text{loss_reno})}\mathbb{P}^{(\text{loss_reno})}[\text{cong_ind}] + W^{(\text{delay})}\mathbb{P}^{(\text{delay})}[\text{cong_ind}]}{W^{(\text{loss_cubic})}\mathbb{P}^{(\text{loss_cubic})}[\text{cong_ind}]} \tag{2.5}$$

$\mathbb{P}^{(\text{delay})}$ is defined by $I^{(\text{delay})}$, which is the number of delay-based congestion indications for the interval. This is then divided by N which is the number of acknowledged packets for the interval. $\mathbb{P}^{(\text{delay})}$ is then weighted by W, which is defined by the corresponding proportion of which CWND has been reduced for the interval, divided by $I^{(\text{delay})}$.

For the loss-based proportions we set $\mathbb{P}^{(\text{loss_reno})}$ equal to 0.5, as Reno performs a multiplicative decrease of the CWND when a loss event is encountered resulting in the CWND being halved [4]. Cubic reduces its CWND by a decrease factor β, which in general is set to $\beta \approx 0.7$ [60].

From this we get ϕ which is the chance for Vegas to witness a congestion event of any kind, normalized by the chance to experience a congestion event used in the network [74]. In our case this is Cubic.

The algorithm for a model based DA-LBE adaption of Vegas is defined as follows:

Algorithm 1 Update weight according to model for Vegas.

Every T_ϕ**:**
    if *loss events* > *CALC_THRESH* **then**
        $\phi \leftarrow$ equation (2.5)

Every T_w**:**
    $q_n \leftarrow \overline{RTT} - RTT_{base}$
    $\hat{q}_n \leftarrow \frac{\alpha}{\zeta(t_n, t_d)}$
    $\epsilon_{n-1} \leftarrow$ equation (2.2)
    $w_{base} \leftarrow \frac{q_n}{\hat{q}_n} \times \frac{1}{\phi}$
    $\bar{w}_n \leftarrow w_{base} \times (1 + \epsilon_{n-1})$
    $\hat{w}_n \leftarrow$ with $\bar{w}_n$ limit increase using equation (2.3)
    $w_n \leftarrow$ with $\hat{w}_n$ clamp using equation (2.4)
    $\mu_n \leftarrow w_n \phi$

    if $w_n == 1.0$ **then**
        $B \leftarrow 1 - \frac{1}{\mu_n}$

We only perform the calculation of ϕ is there are enough loss-event to make a decent calculation. q_n is the measured queuing delay for that interval, while $\hat{q}_n$ is the modeled queuing delay. α is the lower threshold of Vegas.B is the back-off parameter which defines the chance to ignore a loss-event. This is only used when $w_n == 1$, or in other words; when Dalbe is at its most aggressive.

Adjusting the Aggressiveness of Vegas Similarly as stated by Wallenburg *et al.* the model presented by Hayes *et al.* [35] for Vegas did not map directly to the kernel module in this work. In practice adjusting the kernel parameter α is not feasible to as this value is globally set and thus changing it would affect all other connections using Vegas. In adition to this, the α parameter is set statically as a global variable in the Vegas in the Linux kerenl. This makes it almost impossible to adjust from a concurrent module.

Instead of adjusting the perceived load by modifying α directly, we can alter the parameters which Vegas uses to determine the price of congestion. For Vegas this is referred to as the queuing delay and measured by calculating the difference of the based RTT and the RTT of the last acked packet [9]. By inflating the RTT Vegas uses to calculate this queuing delay we can make it perform more or less aggressively.

Algorithm 2 Algorithm that alters the RTT passed on to Vegas.

Every *On every reveived ACK*:
$q_{ack} \leftarrow RTT - RTT_{base}$
$RTT \leftarrow RTT_{base} + \frac{q_{ack}}{\mu_n}$

Proportional Integral Differential Based Control

For PID the signal is based on a combinations of current error, past history and the project error. A normalized error signal ϵ_n is mapped to the weight w_{n-1} of the previous interval to enable easy scaling.

$$\epsilon_n = \frac{\zeta(t_n, t_D) - \bar{x}(t_{n-1}, t_n)}{\bar{x}(t_{n-1}, t_n)} w_{n-1} \tag{2.6}$$

ϵ_n is then used together with the gains for the proportional (K_p), integral (K_i) and differential (K_d) parts to create the PID signal u_n. u_n is used to determine the weight w_n for the next interval.

$$I_n = I_{n-1} + T_w \epsilon_n \tag{2.7}$$

$$u_n = K_p \epsilon_n + K_i I_n + K_d \frac{\epsilon_n - \epsilon_{n-1}}{T_w} \tag{2.8}$$

Note that the key to a well performing PID controller is tuning its gains K_p, K_i and K_p.

PID-Based Control in Practice

When updating ϕ while using PID based control we have to take into account that the underlaying CC of Dalbe may be loss- or delay-based. A part from calculating w differently, the only difference from the model for Vegas.

2.5.3 Meta Congestion Control

As the DA-LBE framework collects various statistics from the network, which are then used to adapt an underlying congestion controller, it is often referred to as a *meta* congestion controller.

Algorithm 3 Update weight according to PID-Based control.

Every T_ϕ**:**
  if *loss events* > *CALC_THRESH* **then**
    $\phi \leftarrow$ equation (2.5)

Every T_w**:**
  $\epsilon_n \leftarrow$ equation (2.6)
  $u_n \leftarrow$ with ϵ_n equation (2.7 and 2.8)
  $\hat{w}_n \leftarrow$ with u_n equation (2.3)
  $w_n \leftarrow$ with $\hat{w}_n$ limit increase using equation (2.4)
  $\mu_n \leftarrow w_n\phi$

  if $w_n == 1.0$ **then**
    $B \leftarrow 1 - \frac{1}{\mu_n}$

2.6 The Linux Operating System

The Linux Operating System (OS), developed Linus Torvalds, is part of a large family of UNIX-based OS'. Linux has become very popular over the years, which mainly is the result of it being an open source project under the General Purpose Licence (GPL). Which allows anyone to dive into the enormous code base [2], and contribute to the ever growing OS.

The operating system is widely used on the server side, but over the years many distibutions of Linux have been made to target the desktop users *e.g.* Debian, Fedora and RedHat. Linux has also achieved great success in the world of mobile computing by being the backbone for the very popular Android OS. Being a UNIX-based system, Linux also shares many of the fundamental design ideas as other UNIX-based systems such as FreeBSD, Solaris, and even Mac OS X.

One of the major advantage of Linux is that it does not strictly stick to any type of kernel architectures, but rather it makes an attempt at adopting the best choices from well known kernel architectures. Other advantages of Linux is that;

- it is *cost free*;
- it is *fully customizable*;
- it can run on *inexpensive, low-cost hardware*;
- it is *designed for efficiency*;
- it is *maintained by professionals*;

[2] Linux entered 2020 with about 27.8 million lines of code [51]

- it can be made *small and compact*;
- and it is *compatible with most hardware*.

This makes Linux a strong competitor to its commercial counterparts such as Windows and Mac OS X.

2.6.1 Versioning

Up until kernel version 2.5, the version system had a different meaning to what is used in present time. It used to follow a *x.y.z* numbering system where the first number, *x*, referred to the major version. The second number, *y*, was the minor version, and the third *z* referred to the current release. If the minor version number was odd this would mean that the release was in a unstable state, while if it was even the release was considered stable.

In later years the number have a different meaning. The first two, *x.y*, refer to the major version, containing about 13000 change sets with changes to several hundred thousands lines of code [49, *process/2.Process.html*]. At the beginning of each so called development cycle, the merge window is said to be "open". This allows for patches with new code that is considered stable to submitted and merged into the mainline kernel. After approximately two weeks Torvalds "closes" the merge window and the current mainline kernel becomes the first of the "rc" kernels (*e.g.* 5.6-rcN is destined to be 5.6). During the weeks following the close of the merge window, only fixes to the "rc" kernels are allowed. As fixes make their way into the kernel multiple versions of the "rc" kernel will be released by Torvalds until the kernel is considered sufficiently stable for an official release.

When a new official kernel has been released, ongoing maintenance is made by the "stable team" which may lead to additional versions of the official kernel. In these cases a third number, *y*, is used to determine the version (*e.g.* 5.6.4).

2.6.2 Contributing

The Linux source code consists of tens of million lines of code. Way beyond what one person is capable of maintaining by them self. For this reason the code base has been broken into a set of subsystems which is maintained by a designated developer. Each maintainer has the responsibility for the code in that subsystem, and all patches that involve the subsystem should be submitted to this person. Subsystem maintainers each maintain their own version of the source three which they request to be merged into the mainline kernel when the merge window is open and subsystem is stable enough.

The Netdev Group

The *netdev* group is responsible for the networking subsytem in the kernel source tree [49, *networking/netdev-FAQ.html*] and all network related questions should be directed to the *netdev* mailing list [3].

Two trees are always in play, *net* and *net-dev*, which are maintained by David Miller, who is the main networking maintainer. The *net* tree is used for networking related fixes to the mainline tree maintained by Torvalds. The *net-next* tree is used for new network related features and drivers which are targeted for the next mainline release.

2.6.3 Loadable Kernel Modules

In Linux 1.2 Loadable Kernel Module (LKM)s have been a method for adding functionality to the kernel source code [38, *x73.html*]. LKMs are usually used for adding device drivers, file system drivers and system calls. This isolates functionality that does not strictly need to be wired into the core of the kernel, making the kernel more modular.

The fact that the LKMs do not have to be wired to the kernel, makes it possible to build and deploy them at run-time, and is a major selling factor for LKMs. This decreases the amount of times the kernel needs to be compiled, helps isolate bugs, and not to mention providing a modular system.

User Space and Kernel Space

The OS is responsible for maintaining the integrity of the system [67]. Through techniques such as *virtual memory* and *file systems*, it manages the shared resources of the system among the running threads and processes (tasks). In addition to this, it *schedules* these running tasks by deciding which should be allowed to run at each time and for how long they should be allowed to run.

Another important factor for maintaining the integrity of the system, and not to mention the safety of the system, is by restricting certain access to crucial system resources from some of these tasks. The OS may achieve this in two ways; (1) by assigning certain *privilege levels* to the tasks, and (2) by assigning memory private memory regions for the running tasks. Usually both are applied, and depending on the architecture in which the OS running on, there may be several privilege levels. Commonly for these levels are that applications run at the lowest privilege level, often referred to as *user space*, to protect the integrity of the system from unpredictable, malicious code. While the OS runs at the highest privilege level(s), referred

[3]Mailing lists are frequently used for development on the Linux kernel. Each subsystem has their own mailing list.

to as *kernel space*, which allows it to manage the system, and thus maintain the integrity of it.

Kernel Modules Versus Applications

Other than that kernel modules run in *kernel space* and Applications run in *user space*, there are some other, important, distinguishing factors between them. The first distinguishing factor is that an most applications usually perform a single task which runs from start to finish, while a kernel module only registers it self to the kernel, with the intent of serving a purpose in the future [14]. This in comparison to each other the flow of the kernel module can be compared to *event driven* programming, while the flow of an application is more *sequential*.

Another distinguishing factor is that an application may be "lazy" in terms of freeing up allocated resources, and instead relying on garbage collecting mechanisms provided by the OS. A kernel module, on the other hand, must make sure that all resources, allocated at initiation, are freed at the point of release. This is very important, as this will otherwise, unnecessarily, consume the precious resources in the system.

Lastly, an most importantly, is how faults are handled. For an application, a segmentation fault[4] is considered quite harmless during the development process, as it at most crashes the process, which can in most cases be easily debugged and fixed. For a kernel module, however, a segmentation fault is much more crucial. If not handled, kills the current process at, if not the entire system, and the cause may be very hard to trace back.

2.6.4 TCP/IP Stack in Linux

The TCP/IP stack in the Linux is implemented as set of function pointers, referred to as *hooks*, which point to the entry and exit of each layer. Within each layer there is usually a chain of function calls which manipulate a network buffer, before passing it on to the layer beneath or above, depending on which direction the data is headed. This network buffer, *sk_buff*, represents the packet structure in Linux, and carries information about the header, the data, and additional information about paged data [61].

Moving Down the Stack using TCP

Figure 2.2a illustrates a simplified example of the major functions that data passes through when sending data. Using a socket configured for IPv4 and TCP. When the application calls the library function *send*, from the

[4]A segmentation fault occurs when the process attempts to access a memory location which it does not have access to. This mechanism protects the memory outside boundary of the process, but usually causes the process to crash in doing so.

application layer, this is actually hooked to the sockets *inet_sendmsg*, which is responsible for calling the correct protocol function based on the setup. In this case it calls the *tcp_sendmsg* function, as the socket is configured for TCP. Now in the *transport layer*, the data from user space is broken into smaller segments, and attached to the *sk_buff* with protocol specific information, before its placed in the socket send buffer. If the TCP state machine is able to transmit the buffer, it invokes the *ip_queue_xmit*. In this function, which is considered as the *network layer*, IP specific information is attached to the *sk_buff* before it is passed on to the network device by invoking *dev_queue_xmit*. At the final layer before entering the network, *link layer*, Quality of Services may be implemented, as well as Queuing Disciplines, before the *sk_buff* is added to the device driver's transmit queue, and finally transmitted over the network.

Moving Up the Stack using TCP

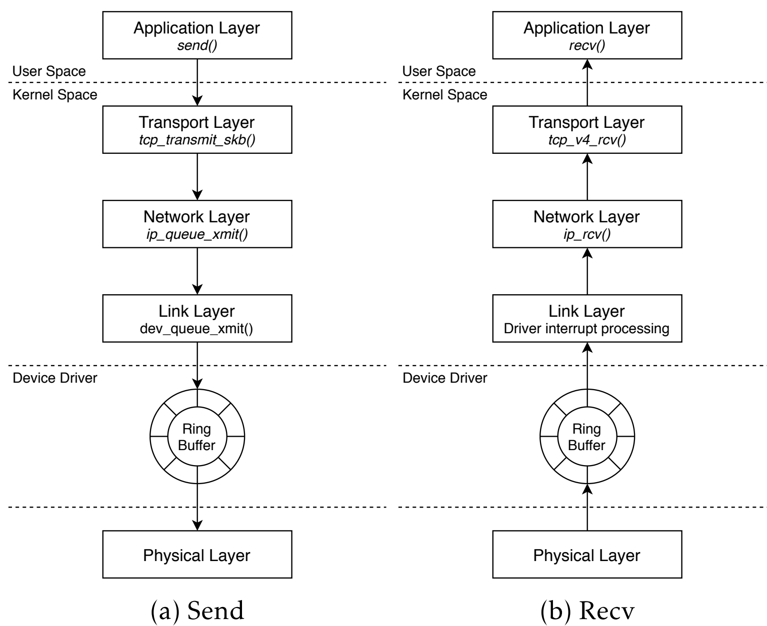
Figure 2.2b illustrates a simplified example of the major functions data that data passes through when receiving. Using a socket configured for IPv4 and TCP. At the reception of a packet by the device driver, an interrupt is generated, notifying that the data has been placed in the receive queue. The packet, now represented as a *sk_buff* is processed in the *link layer*, were the IP family is determined. In this case this is IPv4, and the *sk_buff* is passed to the *network layer* through the *ip_recv* function. At this point, IP specific decisions are made for the packet, such as routing and firewall policies, before is its passed on to the protocol handler. In this case the protocol used is TCP, and the handler used is the tcp_v4_recv function. Now in the *transport layer* the application socket is determined, followed by processing the TCP segment. The processing of the segment determines if it is and ACK, contains data, and other TCP specific fields. Finally, if the segment in fact contains data, this is placed in a read queue for the socket, which the application can read from using, in this case, *recv*.

2.6.5 TCP Congestion Control in Linux

In version 2.6 Stephen Hemminger released a patch [37] which cleaned up networking code involving the congestion control algorithms of TCP [15]. This was the result of the increasing interest for experimentation with congestion controllers at the time.

The rework included an interface [73, *include/net/tcp.h*] which made it possible for congestion controllers to be implemented as kernel modules. Specifically, the interface made it possible for kernel modules to hook[5] on to the congestion control functions which where called by the internals of the TCP implementation in Linux.

[5]The term *hook* is often used to referrer to function pointers.



(a) Send (b) Recv

Figure 2.2: A simplified model showing two examples of how packets traverse the networking stack in Linux when TCP and IPv4 are used as the transport- and network layer protocols. Sub-figure 2.2a shows the major functions which the data is passed through from leaving the Application layer, before entering the network. Similarly, sub-figure 2.2b shows the major functions which the data pass through before arriving at the Application layer.

The four required functions, which is implemented as a structure named *tcp_congestion_ops* [73, *include/net/tcp.h*], are the following:

- **ssthresh**: Calculates a new slow start threshold and is called at the time at when the congestion window is reduced.
- **cong_avoid**: Contains the congestion avoidance algorithm for calculating a new CWND for each congestion controller. It is called towards the end of processing an ACK.
- **undo_cwnd**: Is used for calculating a new congestion window size after a loss has occurred.

The TCP implementation in the kernel also depends on a state machine which governs the senders actions when ACKs are received. These states are as follows:

- **Open**: When the sender is in this state the connection is *open* and it acts as normal. This means that there has been no unexpected events, and all processing of packets go through what is referred to as the *fast path*. The *fast path* is the name used for packets that go through the minimum required processing before entering the network. Meaning that the sender increases the CWND according to the slow start or congestion avoidance algorithms.
- **Disorder**: If any dupACKs or SACKs are received, the *disorder* state is entered. This means that each packet requires more attention, and some are moved from the *fast path* to the *slow path*. The *slow path* is the name used for packets that go through additional processing before entering the network. This could be due to the sender being in the fast re-transmission or fast recovery state.
- **Congestion Window Reduced (CWR)**: In some cases the sender may receive an explicit congestion event such as an ECN. If this is the case the sender enters the *CWR* state, and reduces the congestion window accordingly. This state can be interrupted by both *Recovery* and *Loss*.
- **Recovery**: This is essentially the fast recovery algorithm of TCP. When the sender receives three dupACKs in a row, this state is entered. During this state the sender reduces the CWND incrementally until it reaches the SSTHRESH. The state is changed to *open* when the outstanding packet that triggered this state has successfully been delivered, or changed to the *Loss* if an RTO occurs.
- **Loss**: If an RTO occurs, the sender enters this state. All outstanding packets are in this case considered lost, and the sender enters the slow start algorithm.

2.6.6 Floating Point Operations in the Kernel

From *user space's* point of view, floating point instructions appear to be as easy as integer instructions, as the OS handles the transition between the integer and floating point [52]. This transition often includes a trap*The term trap refers to the an exception which is caught by the kernel, and which usually results in a context-switch between user space and kernel space to perform a specific task.* which is caught by the kernel, and results in the operation being transferred to the Floating Point Unit (FPU).

Though floating point operations are not impossible to perform from within the kernel, it is highly discouraged use them. The reason for this is that the kernel is not able to trap it self, resulting in it having to manually save and restore the floating point registers, while also turning on and off preemption. This process is quite heavy on the performance of the kernel, and should not be performed unless it is a very special case with no other viable solutions.

Fixed Point Operations

One possible solution for overcoming the absence of floating point operations is to used scaled integers for representing fractional numbers. This is often referred to as *fixed point* numbers as the integer is fixed into two parts, an integer-part and a fraction-part. The fractions are most commonly represented in base 2, as re-scaling can naturally be done using *shift* operations.

The representation used for fixed point numbers is often referred to as *Q-notation*, *e.g.* $Q16.16$ represents a 32-bit integer divided into two equally large parts. The most significant 16 bits represent integer part, while the least significant 16 bits represent the fraction part.

2.7 Summary

This chapter gave an introduction to the relevant background material for this thesis. Building up from basic concepts within the field of computer networking such as the TCP/IP stack, how packets are routed through the network, and how the Internet scales in the presence of network traffic. Following this introduction was a more in-depth description of the TCP protocol, how it handles congestion in the network, and how its friendlyness to competing traffic can be measured. Building upon the concept of congestion control, LBE transport services was briefly introduced, followed by a thorough description of how the introduction of *soft deadlines* an LBE service evolved into the framework referred to as DA-LBE. Finally, an introduction to the Linux OS was presented, together with the concept of Loadable kernel

modules and how they differ from applications, how the networking stack is implemented in the kernel, and how floating operations are discouraged in the kernel.

Chapter 3

Methodology

In this chapter I present the methodology, methods and tools which I applied during this project, and how it played an important role for the outcome of this thesis.

3.1 Approach

The approach I chose to follow for solving the problem was inspired by *agile development*, were the focus was to work in an incremental, iterative process, with high focus on testing [65]. Figure 3.1 illustrates the main idea of the working process, were I made an attempt to divide my work into four phases; (1) a planning phase, (2) a design phase, (3) an implementation phase, and (4) a testing/experimentation phase, which could resemble a sprint cycle in SCRUM.

In this way I could challenge my self to get a good overview of the tasks that needed to be done, portion the amount of work in to smaller tasks, find out in which order these tasks should be solved, and to have a good idea of the progress I had made.

The main idea of the four phases are described in grater detail in the following sections, together with the main methods and techniques that were used in each phase.

3.1.1 Planning Phase

At the beginning of every main task (*e.g.* building the test environment), I would create a *backlog* of smaller tasks that were a subset of the main task. To keep track of these tasks I used a SCRUM-table [65]. In addition to this I made weekly plans, as a means of setting a goal of how much I anticipated to be able to finish for each week.

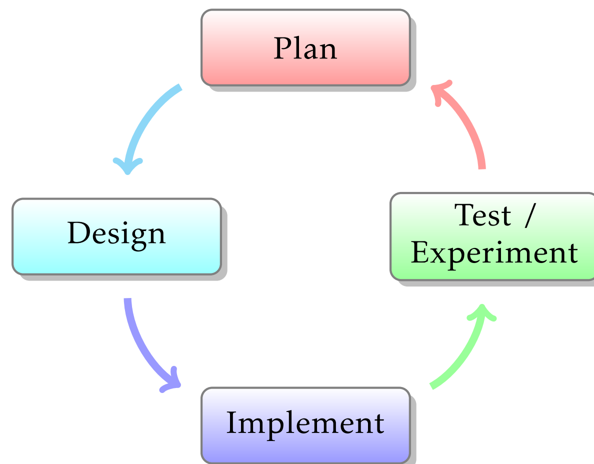


Figure 3.1: A simple model of the work flow used while working on this project. This was a simple adaptation of a typical agile / test driven development work flow, and even though the arrows are pointing in one direction it did not strictly mean that this was the order of that always followed.

3.1.2 Design Phase

After planning I would start working the tasks which I had created for my self. The first thing I would do was to design a solution for the task. At times this would not be very challenging as the task would be very simple, but in other cases the designing phase would be the major part of solving the task.

During this process I would find literature relevant to the task, which often revealed several ways to solve the task. This was very important for identifying the correct approach for solving the task, as well as avoiding "re-inventing the wheel" if a solution already existed.

With the relevant knowledge and literature it became easier to identify the requirements for the system, both functional and non-functional [65], as well as identifying the feasibility of these requirements. As the requirements specification, which in my case was quite informal, would expand, so would the amount of sub-tasks in my SCRUM-like table.

3.1.3 Implementation Phase

While working with the implementation(s) I had a great focus on code conventions and code cleanliness, as what I produced aimed to, eventually, be part of the large, Open Source project, Linux. Putting in a little extra effort in this phase, meant that the next developer should have little to no problem reading and contributing to the code later on.

Also during the implementation phase, a decent amount of time was spent on writing documentation. The documentation varies in depth, as

it mainly aims to provide additional support for building, setting up and using the provided software, without repeating the thesis. I chose to add it as *README* files in the different repositories, as well as part of the appendix.

3.1.4 Testing and Experimentation Phase

The last phase of my agile inspired development process consisted of testing and evaluating my software. In this project the testing and experimentation were closely related, as I would often perform tests based on an experimental setup which would lead to verifying the correctness of the software as well as produce results which also allowed me to reason about the performance of the software. I chose a subset of methods which in my opinion were adequate for testing the software produced in this project.

Unit Testing

Unit testing, or in some cases referred to as Component Unit Testing [5], is a means of testing program components, such as objects, functions [65]. These tests often aim to verify a specific behaviour of a system and is components. In my work, I used unit tests as means of testing where I could isolate certain functionality, *e.g.* mathematical operations and system calls, to verify their correctness and performance in an sandbox-like environment. In this environment I would try to create tests which provided enough coverage to test all operations related to these components, check that the results were correct for all cases. This would often involve creating mock-up servers and clients which would allow for testing certain functions that relied on a connection between two hosts to work.

Micro-Benchmarking

Another form of testing that I performed was micro-benchmarking. Similar to the Unit tests, certain operations are isolated and tested on artificial workloads, with the intention of giving an idea of the operations performance [30]. In this thesis i did some experimentation with benchmarking some arithmetic operations. However, benchmarking is quite difficult to get right, as having complete control of what compilers and the OS does is very difficult. For this reason it did not end up being part of the final thesis, however, I consider it a valuable experience, and thus I feel it should be mentioned.

Event Tracing

Systems operate through discrete events, such as CPU instructions, disk I/O, network packets, system calls *etc.* [30]. Through performance analysis, these events may be summarized and studied. However, in some cases these

summaries may miss crucial events. Event tracing, when done correctly, can allow for inspecting these events individually. This, in turn, may lead to a better understanding of the system, as well as reveal certain performance issues which are not visible through performance summaries.

I used event tracing on many occasions during this work. Some examples where tracing was used was for tracing network packets using tools such as *Wireshark* [27] and *tcpdump* [50, *tcpdump(1)*], for inspecting function overhead within the kernel using *ftrace* [50, *ftrace(1)*], or for revealing memory leaks using *kmemleak* [49, *dev-tools/kmemleak.html*].

3.2 Collaboration

At a point during this project I joined forces with a fellow master student on building a suitable testing environment, which could be utilized by us both. During this collaborative work the agile inspired development process became very useful. For a period of approximately one month, give or take, we ran weekly sprints were we created a plan of which tasks we assumed we would be able to finish, within that week. This was very helpful as it was became very easy to keep track of the tasks we were working on and the progress we had made. In addition to this, we could perform some informal code reviews [5] for the software we implemented, which in my opinion was very valuable for the quality of the software we produced.

3.3 Tools

Some of the methods and tools used during this project have already been mentioned in previous sections. In this section I want to give an extra introduction to what i consider some of the most important tools mentioned, and that have played a major role in the outcome of this thesis.

3.3.1 Common Open Research Emulator

Common Open Research Emulator (CORE) is an open source project by the Boeing Company [16] designed for building virtual networks. It takes advantage of virtualization provided in the Linux operating system, like virtual interfaces and Ethernet bridges. The key features of interest that CORE provides is that it is efficient and scalable, and has a drag-and-drop General User Interface (GUI) [1]. This made it quite easy to emulate different virtual network topologies and add custom scripts to each virtual node for specific network configuration, and was therefore a good option for our test bed.

Architecture

The main components of the CORE architecture is the *core-daemon* and the *core-gui*. The daemon runs as a service on the Linux machine, and has to be started for the other components to work properly. Through the *core-gui* or custom scripts which use the *pycore* modules directly the user can create virtual network topologies. The virtual networks are constructed by CORE node and tied together using Linux Ethernet Bridging. The CORE nodes are small, lightweight, virtual machines which can be accessed through a shell and/or which can be given custom scripts to run at start-up that manipulate links, routes, etc. [1, *architecture.html*]

Performance

The main factors of concern when it comes to the performance of CORE, stated by the developers, are the following; hardware in use, OS version, amount of active processes, amount of network traffic being produced, and the GUI usage [1, *performance.html*]. According tho the developers they found it reasonable to run 35 to 70 nodes on a typical single-CPU Xeon 3.0GHz server machine with 2GB RAM running Linux. In their opinion, the main concern for the user should be *how much traffic* the emulator would be able to handle at once. On the same setup they were able to handle approximately 300000 Packets Per Second (PPS), which represented the amount of times the system would have to deal with a packet. More network hops and paths would effectively increase the amount of context switches and decrease the throughput in a path
[1, *performance.html*].

3.3.2 Linux Performance Events

Linux Performance Events (perf_events), often just referred to as *perf* [50, *perf(1)*], is a set of tools supporting a wide range of performance observability activities for the Linux kernel [30]. It provides static and dynamic tracing of the kernel, based on functionality such as *tracepoints* and *kprobes*. In addition to this it provides support for profiling, an other analytical tools directed at inspecting the internals of the kernel. The fact that this tool set is part of the mainline kernel makes it very trivial to set up and use, and the support it provides for observing the system is in many cases adequate for solving most performance related tasks.

3.3.3 Function Tracer

The Function Tracer framework, refered to as *ftrace* [49, *trace/ftrace.html*], is an internal tracer for the Linux kernel. This tools is designed specifically to give kernel developers an insight of what is happening within the kernel.

Ftrace is most commonly known for being a function tracer, however, it should be considered more a tool set, as it provides much more than just function tracing. Its functionality can be very useful for debugging and analysing events that occur outside the realm of user-space.

The process of setting up *ftrace* is more complicated than for *perf*. Though these exist front-ends for controlling *ftrace* [50, *trace-cmd(1)*], which resembles the front-end of *perf*, *ftrace* is indented at being available for most embedded systems that do not have build tools. For this reason the original way of controlling *ftrace* is through and interface based on the *debugfs*. This interface allows for configuring and running the tracer, as well as displaying the output, by the utilizing bash commands such as *echo* and *cat*.

3.4 Summary

In this chapter I presented, in detail, the methodology applied in this project, which was inspired by an, iterative, agile development process. I also mentioned some methods and tools which were important for each phase of the project, and how the methodology became very useful while collaborating with a fellow student. Lastly, gave a more in-depth description of a set of tools which played an important role for the outcome of the thesis.

Chapter 4

Design and Implementation

In this chapter I present how the meta congestion controller was designed and implemented. Specifically, I present a set of architectural decisions that were made based on the requirements. In addition to this I describe how the meta congestion controller may be used in real life example, how it was tested and debugged, how errors were handled, and what license it is released under. Finally, I discuss some shortcomings, and how they may be solved.

4.1 Requirements for the Implementation

The first research question (RQ1) for the thesis is:

> *How should a meta congestion controller for DA-LBE transport services be designed and implemented for it to be considered a stable, long-term, solution suitable for the Linux operating system?*

To answer this question I define a set of requirements for implementing the meta congestion controller, that when fulfilled, should provide an adequate answer to the question.

- The meta congestion controller must follow the common conventions used for Linux kernel development.
- The meta congestion controller must fit into the existing architecture of the Linux kernel with minimal changes.
- The meta congestion controller must be thoroughly tested and debugged.

4.2 Code Convention

There are many common conventions used in modern software development, and all vary based on the project and the programming language being used.

An example of such a convention is the one used for development in the Linux kernel [49, *process/coding-style.html*]. The source code of the DA-LBE meta congestion controller makes an attempt at following these guidelines as much as possible in an attempt to keep the structure and style of the code true to that which is used in the mainline kernel.

In addition to this, the source code, which is heavily based on the algorithms by Hayes *et al.* [35] and chapter 2, makes and effort to follow the flow of the algorithms as much as possible. In my opinion, this increases the comparability between the written material and the source code, making for a more pleasant code development experience.

4.3 Architectural Decisions

In this section I discuss the important architectural decisions for the DA-LBE meta congestion controller, referred to as **TCP-DALBE (Dalbe)** in this chapter, and how I implemented them.

4.3.1 Architecture

Generally speaking, Dalbe is implemented as a loadable kernel module [38] which interfaces the TCP congestion control interface [15]. This interface is a *C* structure of function pointers, which are referred to as *hooks*.

This allows it to be loaded manually after the kernel has booted successfully, while seemingly being recognized as a genuine congestion controller by the OS. However, considering that Dalbe is a *meta* congestion controller, the architecture is a bit more intricate than that of a typical congestion controller.

The following are five architectural considerations which are important to pay attention to when implementing a meta congestion controller;

1. Where does the meta congestion controller fit into the networking stack of the Linux kernel?

2. How does the meta congestion controller handle the underlying congestion controller?

3. How does the meta congestion controller maintain metadata for each connection?

4. In what way(s) can the meta congestion controller be configured?

5. How does the meta congestion controller allow for real numbers in the absence of a Floating-Point Unit (FPU)?

```
struct tcp_congestion_ops dalbe_ca_ops __read_mostly = {
    .init = dalbe_init,
    .release = dalbe_release,

    .ssthresh = dalbe_ssthresh,
    .cong_avoid = dalbe_cong_avoid,
    .set_state = dalbe_set_state,
    .cwnd_event = dalbe_cwnd_event,
    .in_ack_event = dalbe_in_ack_event,
    .undo_cwnd = dalbe_undo_cwnd,
    .pkts_acked = dalbe_pkts_acked,
    .get_info = dalbe_get_info,

    .setsockops = dalbe_setsockops,
    .getsockops = dalbe_getsockops,

    .owner = THIS_MODULE,
    .name = "dalbe",
};
```

Figure 4.1: A listing of the functions implemented by the meta congestion controller from the TCP pluggable congestion controller interface.

Placement in the Network Stack

As Dalbe interfaces the TCP congestion control interface in the Linux kernel, it is part of the existing architecture of the networking stack. Dalbe utilizes some of the function hooks of the plugggable congestion controller interface to perform its calculations and manipulate the underlying congestion controller. Dalbe does in general not need to implement all these functions for it to be able to make correct calculations of the DA-LBE values, however it is important that Dalbe implements all the functions which the underlying controller implements, as ignoring some of these may lead to undefined behaviour.

Figure 4.1 shows what function hooks Dalbe implements. The trigger for these function hooks may depend on what state the TCP connection is in, if it is an outgoing or incoming packet, system call, interrupt, or if an error occurs. This makes the architecture quite complex and difficult to debug, and for this reason I have added a set of flow charts that illustrate simplified examples of when these functions may be invoked by the TCP process.

Figure 4.2 gives an insight to the more general view of the architecture. If one of the functions is invoked, Dalbe may perform any needed calculations (*e.g.* altering RTT or packet accounting) before calling the same function on the underlying congestion controller. In other cases Dalbe relies on the underlying to perform its calculations (*e.g.* updating CWND and SSTHRESH for calculating delay based congestion events) before Dalbe is

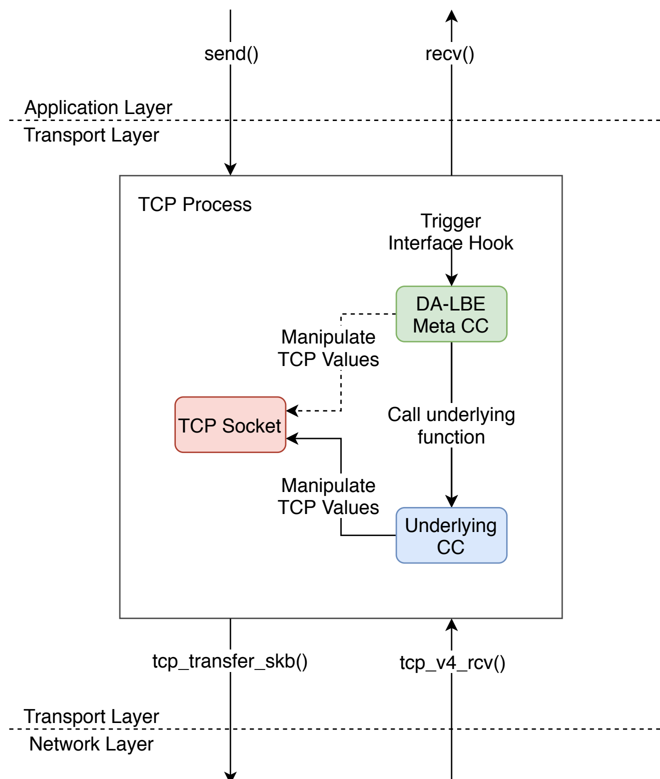


Figure 4.2: A model of how the meta congestion controller fits into the transport layer of the networking stack in Linux. Each time there is an outgoing or incoming packet, it may trigger one of the meta congestion controller functions. If this is the case, the function may manipulate the TCP socket directly, either before or after invoking the underlying congestion controller. The underlying congestion controller may in turn also manipulate the TCP socket before returning to its caller; the meta congestion controller.

able to perform any reasonable calculations. Both the underlying congestion controller and Dalbe may alter the values stored in the socket structure.

Keeping Track of Metadata

Each Dalbe connection needs to keep track of a certain set of values (metadata) which are used to update the weight *etc.* for each interval, see algorithm 1 and 3 in chapter 2. Dalbe does so by keeping a unique structure, *struct dalbe*, for each connection.

These structures are stored in a hashtable which is created by Dalbe when it is inserted into the kernel using *insmod*. After the unique structure for the connection has been allocated, Dalbe places it into the hashtable using the socket address as the key. The reason for this is that the socket (*struct sock *sk*) [73, *include/net/sock.h*], which is unique for each connection, is passed in as a parameter for each hook that Dalbe implements from the pluggable congestion controller interface, and thus it is quite suitable for identifying the connection. To aid the 32-bit hashing algorithm on a 64-bit system we use only the 32 lower bits of the socket address. Figure 4.3 illustrates how the structure is stored in the hash table using *hash_add*, while figure 4.4 illustrates how it fetched using *hash_for_each_possible* from a call to one of the other functions.

Handling Underlying Congestion Controllers

Dalbe keeps track of the underlying congestion controller similarly to how the TCP connection keeps track its congestion controller. As every congestion controller in Linux is registered to a simple linked list in the kernel, Dalbe can take advantage of this by looking up the underlying congestion controller by name in this list. The list holds a pointer to each structure that contains the hooks which each congestion controller implements. When Dalbe has fetched this pointer, it can keep it stored in its own metadata structure until the connection is closed. Figure 4.3 illustrates how Dalbe reads and writes from this list. Note that Dalbe will also be part of this list after initiation.

4.3.2 Configuration Possibilities

Dalbe may be configured either at the time the module is loaded, or at run-time. The following subsections describes how this is made possible.

Module Parameters

Configuring Dalbe using module parameters allows for setting values that should not be changed too often for a kernel module. In this implementation only a small set of values are made available as parameters, which server the

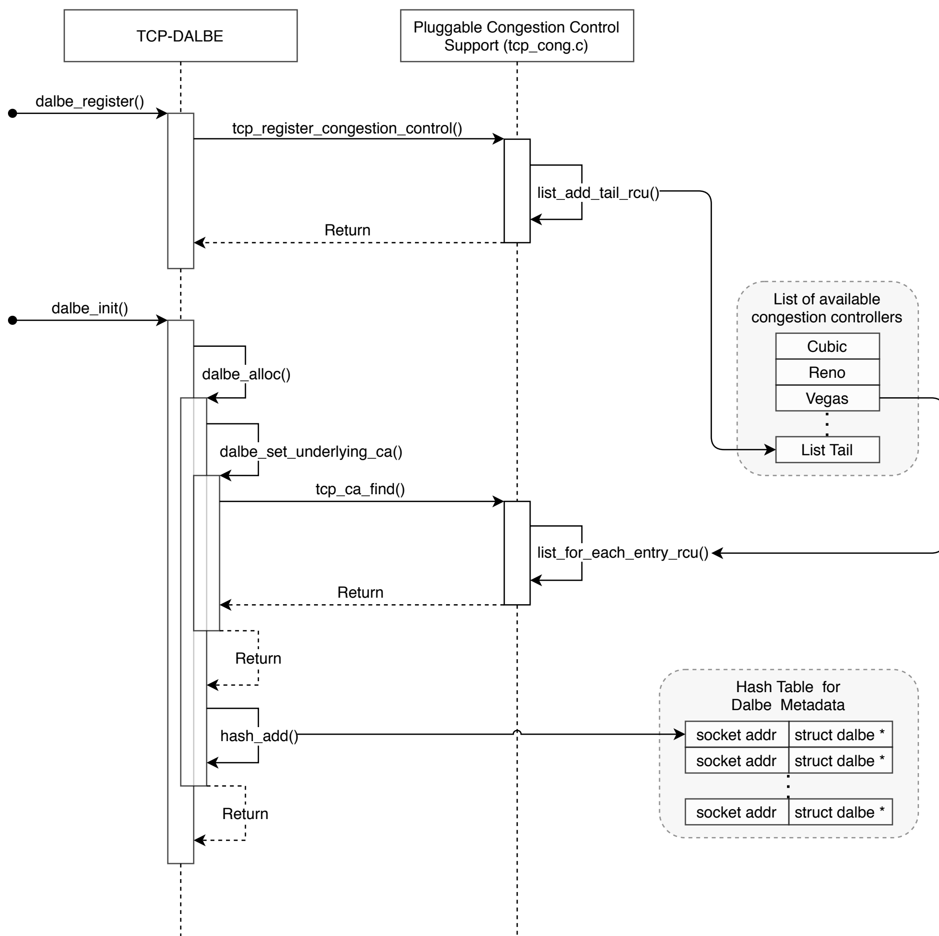


Figure 4.3: This model illustrates how the meta congestion controller is registered in the kernel as a congestion module through the *dalbe_register* function, how the meta congestion controller is able to fetch a pointer to the underlying congestion controller through the *tcp_ca_find* function call, and how the Dalbe structure is stored in the hashmap using *hash_add*.

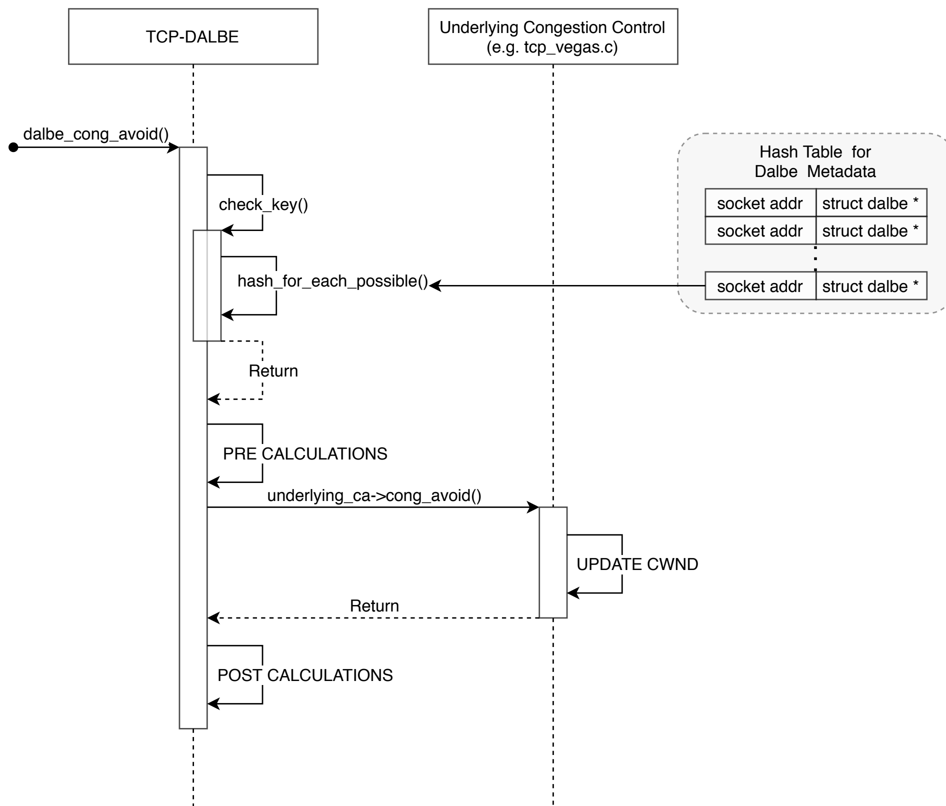


Figure 4.4: This model illustrates an example of what a call to one of the Dalbe meta congestion controller functions involves, in this case the *dalbe_cong_avoid* function. The first action performed is to fetch the *dalbe struct* from the hash map. If this is successful, the meta congestion module may perform some calculations before invoking the corresponding function of the underlying congestion controller. Upon return from the underlying congestion controller, the meta congestion controller may perform some additional calculations before returning.

role of default values that are not intended to be changed after the module has been inserted.

Acquiring the α parameter from Vegas One problem I encountered when developing the model based control for Vegas was that the α parameter, which was needed by model, was not possible to fetch from inside the meta congestion controller module. The reason for this being that the Vegas implementation in the Linux kernel, also a kernel module, has both α and β set as static variables [73, *net/ipv4/tcp_vegas.c*]. The only way to alter these parameters after the Vegas module was loaded, is through the use of *sysctl* [50, *sysctl(8)*].

After researching for a proper solution to perform something similar to *sysctl* from inside our kernel module[1], I came to the conclusion that the solution would be very unclean and "hacky". In fear of a hacky solution that would be conflicting with the requirements for the implementation, I settled on a solution where the user has to be responsible for setting the Vegas α and β values manually using *sysctl*.

With this solution I also had to implement the Vegas α value as a module parameter for Dalbe.

Custom Socket Options

Communication with kernel modules from user space is usually done via some kind of API or I/O control [50, *ioctl(2)*]. In my case I was fortunate enough to have a well defined API in my possession. The socket API, provided by the Linux OS, allows the application to configure the connection properties quite well using socket options [50, *setsockopt(2)*, *getsockopt(2)*]. Though the socket API works quite well, it was limited in terms of passing information directly to the congestion controller module.

For this reason I expanded the TCP congestion controller interface with two more function hooks; *setsockopts* and *getsockops*. This set of hooks allow the congestion controller modules to define custom socket options specific to the controller, which is useful if the user wants to send a set of options directly to the congestion controller.

4.3.3 Fixed Point Operations

Dalbe requires the used of real numbers for its DA-LBE related calculations. However, the use of real numbers, such as floating point, in the kernel is quite frowned upon as this is very architecture specific and may lead to a very buggy and slow system. For this reason Dalbe utilizes fixed point numbers, which is a solution for fractional numbers using integers. However,

[1] I even resulted in posting a question on StackOverflow.com in hopes of finding a good solution for this problem, but with little success.

the use of fixed point operations can make the code quite cluttered and difficult to debug, if it is not implemented with a well defined structure. For this reason Dalbe introduces a set of functions that aid the developer when performing these fixed point operations.

Function Definitions

Dalbe introduces two basic arithmetic operations for fixed point calculations; multiplication and division. For multiplication *sq_mul* and *uq_mul* are defined, and for division *sq_div* and *uq_div*.

sq_mul* and *uq_mul allow for multiplication of two fixed point numbers. The functions allow for the two numbers to have different scale factors. However the result will always scale to the type with the largest integer part, to preserve the integer part.

sq_div* and *uq_div allow for division between two fixed point numbers. Also here, the functions allow for the two number to have different scale factors, and the result will always scale to the type with the largest scale factor.

Choosing inline function instead of macro Initially, Dalbe used GCC Macro one-liners for its fixed point operations, in an attempt to introduce as little computational overhead to the operations while still having function-like syntax. However, this produced code that was difficult to read and debug. For this reason, the fixed point operations where expanded into *inline* functions instead. There where two major benefits of this design decision; (1) The code became easier to read and debug, and (2) it allowed for a little more complexity within the function. In terms of computational overhead, this introduces no additional overhead when done correctly [28].

I chose to add some more complexity when expanding to inline functions as I then could work with fixed point numbers with variable scale factors. This could of course also have been achieved with macros, using *do-while* blocks. However, the use of macros in such a way would even further complicate the code as multiple lines would be inserted where the macro would be used.

Type Definitions

Dalbe introduces four fixed point types, with two different scale factors. Two signed types; *sq16* and *sq30*, and two unsigned types *uq16* and *uq30*. These types are just type definitions of the variable types; *s16*, *u16*, *s30*, *u30*, and the compiler does not differentiate between these types.

```c
/*
 *  Macro based multiplication:
 */
#define _Q_MUL_ROUND(a, b)    \
    ((a * b) & 1) * Q_SCALE_FACTOR
#define UQ_MUL(a, b) \
    (u32)((((u64)a * (u64)b) + _Q_MUL_ROUND(a, b)) /
    Q_SCALE_FACTOR)

/*
 *  Inline based multiplication:
 */
static inline u32 uq_mul(u32 a, u32 a_scale_factor, u32 b,
    u32 b_scale_factor)
{
    u32 scale_factor = MAX(a_scale_factor, b_scale_factor);
    u32 round_part = ((a * b) & 1) * scale_factor;

    return (u32)((((u64)a * (u64)b) + round_part) /
    scale_factor);
}
```

Figure 4.5: A listing showing the difference between a GCC macro based multiplication operation for fixed point numbers, and the same code written with inline functions and an additional line of complexity.

sq16* and *uq16 Introduce a fixed point number with an integer part of 16 bits and a fraction part of 16 bits. *sq16* allows for signed numbers to range between $-2^{15} = -32768$ and $2^{15} - 1 = 32767$, as the most significant bit is reserved for the two's compliment of the number, and *uq16* allows for unsigned numbers between 0 and $2^{16} - 1 = 65535$. The precision of this type is defined by its fraction size of 16, which allows for a precision of $2^{-16} \approx 10^{-5}$.

sq30* and *uq30 Introduce a fixed point number with an integer part of 2 bits and a fraction part of 30 bits. *sq30* allows for signed numbers to range between $-2^1 = -2$ and $2^1 - 1 = 1$, as also here the most significant bit is used for the two's compliment of the number, and *uq30* allows for unsigned numbers between 0 and $2^2 - 1 = 3$. The precision of this type is defined by its fraction size of 16, which allows for a precision of $2^{-30} \approx 10^{-9}$.

The choice of scale factors are based on my belief that these two should be sufficient to handle the calculations performed by Dalbe with good precision without over complicating the source code too much. The Q16.16 based numbers are meant as an all round solution where the numbers are not predictable, while the Q2.30 based numbers are meant for use on values

between 0 and 1.

Macro Definitions

In addition to the multiplication and division functions, Dalbe introduces a set of macros which can be used for conversion from integers to fixed point and vice versa.

4.4 Changes to the Kernel

Some of the architectural decisions proposed in previous sections required minor changes to the Linux kernel. These changes were kept to an absolute minimum, as introducing unnecessary changes to the kernel is frowned upon, and will decrease the chance of having the work accepted in the next mainline kernel. Only changes that were absolutely crucial for the meta congestion controller to work were added, to the following files:

- [73, ***include/net/tcp.h***] contains part of the changes made for allowing the meta congestion controller to find the underlying congestion controller by name. Also, this is were the TCP congestion controller interface is defined, which was expanded to allow for custom socket options. This was already a part of the previous work done on the meta congestion controller by E. Band and and D. Hayes. However, the socket options have been refurbished in this work, by attempting to make them fit better into the architecture, implementing proper move operations between user space and kernel space, and by implementing adequate error handling.
- [73, ***include/uapi/linux/dalbe.h***] contains a set of constants defining the available, custom, socket options for the DA-LBE meta congestion controller. The placement makes them available from user space after the Linux headers have been installed correctly.
- [73, ***include/uapi/linux/tcp.h***] adds one socket option constant for the TCP layer. This option is used for informing that the socket option used is in fact a custom option destined for the custom socket option hooks.
- [73, ***net/ipv4/tcp.c***] contains the changes to the *do_tcp_setsockopt* and *do_tcp_getsockopt* procedures, which invoke the *setsockops* and *getsockops*, if the option is set.
- [73, ***net/ipv4/tcp_cong.c***] contains the remaining changes that allow for the meta congestion controller to find the underlying congestion controller by name. Specifically, this is were the function is exported in a way that makes it available in the Linux kernel.

At the time being Dalbe builds upon a custom version of Linux 5.4 from the *net-next* tree by D. Miller [53]. The decision of using version 5.4 is simply a result of choosing a stable version to work with during the testing and experimentation, and is not absolute [2].

A patch with the minor changes to the networking code described in this section is provided together with the source code for Dalbe.

4.5 Usage

When the custom kernel has been built and installed, applications can be built using the existing socket API [50, *socket(2)*], and there are no other requirements for the developer other than to make themselves familiar with the documentation of the kernel module and the DA-LBE framework, given that they are familiar with network programming. Following is an example of a simple client written in C, utilizing the DA-LBE framework with MBC as the weight policy and Vegas as the underlying congestion controller:

Figure 4.6 illustrates what little extra effort has to be put into making a client work with DA-LBE. And similarly, figure 4.7 illustrates the little extra effort has to be put into making a server work with DA-LBE. The only change is the socket options, which have to be set for the connection. On the client, this is done by directly applying the socket options on the main socket, while on the server the socket options have to be applied on the client socket (*c_sock*). Note that the client and server can independently set the socket options if either wants to use DA-LBE meta congestion control.

For more complete examples, two server-client examples can be found in the repository of the test bed [72]; one is written for *Python3* and the other is written for *C++*. In addition to this, as a server-client setup made specifically for unit testing of the socket API can be found in the test repository [70].

4.6 Testing

The methods used for testing kernel modules vary depending on their purpose and to which sub-system they belong. There is a large variety of frameworks for performing various tests on the Linux kernel such as *Linux Test Project* [23] and *Autotest* [22], however, in many cases they may be to general or do not target specific parts of the code that are of interest for testing. For this reason developers tend to make their own tests which are tailored to fit their specific purpose, *e.g.* utilizing techniques such as *ioctl* [50, *ioctl(2)*] to target specific functions within their code.

[2] To create new patches requires little effort, however applying them together with re-building every time a new version is released is quite cumbersome when working within a time frame.

```c
/* Includes for socket operations etc. */
#include <linux/dalbe.h>

int main(void)
{
  int sock = socket(AF_INET, SOCK_STREAM, 0);

  setsockopt(sock, IPPROTO_TCP, TCP_CONGESTION,
             "dalbe", strlen("dalbe"));
  setsockopt(sock, IPPROTO_TCP, DALBE_UNDERLYING_CA,
             "vegas", strlen("vegas"));
  setsockopt(sock, IPPROTO_TCP, DALBE_TARGET_DEADLINE,
             1300, sizeof(unsigned long));
  setsockopt(sock, IPPROTO_TCP, DALBE_DATA_SIZE,
             1000000, sizeof(int64_t));
  setsockopt(sock, IPPROTO_TCP, DALBE_W_POLICY,
             0, sizeof(int)); /* 0 = MBC, 1 = PID */

  struct sockaddr_in saddr;
  saddr.sin_family = AF_INET;
  saddr.sin_port = htons(6655);
  saddr.sin_addr.s_addr = inet_addr("server.address");

  connect(sock, (struct sockaddr *)&saddr, sizeof(saddr));

  /* Send the specified amount of data to server .. */

  return 0;
}
```

Figure 4.6: A simple example of a client written in C. The client utilizes the socket option interface (*setsockopt*) to configure the connection for DA-LBE transfers.

In terms of testing congestion controllers and other networking specific features within the Linux OS, making custom tests seems to be the common case. However, being able to test all parts of the congestion controller required setting up connections and actually transfer data between two, or more hosts.

4.6.1 Unit Tests

One way Dalbe was tested was through unit tests using the *Google Test* [29] framework for C++. By doing so I could generate light weight tests that help me identify errors and memory leaks. I could also extract parts of the code, *e.g.* the fixed point operations, and verify their correctness in user space before applying them to the kernel module.

```c
/* Includes for socket operations etc. */
#include <linux/dalbe.h>

int main(void)
{
  int sock = socket(AF_INET, SOCK_STREAM, 0);

  struct sockaddr_in saddr;
  saddr.sin_family = AF_INET;
  saddr.sin_port = htons(6655);
  saddr.sin_addr.s_addr = inet_addr("server.address");

  bind(sock, (struct sockaddr *)&saddr, sizeof(saddr));
  listen(sock, 1);

  struct sockaddr_in addr;
  socklen_t addr_len = sizeof(struct sockaddr_in);
  int c_sock =
      accept(sock, (struct sockaddr *)&addr, &addr_len);

  setsockopt(c_sock, IPPROTO_TCP, TCP_CONGESTION,
             "dalbe", strlen("dalbe"));
  setsockopt(c_sock, IPPROTO_TCP, DALBE_UNDERLYING_CA,
             "vegas", strlen("vegas"));
  setsockopt(c_sock, IPPROTO_TCP, DALBE_TARGET_DEADLINE,
             1300, sizeof(unsigned long));
  setsockopt(c_sock, IPPROTO_TCP, DALBE_DATA_SIZE,
             1000000, sizeof(int64_t));
  setsockopt(c_sock, IPPROTO_TCP, DALBE_W_POLICY,
             0, sizeof(int)); /* 0 = MBC, 1 = PID */

  /* Send the specified amount of data to client .. */

  return 0;
}
```

Figure 4.7: A simple example of a server written in C. The server utilizes the socket option interface (*setsockopt*) to configure the connection for DA-LBE transfers. Note that this is done on the client socket, after the connection with the client has been accepted.

User Space API Tests

In terms of quickly identifying bugs and errors that would lead to the process, kernel module, or even the entire system failing, I created a simple test suite that targeted the socket API in Linux. The tests generated a local server and client which connected with each other and performed a simple transfer. By doing so I could quickly verify that the kernel module was running, and I could check for memory leaks. Another great benefit of these tests was that I could verify that the custom socket options where performing correctly, by invoking them and checking within the kernel module that the correct values had in fact been.

Fixed Point Library Test

The unit test framework was also very useful for when I wrote the functions for the fixed point operations. I wrote and tested all the needed functionality in user space before applying them to the kernel module [3]. This allowed me to do more experimentation without being afraid to break the system, and it helped me explore the limitations of the fixed point operations.

4.7 Debugging

The unit tests were quite helpful for checking that everything was running and that I was not making any major mistakes. However, they did not tell anything about the internal behaviour of Dalbe, such as the correctness of the DA-LBE framework operations, where a memory leak originated from, or what caused a crash. For this reason I had to rely on other methods such as custom logging and debugging tools available for the Linux OS

4.7.1 Debugging by Logging

To verify that the DA-LBE calculations were correct, I applied the same type of logging system used by Wallenburg for creating debug plots [74], but with some minor modifications. These values were printed by the kernel module during testing using *printk*, which behaves quite similar to *printf* [50, *printf(3)*], but the output is placed in a kernel buffer. The output can be read using *dmesg* [50, *dmesg(1)*]. This process is a little more involved than using user-space print functions, but nevertheless the results are quite similar.

Figure 4.8 illustrates an example of how the debug values were plotted for model based Vegas. Each graph shows statistics from relevant variables used in the DA-LBE calculations, sampled at a 10 second interval. This

[3] Initially, before I made the decision of using inline functions, I had fixed point operations in our kernel module. However, it was all macro based and messy.

specific example is from one of the network performance experiments described in chapter 6. The experiment runs a DA-LBE flow with a deadline of 1300 seconds, utilizing Vegas as the underlying congestion controller and MBC as the weight policy, while competing with other BE traffic flows. Note that the following list of figures only indicates how the debugging graphs should be interpreted, while the actual analysis of these figures can be found in chapter 6.

- Figure 4.8a illustrates the relationship between w, μ, ϕ, and the *backoff* parameters. As w moves towards 1, the aggressiveness of the DA-LBE flow should increase. If w reaches 1, the *backoff* value should be above 0. ϕ is the probability of congestion, and μ is the product of ϕ and w which is used to alter the RTT.

- Figure 4.8b illustrates the relationship between the average rate and the target rate. Also plotted is the model error, which relies heavily on the rates, and indicates the precision of the model at each interval.

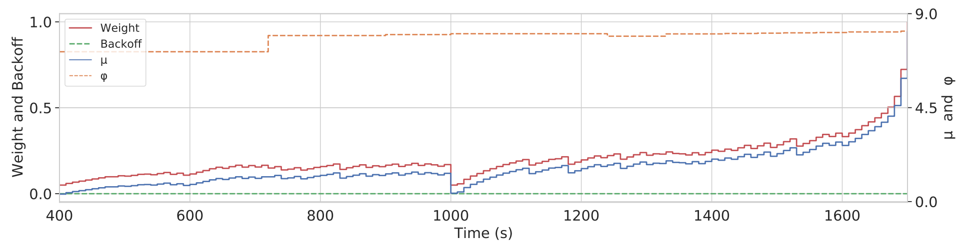
- Figure 4.8c illustrates the relationship between the measured and modeled queuing delay, and which also gives an impression of the precision of the model for each interval.

- Figure 4.8d illustrates the relationship between the measured average RTT, and the base RTT, which is what Dalbe uses to calculate the model queuing delay. In addition to this, a smoothed average of the actual RTT for each interval is plotted, which helps on getting an impression of the network.

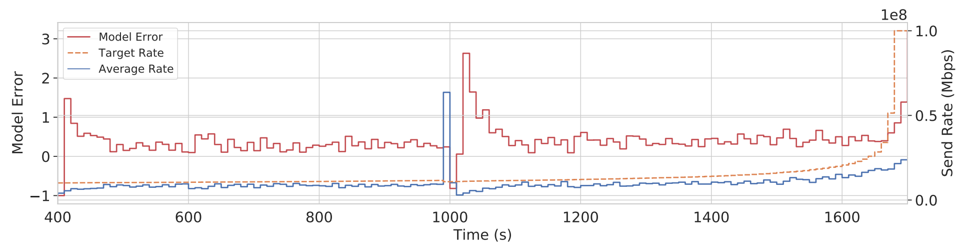
- Figure 4.8e illustrates the congestion window and slow start threshold. These values should give an indication of the actual aggressiveness of the DA-LBE flow.

4.7.2 Debugging Memory Usage

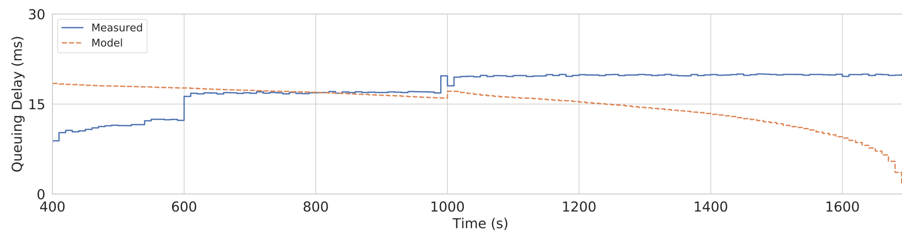
To detect memory leaks caused by the kernel module I used *kmemleak*, which works similar to a tracing garbage collector but without freeing memory [49, *dev-tools/kmemleak.html*]. *Kmemleak* runs in the background and monitors the memory usage of the kernel. If a leak is suspected it traces it back to the source and makes an entry in a debugging log located in the file system. This was very useful for me as I had *kmemleak* running while conducting tests on a virtual setup. After each test I could check the logs to see if any leaks had been detected, and if so I could trace them back and fix them.



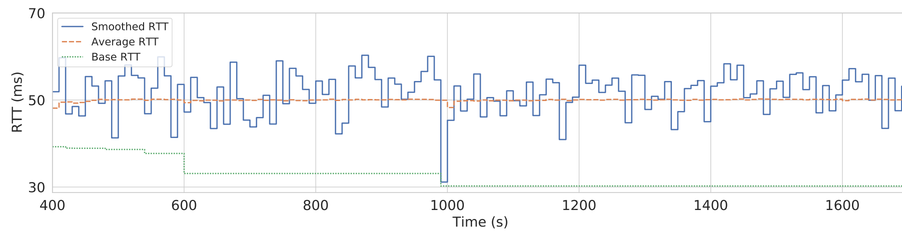
(a)



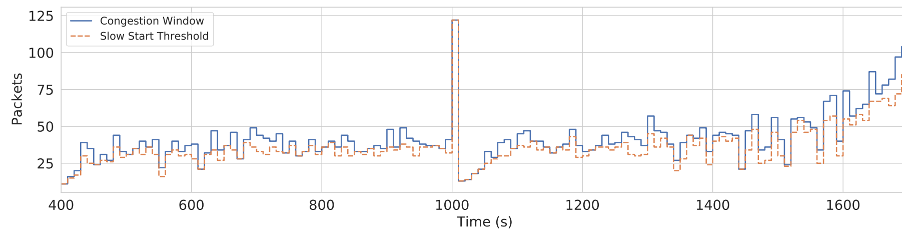
(b)



(c)



(d)



(e)

Figure 4.8: An example of the graphs used for debugging. Each sub-graph displays a certain set of metrics from a DA-LBE flow in a network performance related experiment.

4.7.3 Debugging Kernel Panics

In the event of the system crashing due to an unexpected behaviour, I used *kdump* and *crash* utilities to trace back the cause. This was made possible by building the kernel with configurations set for debugging, and having *kdump* running as a service while the system crashed. If the system would crash, kdmup would log a summary of the so called kernel panic. With this log I could run *crash*, which is quite similar in use as *gdb*, and allows for tracing the cause of the crash after reboot.

In most cases I did not have to use these tools to trace back the kernel panics, as their cause would often be quite trivial to find in the code. However, in those few cases were the crash was not so obvious to trace back by my self, these tools were very useful.

4.8 Error Handling

One major concern when implementing Dalbe was handling errors correctly and avoiding unexpected behaviours, as they could lead to the whole system crashing. Dalbe handles errors according to the common conventions for the return type of the function. If the return type is not void, it returns the correct error value according to the error which occurs.

Dalbe also makes an attempt at handling all unexpected behaviour. Though this is very difficult to predict in some cases. One common case of undefined behaviour that leads to a kernel panic is division by zero. This is however quite trivial to prevent by always checking to see if value of the divisor is zero, and avoiding the calculation if it is so.

4.9 Licensing

Generally, the source code of Linux is released under the GPL license, which allows it to be freely used and modified both privately and commercially. Dalbe is released with a dual license consisting of GPLv2 and Berkeley Software Distribution (BSD)-3-Clause. The main reason for this is that the BSD licensing is that it allows for the code to be rewritten for other operating systems, but it also prevents large vendors (*e.g.* Microsoft and Apple) to use the software as their own without recognizing the original authors. By releasing it under both licenses, the code can be part of the open source project of Linux, while protecting the original work by D. Hayes *et al.* .

4.10 Shortcomings

Though Dalbe provides a functional meta congestion controller for the Linux operating system, in its current state it has some shortcomings. The recognized shortcomings are presented in this section, with some thoughts on how they can be fixed.

4.10.1 Available Model Based Controllers

At the point being there is only one model based controller implemented in Dalbe, and that is for Vegas. Implementing models for DA-LBE requires an underlying congestion controller with a well defined model of congestion. A model for Cubic was defined by Hayes *et al.* [35]. However, this requires the used of a cube root, which is quite difficult to achieve without floating point operations, and was never a major goal for this work.

However, in the testing repository for the meta congestion controller [70], I have added a possible solution for solving this problem utilizing the fixed point operations defined in previous sections. This solution combines a *Binary Search* with one iteration of *Newton Raphson* method for solving cube root. I my opinion, this is an almost adequate solution as it scales to $O(\log n)$, and may be worth testing. However, the source code for Cubic utilizes a table of predefined values for cube root, which is faster and should be considered as a possible solution when building a model.

4.10.2 Support for Loss-Based Congestion Control

A goal for the implementation was to achieve PID based control for all underlying congestion controllers. I did make an attempt on getting this finished in time, and code for this can be found in the *loss-based* branch of the meta congestion controller repository [71]. However, due to limited time, I had to focus on getting other parts of this thesis finished by the deadline, and for this reason this was put aside. The current meta congestion controller supports both MBC and PID based control for Vegas, but PID is not available for loss-based congestion controllers.

4.10.3 Modular Architecture

The current architecture of Dalbe works well for the current functionality it provides. However, the current architecture is not very modular. This concern of modularity arose when I considered how the meta congestion controller module would be expanded to support MBC for multiple congestion controllers. I chose to continue with the architecture we resulted with, however I recognized the need for a better architecture in the future. A thought in the direction of a possible solution for this would be to take

inspiration from the pluggable congestion controller interface [15], and implement something similar for allowing multiple models.

4.10.4 Metadata from Underlying Congestion Controllers

One problem already explained in previous sections was that the α and β values from Vegas were hard/impossible to fetch from within our kernel module. Though I have a functioning solution utilizing the kernel parameters at the moment, I realize that this problem may occur again when implementing models for additional congestion controllers. The solution to this may actually be to make the needed variables from the underlying congestion controllers available by changing the source code.

4.10.5 Current use of Fixed Point Types

The current meta congestion controller utilizes only the Q16.16 fixed point types, as they provided a simple adequate solution, and the code became more complicated and hard to debug when I combined Q16.16 with Q2.30. I have yet to witness any major problem by making this decision. However, I believe that when further work is made on the meta congestion controller, it can be useful to have the choice available for combining the two types.

4.10.6 Passing Fixed Point Values From User Space to Kernel Space

At the moment the tuning parameters for PID based control may be passed as socket options and/or kernel parameters to the module. These parameters are based on real numbers, and thus need to utilize the fixed point values, something that is not standardized int user space nor kernel space. I solved this by creating a conversion function from floating point to fixed point when implementing the client and server for experimentation [72], however this may not be a very good log-term solution. I believe that the best solution for this is to remove the possibility for configuring these values at at initiation of the module and at run-time, as they are considerably difficult to tune, and once the correct values are found, they usually stay unchanged.

4.11 Summary

In this chapter I presented how the meta congestion controller, referred to as Dalbe, was designed and implemented. From as set of requirements, derived from the first research question, I showed how the implementation took shape. More specifically, through a set of architectural decisions I described how the Dalbe fit into the existing architecture of the networking stack of the Linux kernel, how it handles underlying congestion controllers, how metadata is maintained on a per-connection-basis, in what way it may

be configured, and finally, how the absence of a FPU handled. In addition to this I described how some of the architectural decision resulted in some minor changes to the Linux kernel, and I gave an example of how the meta congestion controller could be used in server/client example. I also gave an introduction to how the meta congestion controller was tested and debugged, and the licenses it is released under. Finally, I discussed some shortcomings, and how they may be solved.

Chapter 5

Test Environment

This chapter describes how I, together with a colleague, built, configured, and verified a test environment for testing and evaluating the DA-LBE meta congestion controller. Two test environments are presented; (1) A virtual test bed used for unit testing and debugging, (2) a hardware based test bed for testing and experimentation in a more realistic environment.

5.1 Requirements

The second research question (RQ2) defined for this thesis is;

> *How can a test environment be built and configured to be suitable for thorough testing and experimentation of the DA-LBE transport services?*

From this question I derive a set of requirements which I consider crucial to satisfy for the research question to be answered. The requirements are also meant to guide me while building, configuring and verifying the test environment.

- The test environment must be stable and provide reproducible results.
- The test environment must allow for network emulation of different topologies.
- A software suite must be provided, which allows for easy construction and execution of experiments.

5.2 Collaboration

It is important to state that some parts of the setup and development of the test environments and the software suite was done as a collaboration with colleague, Mattis Bratland, referred to as Mattis in this thesis. During this chapter I will make notice of what parts where done solely by me (I), in collaboration (we), or solely by Mattis.

5.3 Testing on Virtual Machines

One of the key difficulties of working on a kernel module is that if an error is not handled correctly it will in most cases crash the entire system. For this reason, in the early stages of testing, when the purpose of my tests was mainly to make the meta congestion controller work correctly and locate bugs, I utilized a virtual test bed on my personal desktop. This proved to be very useful as the virtual machines could just be rebooted and restored if they encountered a bug that would lead to system failure. Another good reason for using this kind of setup was to be able to monitor live network traffic between the virtual machines using Wireshark [27].

The virtual test bed consisted of three VirtualBox [19] Virtual Machine (VM)s running Ubuntu Server 18.04 LTS. All the VMs were attached to the host machine using Network Address Translation network so that I was able to control them using SSH [50, *ssh(1)*]. The router node was connected to both the edge nodes using VirtualBox's Internal network and was connected to a virtual interface on the host machine using a Bridged network [20, *Chapter 6*].

Though I could probably have made due with just virtual interfaces on our host machine, I wanted to work with an environment that was similar to that which I would conduct my experiments on later in the process. In this way, when I later moved over to the hardware based test environment, the scripts and configuration files used for the virtual setup could be applied with little to no effort.

5.3.1 Unit Testing with Virtual Machines

In the previous chapter I described how I unit tested the user space API of the meta congestion module. The benefit of doing this on a virtual machine was the possibility for using Snapshots [20, *Chapter 1*] of the running kernel. When (or if) I encountered a bug that led to system failure I could simply reboot and restore the machine at the point before the crash. This made for quite the robust and efficient work environment when it came to testing the module.

5.3.2 Debugging with Virtual Machines

When I was sure that the meta congestion module would not fail from calls to the user space API, I could use the virtual machines to debug the performance and edge cases. To debug edge cases within the module I had to have the module running a connection for several seconds, maybe minutes, enough to trigger all the functions, and introduce congestion-specific behaviour such as loss and changes in the queuing delay. For this to be more efficient, the network traffic and performance could be monitored live using networking tools such as Wireshark and Tcpdump [50, *tcpdump(8)*]. Figure 5.1 shows how this virtual test environment was connected to the host machine.

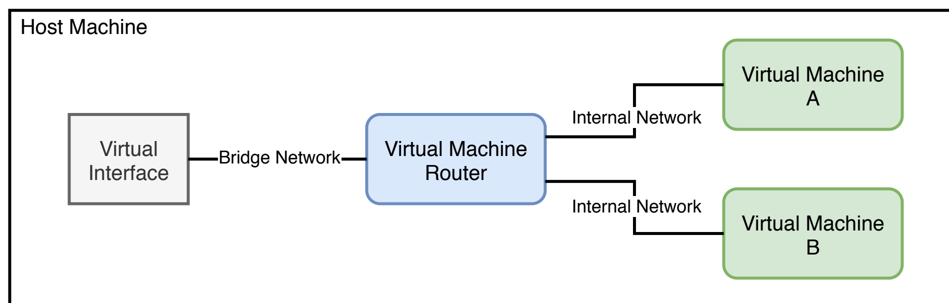


Figure 5.1: An illustration of how the network was set up between the host machine and the virtual machines.

5.3.3 Drawback of Testing on Virtual Machines

Though the virtual test bed was very good for certain tasks, such as debugging and unit tests, it was not very suitable for running the actual experiments. I often witnessed that the host machine would struggle when running the test environment, which would affect the virtual machines negatively, as their performance was highly dependent on the host machine's performance. For this reason I needed something more stable to perform experiments on the meta congestion controller.

Figure 5.2 illustrates this performance issue quite well. Extreme drops in the sending rate can be seen as a kind of "melting" effect in the graphs, which is very noticeable at t = 0 to t = 200 and t = 1800 to t = 2000. During these periods there is only one BE Cubic flow running at the time, and we would expect the the sending rate to be very stable at around 90 Mbps.

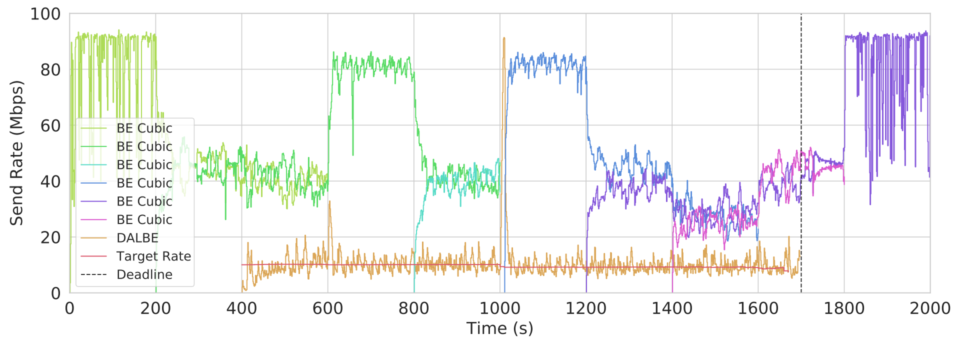


Figure 5.2: An example of how badly the virtual test bed performed when used to run network performance experiments. This was for a Vegas based DA-LBE flow using PID as the weight policy.

5.4 Testing on Hardware

When satisfied that the DA-LBE meta congestion module was behaving correctly with respect to its LBE-ness and that any crucial bugs had been eliminated, I moved over to working with a more stable test environment. I wanted to run experiments identical to those done in previous papers on this topic as well as conducting my own, custom experiments specific for this research. Previous work done on the topic had used a hardware test bed utilizing network emulation to create a dumbbell topology and a two node setup over the Internet. Both were used used for performance and long-term behaviour testing. These were some of the tests I aimed to conduct, however neither the hardware based test environment nor the Internet based test environment was at this point available. Therefore an alternative solution had to be made. The result was to collaborate with Mattis on building a custom, hardware based, test environment on which we hoped to be able to conduct some solid experiments.

5.4.1 Building a Suitable Test Bed

To build our test bed we used leftover parts from an "old" gaming computer, a Gigabit Ethernet Switch, four MONROE nodes [1], and connected it all together with two NICs[2]. The computer parts and switch were provided by Mattis, while the MONROE nodes and NICs were borrowed from Simula Research Laboratories. With this setup I aimed to emulate a simple dumb-

[1]MONROE project is a measurement platform for measurements and custom experimentation on operational mobile broadband networks. We were fortunate enough to borrow four leftover nodes from this project.

[2]The reason for us needing two and not one NIC was due to the test bed being used for testing a meta congestion controller based on Multipath TCP (MPTCP).

bell topology, as shown in figure 5.5, on which I could perform tests and experiments on the meta congestion module.

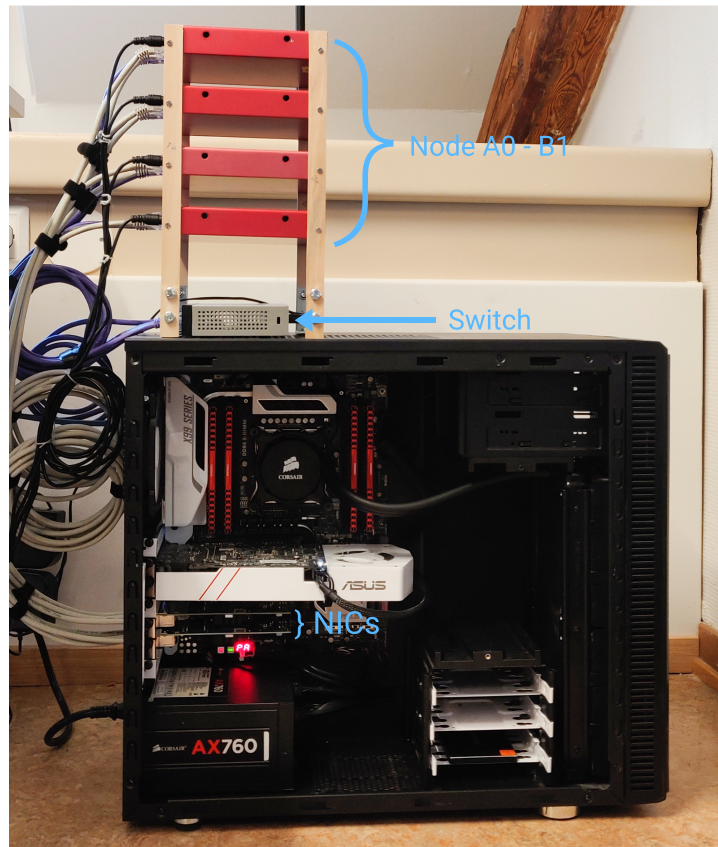


Figure 5.3: An image of our test bed fully assembled. The large black computer is the router machine. Below the white graphics card are two NICs which were used to connect everything together. Mounted on top of the black case are the edge nodes, and below them is the switch used to forward network traffic from the control ports of the edge nodes through the router node.

Our hardware test bed consisted of six components, four edge nodes (A0, A1, B0, and B1), a router node and a switch. The reason for using a switch was to give the edge nodes access to the Internet. As shown in figure 5.4, the switch connects the edge nodes to the router node, which forwards their traffic to the Internet. This was done to shield the more vulnerable edge nodes from direct access from the Internet, while still being able to control the nodes via SSH. The Router was set up with sufficient security to handle direct contact with the Internet. Mattis was mainly involved in configuring the test bed for use on Internet and the security measures involved.

The four edge nodes were statically placed in each their sub-network

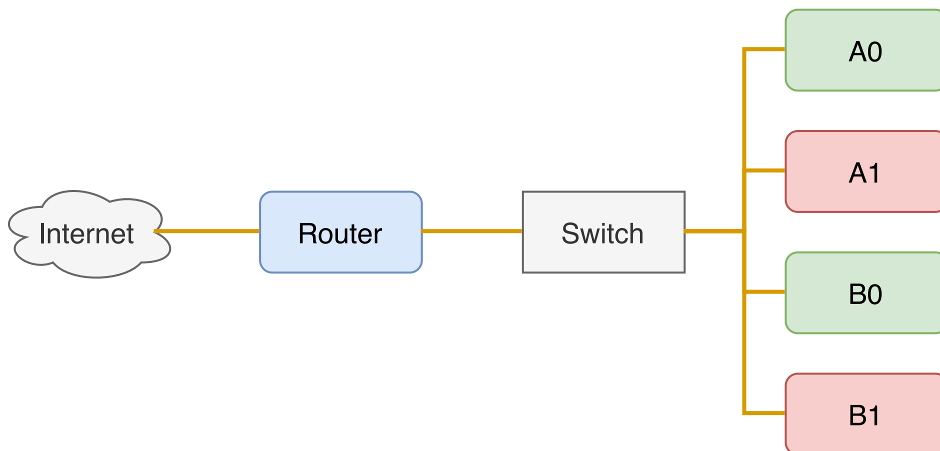


Figure 5.4: A figure showing how our hardware test bed was connected. The orange lines show how the edge nodes were connected to the Internet via the router node.

specified in their networks interface configuration files, and set up with adequate routing tables using ip [50, *ip(8)*], allowing them to reach the other sub-networks. Edge node A0 was set to communicate with A1 and its task was to perform Dalbe transfers. Edge node B0 was set to communicate with B1 and its task was to perform BE transfers as well as generating background traffic for more realistic results. The router node was configured via sysctl [50, *sysctl(8)*] to forward all traffic between the sub-networks.

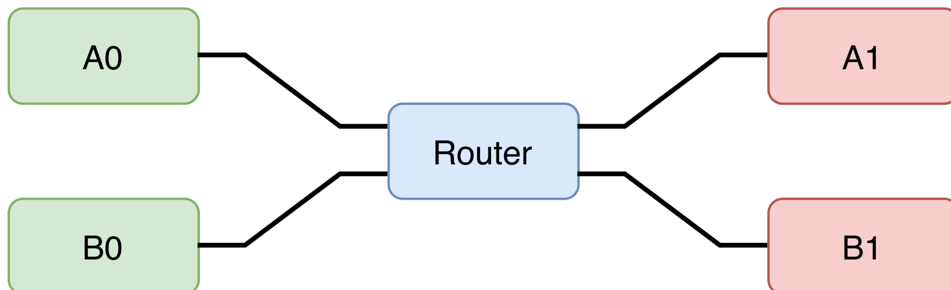


Figure 5.5: A very simple dumbbell topology consisting of five nodes; four edge nodes, two on each side of the router node.

5.4.2 Hardware Setup

The edge nodes were not very powerful, with a 4-core processor running at a maximum of 1GHz and 4 GB of Random Access Memory (RAM). For this reason we made sure to have only the bare minimum of processes running on them at every time. We also made an effort to monitor their memory

usage and the average load of their processors while conducting experiments to make sure we were not overloading them in the process. Their storage capacity was the major issue. With only 16 GB of storage where, in some cases, 80% was in use we had conserve what space we had left. We made sure that only logs that accumulated to small sizes would be stored on them at any time, and clean-ups were done regularly.

On the other hand, the router node was specked with a relatively powerful processor running at a maximum of 3.5 GHz, 16 GB of RAM and 256 GB of Solid State Drive (SSD) storage. We therefore chose to capture network traffic, emulate networks and orchestrate experiments from the router node.

Edge Node	
Processor	AMD GX-412TC @ 1GHz (4 Cores)
Network Card	Intel I210 (rev 03)
Memory	4 GB DDR3 1333MHz
Storage	16 GB SSD SATA3
Operating System	Ubuntu Server 18.04 LTS
Kernel	5.3.0-28-generic & 5.4.0+ (modded for A0)

Table 5.1: Relevant specifications of edge nodes.

Router Node	
Processor	Intel Core i7-5930K @ 3.50GHz (12 Cores)
Network Card(s)	Intel I350 (rev 01)
Memory	16 GB DDR4 2666MHz
Storage	256 GB SSD SATA3
Operating System	Ubuntu Server 18.04 LTS
Kernel	5.5.8 (Tick rate 1KHz & dynticks off)

Table 5.2: Relevant specifications of router node.

5.4.3 Operating System Setup

All nodes ran Ubuntu 18.04 LTS Server edition, however node A0 and the router node had to run specific kernels. Node A0 was set up with a slightly modified kernel version 5.4.0 from the *net-next* [53] development branch, which is described in chapter 4. The kernel on the router node was customized to run with a periodic tick rate for 1KHz and with dynamic ticks (*dynticks*) turned off.

In the Linux operating system the tick rate defines how often tasks are preempted by the scheduler. A higher tick rate means that there is less latency on the system, but it also means there is a greater load on the CPU. Though it was strictly not necessary, we chose to apply this to our

router as some rate limiters would benefit from having this enabled [50, *tc-htb(8)*, *tc-hfsc(8)*]. *dynticks* is a feature which starts the periodic tick rate only when there are tasks to preempt [49, *timers/highres.html#dynamic-ticks*]. This reduces load on the CPU when there are no tasks to preempt, however, in our case we wanted the CPU to be working at a constant rate to reduce latency on packets and thus it was turned off.

5.4.4 Defining a Stable Test Environment

I spent a considerable amount of time studying the configurations done by Wallenburg [74] on his test bed. He had done some good research on making the test bed run as smooth as possible, which we could easily apply to our test bed. I applied the following to our test bed, which in my opinion were crucial configurations for making a stable, reproducible, test environment.

Pause Frames were defined in the standard for Ethernet [41] as a link layer flow control. It may potentially introduce spikes in the stream of traffic as the sender may hold back packets if it receives a pause frame. Pause frames are on a general basis turned off by default, however as our NICs provide support for pause frames and link layer flow control [17, 18] we explicitly turned them off when we ran our experiments.

Interrupt Coalescing is a technique of holding back events that would normally trigger hardware interrupts. This could potentially introduce spikes in the stream of packets, similar to that of pause frames [55]. Wallenburg advised this to be turned off while running the experiments. On our NICs this was toggled off by default, meaning that we did not have to take this into account when configuring our kernels.

Segment Offloading is a technique for breaking larger chunks of data into smaller chunks for easier processing. Offloading can be done on hardware or in software and can be applied to many parts of the network stack. In our case we are mostly interested in TCP Segmentation Offload (TSO) and Generic Segmentation Offload (GSO). TSO is usually done on hardware and allows the device to segment a frame into multiple smaller frames that fit the maximum Maximum Transmission Unit (MTU) of the link [49, *networking/segmentation-offloads.html*]. GSO is done in software and is meant for cases where the hardware cannot perform the offloads. In GSO the socket buffers [73, *include/linux/skbuff.h*] are broken into smaller socket buffers that are easier to process throughout the network stack.

Segmentation offloading is meant to relieve the CPU off stress, by passing the work on to the NIC using TSO and/or increase the average throughput using GSO. However it may also alter the timing of the packets that are

captured which can cause unpredictable behaviour. It is therefore kept off during experimentation to further improve the repeatability and stability of the results.

TCP Metrics are entries in the kernel that store TCP specific information about a destination. This information is cached globally in the kernel and can be used by future TCP connections to set their initial conditions [50, *ip-tcp_metrics(8)*]. This may introduce problems when we want to reproduce experiments, and should therefore be turned of while experiments are run. In addition to turning TCP metrics off during testing, the cache should be flushed as metrics may have been saved in between experiments.

Rate Limiters in Linux are used to limit the average sending rate in Mbit/s on a specific link. A major part of Wallenburg's thesis revolved around determining the best rate limiter for such a test bed [74]. Through a series of experiments he compared Hierarchy Token Bucket (HTB) [50, *tc-htb(8)*] to Hierarchical Fair Service Curve (HFSC) [50, *tc-hfsc(7)*], concluding that the best rate limiter for this case would be HFSC. For this reason our choice of rate limiter was HFSC.

System Clock Synchronization on Wallenburg's test bed was done using Network Time Protocol (NTP) [54]. This was due to the NICs lacking of support for hardware time stamping. Our NICs, however, did support hardware time stamping. Therefore, I set up our edge nodes to synchronize their system clocks to the router using Precision Time Protocol (PTP) [40]. The main reason to use PTP over NTP is the accuracy. PTP allows for micro- to nanosecond accuracy on the system clocks which can be very useful when packets are captured on multiple nodes. The main drawback of PTP that I witnessed was its somewhat complicated setup. I ended up setting PTP as a service on the five machines, which we then turned off during experiments to not generate additional traffic on our links that could potentially give us some erroneous results.

In addition to Wallenburg's findings I further expanded the kernel configurations with the following.

Processor Frequency Scaling To make sure that all nodes were running at well performing pace, we set the processor scaling governor for each core on each node to *performance*. This was only toggled on while running experiments, as we did not want to exhaust our nodes.

IP Multicast Membership Messages are General Queries sent periodically by multicast-enabled hosts. These queries are use to build and refresh

multicast membership groups within the multicast system [11]. Though these queries do not consume a lot of bandwidth, we permanently turned multicast off on all interfaces used for testing on the router node and edge nodes using [50, *ip(8)*].

5.4.5 Network Emulation

In my experiments I wanted to emulate a highly congested link which handled all traffic between two networks, referred to as a bottleneck link. The two networks were referred to as the EAST and WEST networks. We were in a very similar situation as Wallenburg [74] (only five machines at our disposal) and could therefore take inspiration from their setup and apply it to ours. The bandwidth was set to 100 Mbit/s and the delay to 30 milliseconds. This was based on the values used by Wallenburg which they argued would represent a *good* Internet connection between Oslo and somewhere in the mainland of Europe [74]. Each router was equipped with a FIFO (drop tail) queue set to the size of the BDP in packets [3]. This would allow a single TCP stream to fully utilize the capacity of the link, but generate loss when congested.

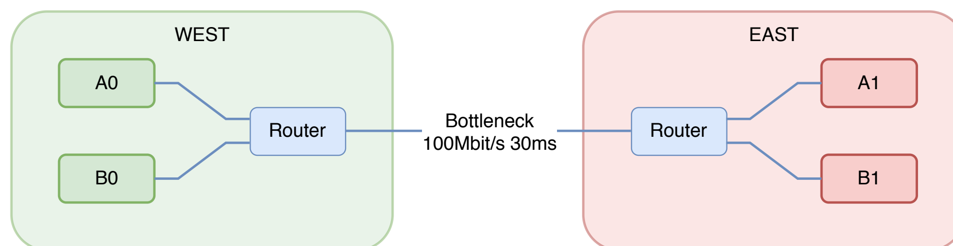


Figure 5.6: A more advanced figure of the dumbbell topology, showing how the WEST and EAST networks were set up. The two networks are connected by a 100 Mbps, 30 ms propagation delay link.

Initially we used CORE as our way to emulate the specified topology. This was great for creating network topologies as it provided a user interface, as shown in figure 5.7, and was quite easy to use. However, even though it was easy to create the topologies, I quickly found out that applying rate limiters, queues, and delay to the nodes still required me to add custom scripts to the virtual machines created by CORE. This grew tiresome in the long run because the virtual machines would not report back if there were errors in the scripts. I therefore resulted in taking a step back, and re-implementing the network emulation using tools provided by Linux for virtualization and traffic shaping.

[3]BDP of 250 packets. $100Mbit/s * 30ms = 250packets$.

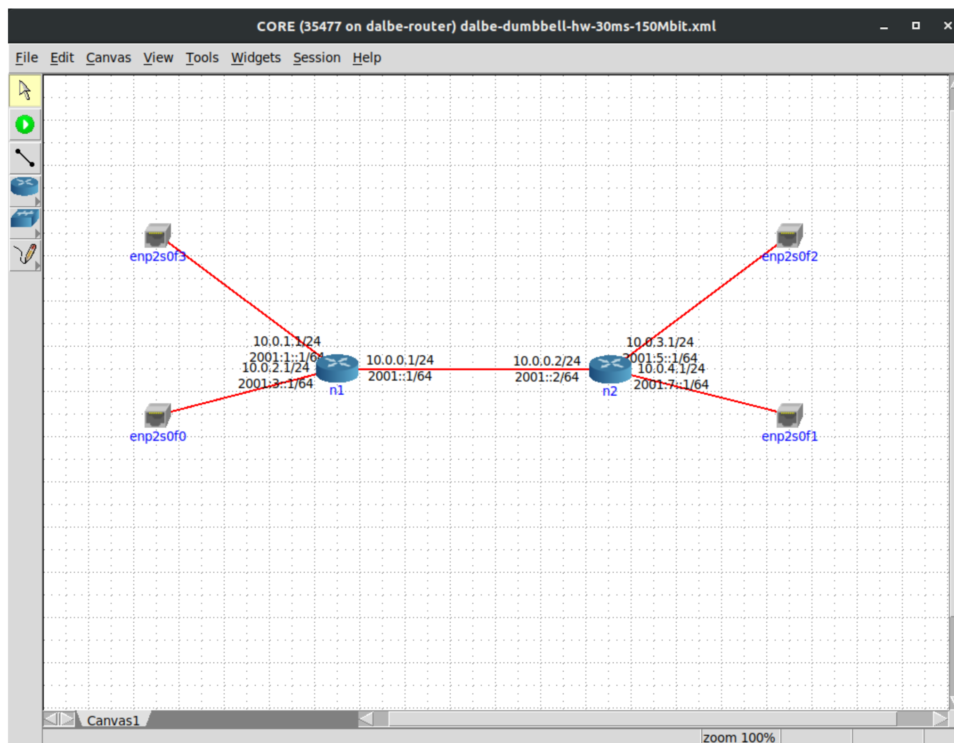


Figure 5.7: A screenshot of CORE GUI in action. This displays how we could easily, visually set up our network typologies.

Applying Network Emulation Using Linux Tools

Figure 5.8 shows how the two networks were connected by introducing two virtual interfaces on the router node. The two virtual interfaces were set up as IFBs to which I could apply traffic shaping and this way emulate my topology and it's network specifics.

All ingress traffic on the router node from the WEST network was filtered through, what we called, the EASTBOUND IFB. The EASTBOUND IFB applied a rate limit of 100 Mbit/s using HFSC [50, *tc-hfsc(8)*] and a *pfifo* droptail queue [50, *tc-pfifo(8)*] of 250 packets using Traffic Control [50, *tc(8)*] queuing disciplines. The egress traffic from the EASTBOUND IFB, which at this point has passed through the described queuing disciplines is then forwarded to the correct outgoing physical interface based on the systems forwarding tables. At the point of leaving the router node, 15 milliseconds of delay is applied to the traffic using Netem [50, *tc-netem(8)*]. The same is done for WESTBOUND traffic from the EAST network. Together this generates the desired emulated bottleneck with a base propagation delay of 30 milliseconds.

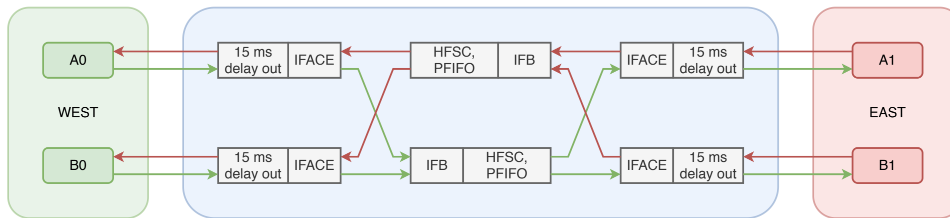


Figure 5.8: An advanced model of how the dumbbell topology was set up on the hardware based test bed. All ingress traffic, may it be EAST or WEST bound, passes through an IFB. The IFB is responsible for applying the HFSC and PFIFO qdiscs, before passing the traffic on to the outgoing NIC. The outgoing NIC applies a 15 ms delay.

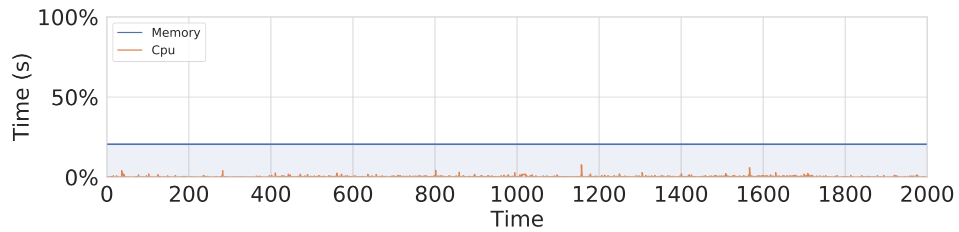
5.4.6 Verifying the Performance of the Test Environment

Before I could conduct any experiments on our hardware based test bed, I wanted to verify that it was in fact performing without any problems. As mentioned in previous sections, our main concern was that the edge nodes would not be powerful enough for our experiments, and thus giving us non-reproducible results. For this reason I wrote some simple software in *C++* that monitored and logged the average load and memory usage at random exponential intervals. The randomness was added to the sampling to prevent the monitor from accidentally follow a similar pattern to the scheduler in the OS, which could lead to the samples being non-representative of the actual load in the system. I ran this software in the background while conducting experiments on all the nodes except the router node, as I believed it to be more than powerful enough.

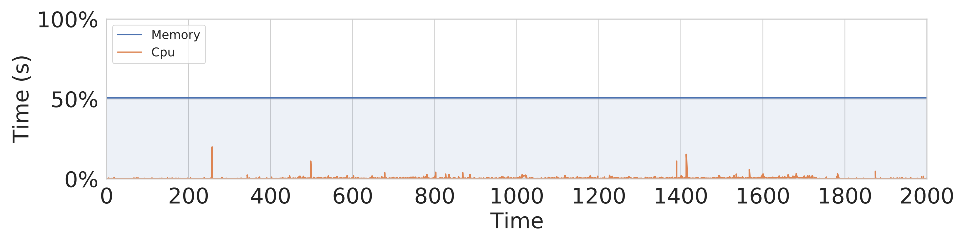
Figure 5.9 shows the graphs I could produce from the output of the monitor software. This could be checked after running a test or an experiment to verify how the nodes had performed under stress. Essentially, what I wanted to avoid was 100% load on the CPU and/or RAM.

5.5 Experiment Orchestration

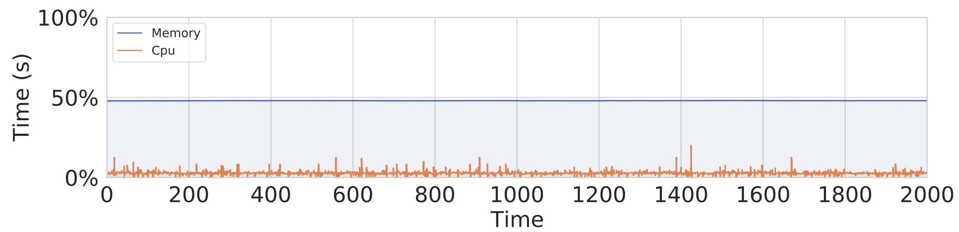
Our software suite for orchestrating the experiments was written by both Mattis and I in Python [26, *version 3.8*] and used Fabric [24] for running bash commands remotely over ssh. The advantage of this was that we could orchestrate the experiments from our local desktop, or as we did when we ran our experiments, we orchestrated them from the router node. The reason for orchestrating from the router node was to avoid delay between our local desktop and the nodes, which at times could be quite high and unpredictable. All experiments were written as JSON [10] configuration files which the software could translate into commands that were then run on the respective machines.



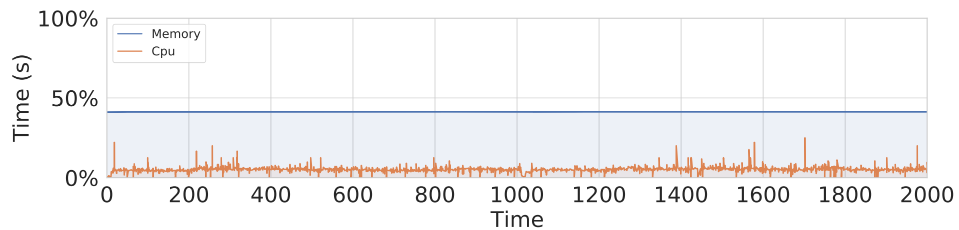
(a) Node A0



(b) Node A1



(c) Node B0



(d) Node B1

Figure 5.9: The figures show the average load and the memory usage on each edge node, taken from one for the network performance experiments in chapter 6. This was use to monitor the edge node during experimentation and testing.

5.5.1 Experiment Execution

Each experiment went through a well defined set of steps which can be described as follows.

Step 1 - Parse the Configuration File The software reads the configuration file, and translates it into two sets of commands for each node that is specified. One set contains untimed commands that should be run before the experiment timer starts, and the other is a set of timed commands that should be run during the experiment at a specified time, in seconds, relative to the start time. Untimed commands are usually used for e.g. starting listening servers or starting network traces.

Step 2 - Connect to all Nodes If the configuration file has successfully been parsed, a connection i established between all the nodes, which is kept alive during the experiment.

Step 3 - Set the Network Configurations for Each Node Before any command is executed on the nodes, all configurations are set for the interfaces that are used that are not permanently set. This is to provide a stable, predictable network for the experiment, described in previous sections.

Step 4 - Execute All Commands When the experimental setup is ready, all the commands can be run in their given order. As for the untimed commands they are just executed in the order in which they were defined in the configurations file.

Step 5 - Re-Set the Network Configurations for Each Node When the experiment has ended, hopefully in a graceful manner, all configurations that were set in step 2 are set back to their default values.

Step 6 - Disconnect From All Nodes This last step is quite self explanatory. When the experiment is over and everything has been reset to their defaults the established connection to all the nodes is torn down.

5.5.2 Data Collection

The data generated by the nodes consisted of PCAP trace files generated by *tcpdump* [31], CSV files generated by our memory- and CPU-load monitoring software, and the system logs which contained debug information about the DA-LBE meta congestion module. As mentioned in previous sections, our edge nodes were not very powerful and had very limited storage capacity. We therefore moved the resource heavy task of capturing packet traces to

the router node, monitoring the ingress traffic on the physical interfaces connected to edge node A0 and B0. The logs from monitoring memory- and CPU-load as well as the debug logs for the DA-LBE meta congestion module naturally had to be generated at each edge node and the DA-LBE-node respectfully. After running one or more experiments we used *scp* [50, *scp(1)*] to collect all the data.

5.5.3 Data Processing

To process the PCAP files we used PcapPlussPluss [64] which is a library for parsing PCAP files, simlar to libtrace [32], but written in C++. Using PcapPlussPluss I wrote a program to generate CSV files with information about the connections and their throughput. I used Python to develop functions that created plots from our collected data. These functions could be imported directly to Jupyter Notebook [48] for viewing.

5.6 Summary

In this chapter I presented how the test environment(s) were built, configured and verified. I explained how a virtual test environment was used for testing and debugging, and how it was not sufficient for running proper experiments, as the load on the host machine became a degrading factor. A detailed description of how I, together with a colleague, built and configured a hardware based test environment was also given. Describing what hardware components were used, the operating systems and their configurations, and how network emulation was performed, showed how we were able to build a stable test environment suitable for experiments on the DA-LBE meta congestion controller. Lastly, I described how we wrote a software suite which allowed us to configure and run experiments on the test environment.

Chapter 6

Network Performance Experiments

In this chapter I present a set of experiments that were used to evaluate the network performance of the DA-LBE meta congestion controller, for both MBC and PID, using Vegas as the underlying congestion controller. In addition to this I investigate the effect of using the fixed point operations as a substitute for floating point operation in the Linux kernel.

6.1 Requirements

The third research question (RQ3) defined for this thesis is;

> *How should a DA-LBE transport service be tested and evaluated to verify its correctness and performance?*

From this question I take into account the requirements defined by Hayes *et al.* [35] which state that a DA-LBE traffic source should;

- be no more aggressive than BE traffic,
- react appropriately to network congestion,
- take advantage of available network capacity when there is no congestion, and
- attempt to finish transmitting its data by the deadline.

These requirements are used as the basis for evaluating the network performance.

6.2 Network Efficiency

The first experiment aims to evaluate the overall network efficiency of the DA-LBE meta congestion controller. Hayes *et al.* [35] defined a simple experiment to test just this, which Wallenburg [74] later implemented on his hardware based test environment. I replicated this experiment to see how well Dalbe performed using both MBC and PID for with Vegas as the underlying congestion controller.

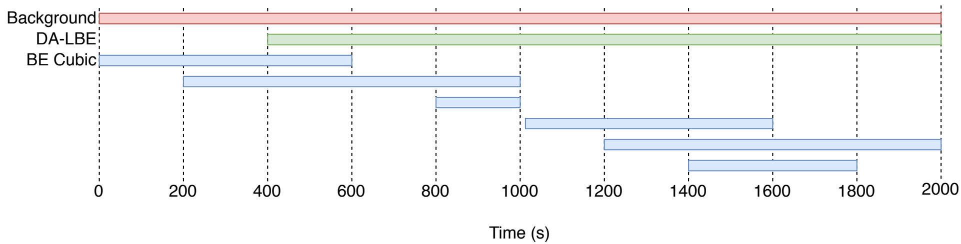


Figure 6.1: Timeline of the network efficiency experiment showing when the different flows start and stop.

6.2.1 Setup

The test scenario is presented in table 6.1, and visually in figure 6.1. Each BE Cubic flow attempts to consume its fair share of the capacity. The DA-LBE flow will attempt to send 1625MB of data with a deadline of 1300 seconds. To achieve this the DA-LBE flow will require an average send rate of 10 Mbps, though the nature of the *soft deadline* does not require it to fulfill this requirement perfectly. To avoid perfect synchronization between TCP flows, 10 Mbps of randomly generated background traffic is added using D-ITG [47]. Both the DA-LBE and BE flows were sending packet sizes which matched the 1500 MTU of the network.

The DA-LBE flow was set up with update intervals of 10 seconds for both w and ϕ, and a calculation threshold of 10 loss events for a new ϕ value to be computed. The choice of the calculation threshold for ϕ was to use similar values to the values used by Wallenburg in his similar experiment, as well as observations made during debugging. Initially, $w = w_{\min} = 0.05$, $\phi = 3$ and $\mu = w\phi$. This allows the DA-LBE flow to start quite passively before gradually increasing its send rate.

For the PID based DA-LBE flow, I used the gain values; $K_p = 0.5$, $K_i = 0.03$, and $K_d = 0.1$. These are the same values used by Hayes *et al.* when testing PID based control. I relied on these values as tuning them is quite difficult, and out of the scope for this thesis.

Both experiments used Vegas $\alpha = \beta = 16$. Having the α and β values equal allows Vegas to react faster to queuing delay, as there is no middle-

zone where the CWND can fluctuate.

Connection	Start	Stop	Duration
DA-LBE	400 s	-	-
BE Cubic	0 s	600 s	600 s
	200 s	1000 s	800 s
	800 s	1000 s	200 s
	1010 s	1600 s	590 s
	1200 s	2000 s	800 s
	1400 s	1800 s	400 s

Table 6.1: Start and end times for each flow for the network efficiency experiment. The deadline for the DA-LBE flow is set to 1300 s, which means that it should finish close to t = 1700 s.

6.2.2 Expectations

I expect that the DA-LBE flow should be able to utilize openings of the available capacity to a certain degree, such as at t = 1000 seconds where there are no competing BE flows for 10 seconds. As the underlying congestion controller mainly reacts to queuing delay, with $\alpha = \beta$, I would expect the reaction to be quite quick, and thus it should be able to utilize the gap quite well. Similarly, it should also be able to react equally fast to the increase in queuing delay when the BE flow enters the network again at t = 1010 seconds, thus the DA-LBE flow should quickly back off, allowing the competing traffic to receive most of the capacity as the deadline is still quite far away at this moment.

Another effect that I would expect, due to the timely convergence of the underlying congestion controller, is that the DA-LBE flow may interpret the exit of BE flows as available capacity. Most notably this would be expected at points where one or more BE flows leave the network, leaving the remaining BE flows in a state where they will try to probe for more capacity. This may open a small gap where the competing flows have not yet reached the new threshold for their sending rates, which may decrease the queuing delay calculated by DA-LBE and the underlying congestion controller, thus increasing its aggressiveness for a short period of time.

Specific Expectations for MBC

Towards the end of the transmission, at t = 1400 seconds, there is a period of 200 seconds where three BE flows are competing for capacity at once. At this point I would expect the DA-LBE flow to have begun its increase in aggressiveness if it has not been able to keep its average send rate close enough to the target rate. As the competing traffic increases drastically

at this point the DA-LBE flow may react to this increase of queuing delay as if it is the cause of it, which may lead to it becoming too passive, thus overshooting the *soft deadline*.

Specific Expectations for PID

On the other hand, for PID based DA-LBE I expect a different result, as the weights are not directly influenced by the queuing delay, but rather some samples from the past, the current moment , and a prediction of the future, I would not expect it to alter its behaviour drastically, but rather keep a stable flow of traffic. I would also expect the PID based flow to keep a more precise average rate for the entire transmission which should allow it to finish closer to the deadline.

6.2.3 Results for Model Based Control for Vegas

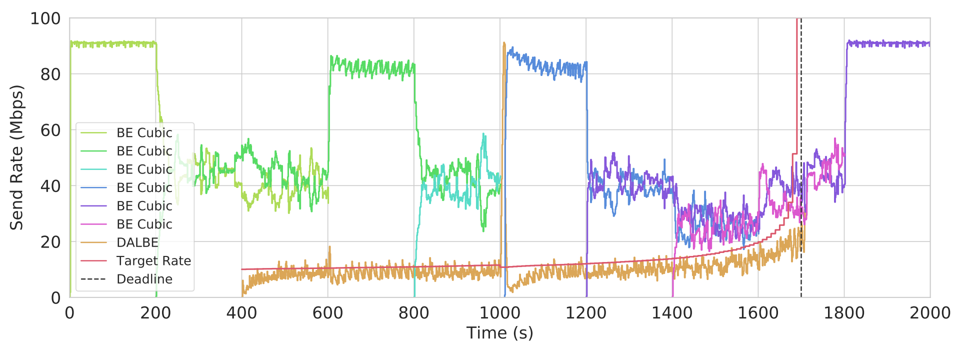


Figure 6.2: One second averages of the throughput for each flow during the network efficiency experiment. The DA-LBE flow uses Vegas as the underlying congestion controller with MBC as the weight adjustment policy.

The results for the MBC based DA-LBE flow is quite close to what I expected. Figure 6.2 illustrates a graph of the throughput of the competing flows, averaged over one second intervals. It is apparent that the DA-LBE flow starts quite passively, as it tries to reach the target deadline. It uses about 100 seconds to reach a sending rate, right below the target rate. Having set $\alpha = \beta$, allows the congestion controller to keep this rate very stable, as there is no gap between the two thresholds. By looking at 6.3b this becomes quite apparent, as it can be seen that the average rate stays very stable during the entire connection with few fluctuations.

In 6.2, at t = 600 seconds, there is a small spike in the sending rate. I would assume this to be due to the first BE flow leaving the network, and the DA-LBE flow confusing this for available capacity. In reality the second BE flow, which is still in the network, has not yet been able to react to the

change, but once it realizes it can claim more capacity it does so, resulting in the DA-LBE flow having to quickly back off. This back off sets it back a little bit, before re-gaining a stable sending rate.

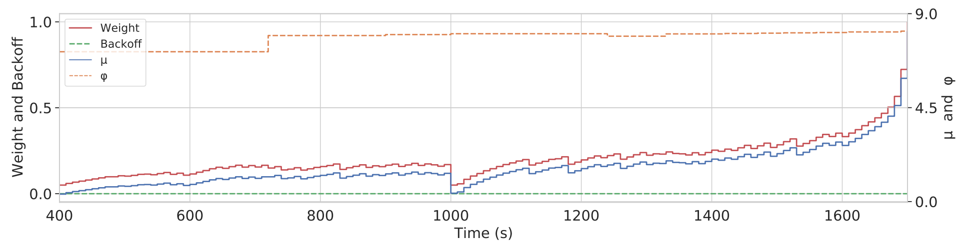
At t = 1000 seconds there is a 10 seconds gap where there is no competing BE traffic, which the DA-LBE flow is able to utilize quite well, producing a large spike in the throughput which can be seen in figure 6.2, as well as the large increase of the CWND in 6.3e. The large drop in RTT seen in figure 6.3d, together with the underlying congestion controllers timely reaction to the decrease in queuing delay allows the DA-LBE flow to quickly utilize this window. However, once the 10 seconds period with no competing traffic is over, the DA-LBE flow quickly reacts to the increase in queuing delay, as the BE flow enters the network and begins to flood the buffers on the routers. At this point, the weights are drastically reduced which can be seen in figure 6.3a. From this setback, the DA-LBE flow uses about 100 seconds to regain a stable rate, as the weights are increased by a limiting factor, presented in equation 2.4 in chapter 2.

Towards the end, at about t = 1400 seconds, the target rate increases rapidly, which quickly turns into an exponential growth. As the DA-LBE flow has to this point been just below the target rate during the entire session, with the exception of the spike at t = 1000, followed by the penalty for regaining is stable sending rate which followed this spike, the DA-LBE flow starts to increase its target rate in an attempt to reach the deadline. This increase becomes very rapid after t = 1600 seconds, as the deadline closes in. What is evident is that the DA-LBE flow is still able to follow the target rate quite well during this period, even though there is an increase in competing traffic. This allows the DA-LBE flow to finish only a couple of seconds after the deadline, which is considered acceptable as the deadline is *soft*.

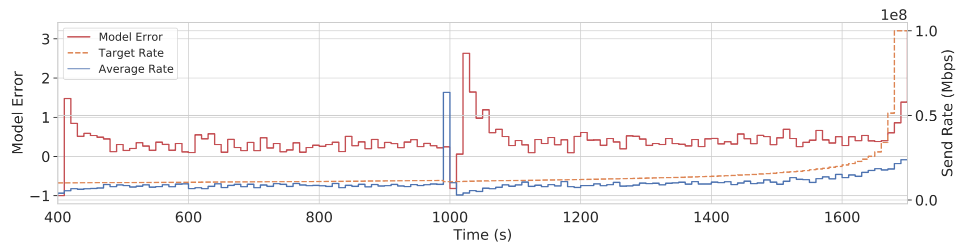
Results for PID Based Control for Vegas

Figure 6.4 illustrates that the PID based flow starts off quite similar to the MBC flow. Starting passively, as the initial values are quite low, yet more rapid than than of MBC, which I would presume is due to the queuing delay not being part of the equation. Figure 6.5b shows that the error is quite high in the beginning, which in turn has an effect on the growth factor of w, resulting in a more aggressive start.

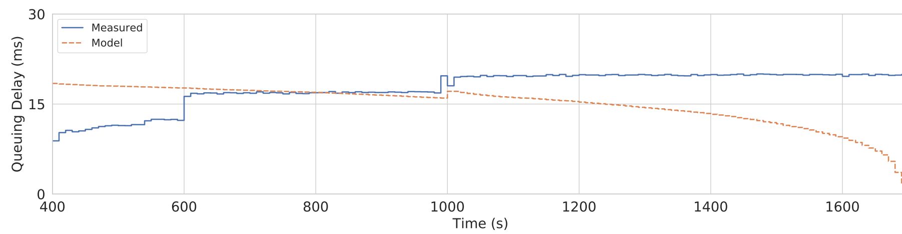
At around t = 550 seconds there is a significant spike in the send rate of the DA-LBE flow. By looking at figure 6.5a it is apparent that this is caused by a large increase in ϕ. The requirement for ϕ to be recalculated is that there is enough loss events for a proper calculation. This suggests that there is a low count in loss events, followed by a large burst of loss events. However, this rapid increase of ϕ also has a negative effect on the LBEness of the DA-LBE flow with respect to the amount of data and the deadline.



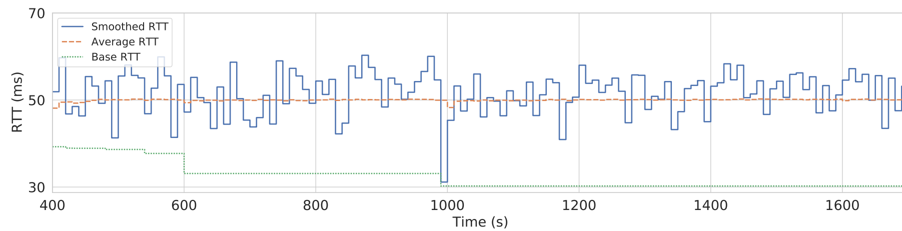
(a)



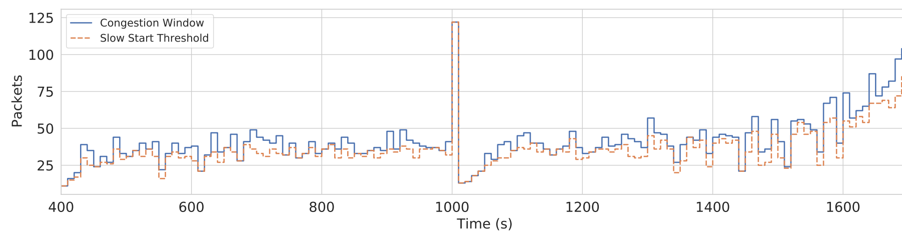
(b)



(c)



(d)



(e)

Figure 6.3: Debug graphs produced for the network efficiency experiment for a Vegas based DA-LBE flow using MBC as the weight policy. 10 seconds increments.

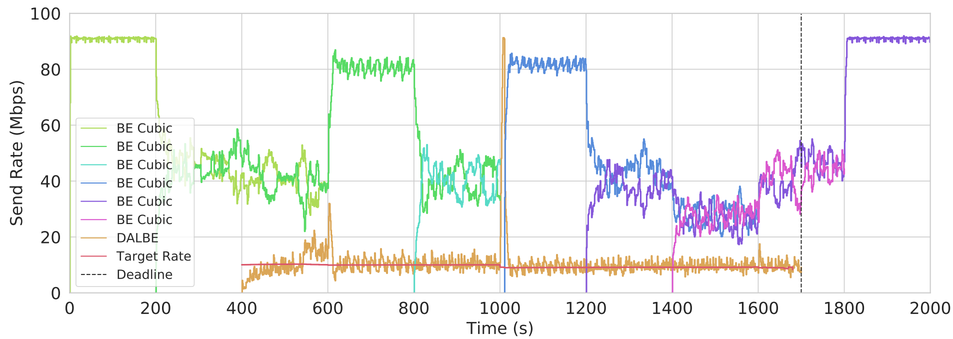


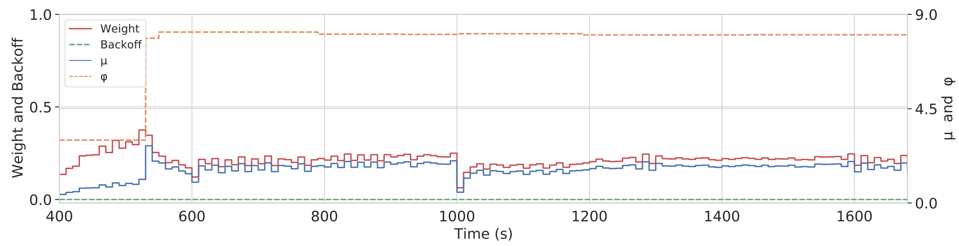
Figure 6.4: One second averages of the throughput for each flow during the network efficiency experiment. The DA-LBE flow uses Vegas as the underlying congestion controller with PID as the weight adjustment policy.

Figure 6.5a shows that w is quite high at this point in time. The increase of ϕ together with the high w value, suggest that the queuing delay is actually being deflated at this point, which explains the increased aggressiveness of the DA-LBE flow.

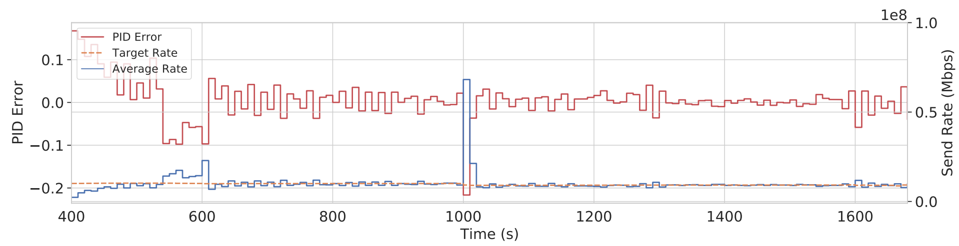
At t = 600 seconds there is a similar spike as seen for the MBC experiment. It is more noticeable when looking at figure 6.5c. After stabilizing from the rapid increase of ϕ, it can be seen that there is a drop in RTT at t = 600 seconds. This suggests that the DA-LBE flow reacts to this change in RTT as if there is more available bandwidth, however, in reality the remaining BE flow is in the process of probing for more capacity, which the DA-LBE flow quickly realizes and backs off.

At t = 1000 seconds, when there are no competing BE flows for 10 seconds, the DA-LBE flow quickly reacts to the available capacity. The main difference from the MBC experiment in this scenario is that the send rate does not decrease to almost zero, and thus it is able to regain stability a bit faster in this situation. Figure 6.5a suggest that this may be due to the w and μ values are not decreased to their minimum values, and thus stabilizing the values does not take as long as well as the aggressiveness staying a little bit higher.

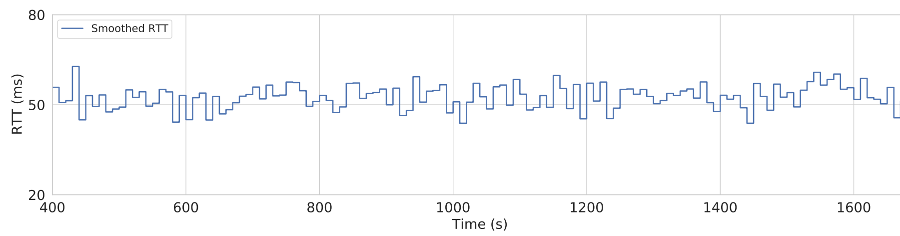
Interestingly, at t = 1400 seconds there is little to no change in the target rate of the DA-LBE flow. By looking at figure 6.5b, it is clear that the average rate stays very close to the target rate for the entire connection, with the exception to the scenarios described above. This allows the DA-LBE flow to finish just in time, yet still not way ahead of the deadline, which would suggest that it was able to maintain a decent amount of LBEness during the connection.



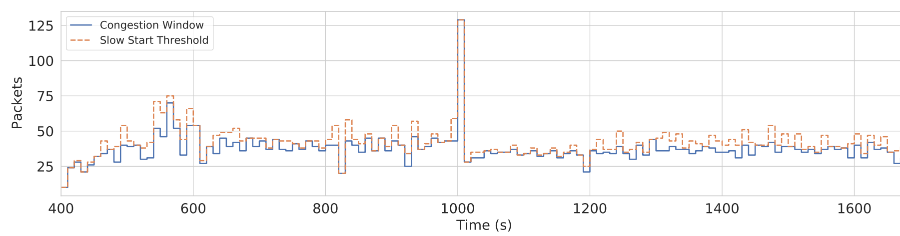
(a)



(b)



(c)



(d)

Figure 6.5: Debug graphs produced for the network efficiency experiment for a Vegas based DA-LBE flow using PID as the weight policy. 10 seconds increments.

6.2.4 Fixed Point Precision

Finally, I investigate the impact the use of *fixed point* numbers has on the quality of the DA-LBE calculations. This is important to investigate, as the meta congestion controller relies solely on a Q16.16 representation of *fixed point* values. This representation can be considered as a *Jack-of-all-trades*, providing a relatively wide range of integer values within (2^{16}), and a relatively good precision down to (2^{-16}). However, when compared to the precision achieved by using *floating point* types, this representation may not be good enough as the accumulated error in precision may become quite large.

Continuing with the debug information produced for the two weight policies, MBC and PID, in the network efficiency experiment, I compare the actual weight w calculated with *fixed point* operations, to an the expected weight calculated using *floating point* operations. This comparison is achieved by calculating the *relative error* between the two values;

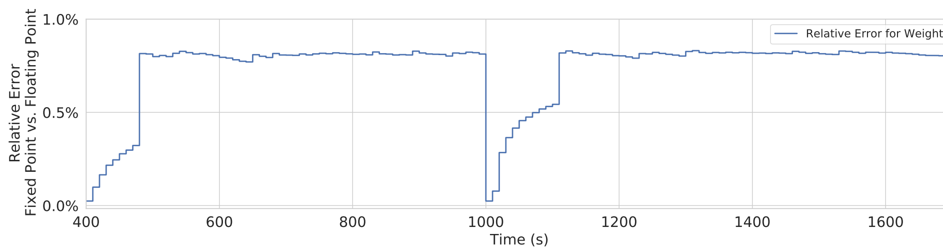
$$\text{error} = \frac{|w_{\text{fixed}} - w_{\text{float}}|}{|w_{\text{float}}|}$$

The choice of investigating w is based on its importance in the DA-LBE calculations, and should be a good representative for the overall error in precision.

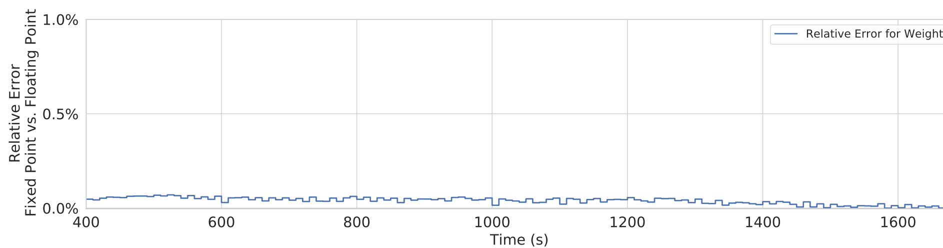
Figures 6.6a and 6.6b illustrates the relative error for w in percentage, computed during the network efficiency experiment, for both MBC and PID respectively From these figures it is quite clear the error introduced when using MBC is noticeable larger than the error introduced when using PID. Also noticeable, especially when using MBC, is the drop in error for the interval at t = 1010, which is the point were the DA-LBE flow has to back off as a new BE flow enters the network.

These observations suggest two things. First and foremost, when more variables are introduced to the calculation, which is the case for MBC, as relies on both queuing delay and the relationship between target rate and sending rate, the error seems to be greater. Secondly, when the values used in the calculations have not stabilized or they change drastically, the fluctuation in the the error seems to be greater. These assumptions are strengthened by the large drop in the error for MBC from t = 1000 to t = 1010, as both the average sending rate and queuing delay is changed drastically around this moment, which causes the weight to be decreased to its minimum value ($w_{\text{min}} = 0.05$), followed by the weight being gradually stabilized over an approximately 100 second period. This drastic drop is not as apparent for when using PID as it has less variables as part of the equation that are changed during this time period, and it has been seen that it is able to stabilize its weigh much quicker than MBC.

This further suggests that having more variables that introduce small



(a)



(b)

Figure 6.6: The relative error between the actual value of w, computed using *fixed point* operations, and the expected value of w, computed using *floating point* operations. 10 second increments.

errors will gradually build up the overall error of the calculations, and as this error increases, the quality of the calculation decreases. However, in this case, even though there is some error in the calculations, it is below 1% at all times, which is relatively low. From this it would be quite safe to assume, also considering the promising results from the network efficiency experiment, that it is not enough to have a major effect on the DA-LBE calculations in this case. However, if this error reaches values much larger than 1%, it could be beneficial to map out the variables that have an effect on the overall error, and limit the error that they are causing by *e.g.* increasing the fractional part.

6.3 Fairness and Completion Times

The second experiment revolves around analysing the fairness and completion times of the DA-LBE meta congestion controller. Both Hayes *et al.* [35] and Wallenburg [74] took this into account when evaluating the network performance of the DA-LBE framework. Wallenburg defined an experiment for testing these two factors which I replicated in my test environment.

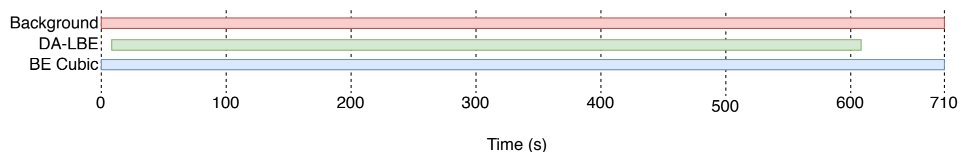


Figure 6.7: Timeline of the fairness and completion times experiment showing when the different flows start and stop.

Connection	Start	Stop	Duration
DA-LBE	10 s	-	-
BE Cubic	0 s	710 s	710 s

Table 6.2: Start and end times for each flow for the fairness and completion times experiment.The deadline for the DA-LBE flow is set to 600 s, which means that it should finish close to t = 610 s.

6.3.1 Setup

The test scenario used for this experiment, described in table 6.2 and visualized in figure 6.7, revolves around running one DA-LBE flow concurrent with one BE Cubic flow. The BE flow is given a 10 seconds head start, which should allow it to reach its maximum capacity before the DA-LBE flow enters the network. The BE flow runs for 710 seconds, giving the DA-LBE flow 700 seconds to finish it transfer. This experiment takes about ten minutes, and was run for fifty iterations, to give a reasonable quantity of samples from which I could compare the fairness and completion times.

The amount of data transferred by the DA-LBE flow is the same as used for the first experiment; 1625 MB. However, the deadline is set to 600 seconds, which is much closer than previously used. This will result in the DA-LBE flow to compete more aggressively for the available capacity to maintain its target rate and possibly finish within the given deadline. To be able to finish its transfer in time, the DA-LBE flow will have to maintain an average rate of 21.67 Mbps.

In addition to the competing BE flow, I generated 10 Mbps of background traffic to avoid synchronization between the two flows. I also used identical configuration parameters, for both MBC and PID, in this experiment as I used for the network efficiency experiments. For both the MBC and PID based flows.

6.3.2 Fairness

First I consider the achieved fairness from the fifty samples. I used the *Jain's fairness index* to compute the fairness from three parts of the transmission. Table 6.3 shows the parts of the samples I used to compute the fairness

Fairness Parts		
Part	Start	Stop
Mid	100 s	355 s
End	356 s	610 s
Avg	100 s	610 s

Table 6.3: Table of the parts used to create fairness indexes.

indexes. I chose these three parts in an attempt to get a more precise view of the fairness throughout the transmission, as relying on only an average of the entire transmission would possibly hide the fact that the DA-LBE flow is performing too aggressively or too passively at certain times.

The middle part was chosen so that I could look at the transfer in a state where I suspected it to not be increasing its target rate by too much for each interval, *i.e.* when the DA-LBE flow should be at its least aggressive. The end part was chosen so that I could look at transfer in a state I suspected the increase in target rate would be the greatest for each interval, *i.e.* when the DA-LBE flow should be going towards its most aggressive. I also included the average over the entire transfer to get an idea of the overall fairness achieved by the DA-LBE flows.

Note that I have excluded the first 100 seconds from each sample, as this is the part where the DA-LBE flow has not yet stabilized. In addition to this I have excluded the time after the deadline has been reached, as the DA-LBE flow should at this point be as aggressive as the competing BE flow. I have done so, as I believe this reduces bias from the results, and thus giving a more precise measurement.

Expectations

With the knowledge of the amount of competing traffic there is in the network, I can make an assumption about the expected fairness that the DA-LBE flows should achieve. Wallenburg showed from his local test bed that having a DA-LBE flow transferring 1625MB of data within the deadline of 600 seconds, allowed for 5125MB to be transferred in the same time by a BE flow, totalling 90% of the entire capacity, while the remaining 10% is consumed by the background traffic [74]. From this I can calculate the expected fairness index by following the Jain's formula:

$$\frac{(1625+5125)^2}{2\times(1625^2+5125^2)} \approx 0.788$$

If the fairness values calculated from my samples are below this value, it would suggest that the DA-LBE flow is not able to finish in time *i.e.* it is performing too passively. On the other hand, if the fairness values from

our samples show are above the expected value it would suggest that the DA-LBE flow is acting more aggressive and thus finishing before the given deadline.

However, if the fairness values from my sampling periods, seen in table 6.3, are too close to 1, or even equal to 1, this would suggest that the DA-LBE flows are not maintaining a LBE like behaviour.

In my previous experiment I have seen that when MBC is used as the weight policy, the average sending rate of the DA-LBE flow may be a little bit below the target sending rate, thus resulting in the flow overshooting the *soft deadline*. From this observation I would expect to see fairness values in the middle of the transmission to be close, but maybe a little bit below an expected fairness value, and the fairness values from the end of the transmission to be a little bit above the expected value as it should be rapidly increasing its aggressiveness to finish close to the deadline. The overall fairness of the MBC flow may be very close to the expected, however this depends heavily on the increase towards the end of the transmission.

However, when PID is used as the weight policy I have observed that the average sending rate stays very close to the target sending rate during the entire transmission, thus resulting in the transfer finish very close to the *soft deadline*. From this observation I would expect all fairness values from the middle, end and the average to be quite close the the expected value.

Results

Figure 6.8 and 6.9 are box and whiskers plots illustrating the fairness values from the 50 iterations of the 10 minute tests. The box extends from the first (Q1) to the third (Q3) quartile values, and the line in the middle represents the second (Q2) quartile, which in this case is the median value. The whiskers show the range of the data which is set to $1.5 \times (Q3 - Q1)$. Outliers, if any, are shown as dots beyond the range of the whiskers.

The three parts plotted represent the middle part (100 - 355 s), end part (356 - 610 s), and the average which combines the middle and the end part. The horizontal line represents the expected fairness value calculated in the previous section using *Jain's fairness index*.

The Results for Model Based Control for Vegas, seen in figure 6.8, show that the fairness of the middle section is a little below the expected value. This would suggest that the DA-LBE flow fulfills its goal of being LBE at a stage where the target rate has not yet begun to increase due to the deadline being quite far away.

The fairness of the end section is quite far above the overall expected fairness, which suggest that the DA-LBE flow is competing quite a bit for capacity in an attempt to complete within the deadline. However, it is still below complete fairness. This is important as it shows that the DA-LBE

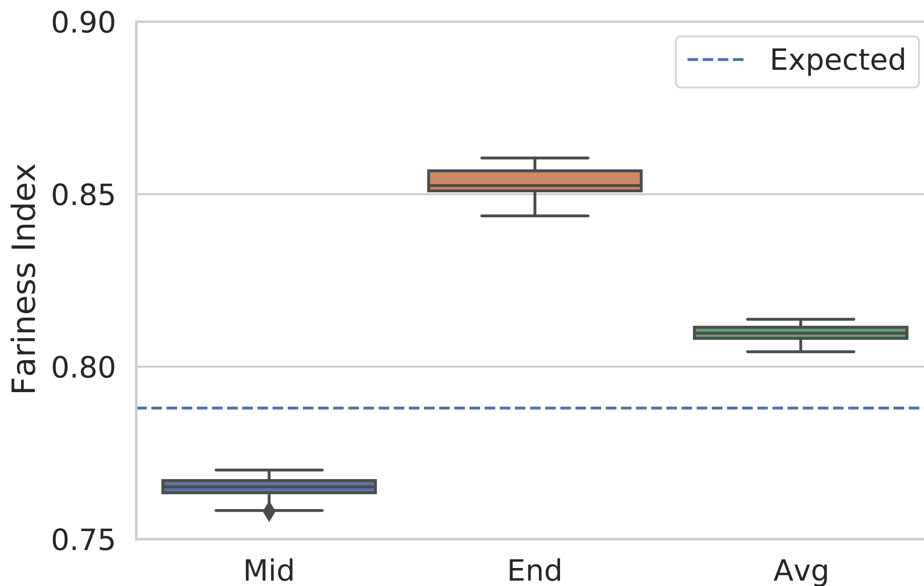


Figure 6.8: Fairness indexes for Vegas based DA-LBE flows using MBC as the weight policy. The box expends from the Q1 to Q3; the middle line represents the median (Q2); the whiskers extend to the $1.5 \times (Q3 - Q1)$; and outliers are represented as dots beyond the whiskers.

flows are not competing equally with the BE flows before the deadline has elapsed, suggesting that the they are still maintaining a correct LBE behaviour towards the end.

The overall average shows that the DA-LBE flows are staying quite close to the expected fairness value. Table 6.4 shows that on average the fairness index of the DA-LBE flows is 0.810, which is 0.022 off the expected fairness. I suspect this to be due to the rapid increase of aggressiveness during the end section. From my previous experiments (see figure 6.2) I have seen that when using MBC the average send rate follows the target send rate very closely, even when the target rate increases rapidly.

Although the overall average fairness is a little bit above the expected fairness, I can see that all the results are quite close to each other with very few outliers, as the standard deviation in table 6.4 is 0.002, which is quite low. This suggests that there is a recognizable, repeatable, stability achieved when using MBC as the weight updating policy, and that the flows achieve an expected LBE like behaviour.

The Results for PID Based Control for Vegas, seen in figure 6.9, show that the fairness of the middle section stays above the expected fairness, with some outliers stretching towards the expected line, which might suggests

Fairness Vegas MBC					
Experiment	Mean	Median	Stddev.	Max	Min
Mid	0.765	0.765	0.003	0.758	0.770
End	0.853	0.852	0.005	0.844	0.860
Avg	0.810	0.810	0.002	0.804	0.814

Table 6.4: Table of values related to the fairness indexes for Vegas based DA-LBE flows using MBC as the weight policy.

Fairness Vegas PID					
Experiment	Mean	Median	Stddev.	Min	Max
Mid	0.838	0.842	0.011	0.805	0.851
End	0.798	0.796	0.009	0.785	0.832
Avg	0.819	0.819	0.002	0.811	0.821

Table 6.5: Table of values related to the fairness indexes for Vegas based DA-LBE flows using PID as the weight policy.

that the aggressiveness, on an average, is quite high at the beginning of the connection. However, the values are still below one, which suggests that the DA-LBE flow is still less aggressive than its competing BE flow, thus still maintaining its LBE behaviour.

The end part, is much closer to the expected fairness. In my previous experiments using PID as the weight adjusting policy (see figure 6.4), I have seen that the average sending rate stays very close to the target rate. This would suggest that the DA-LBE flows are able to maintain an average sending rate very close to the target, thus also achieving a fairness close to what is expected.

The overall average fairness values in this experiment show that most of the DA-LBE flows are very close to achieving the same fairness values. The standard deviation in table 6.5 strengthens this claim, being only 0.002. This would suggest that there is also a repeating stability when using PID as the weight adjustment policy, and that the flows are in this case achieving an expected LBE like behaviour.

6.3.3 Completion Times

Continuing with the data produced for the fairness experiment, I wanted to see how close to the deadline the DA-LBE flows where when they had transmitted all their data. Each flow was given a deadline of 600 seconds, and each flow started 10 seconds after the competing BE flow, which suggests that at 610 seconds the DA-LBE flows should be finished or close to finished.

The completion times are computed as a difference between the time a flow finishes its transmission and the time the flow starts its transmission.

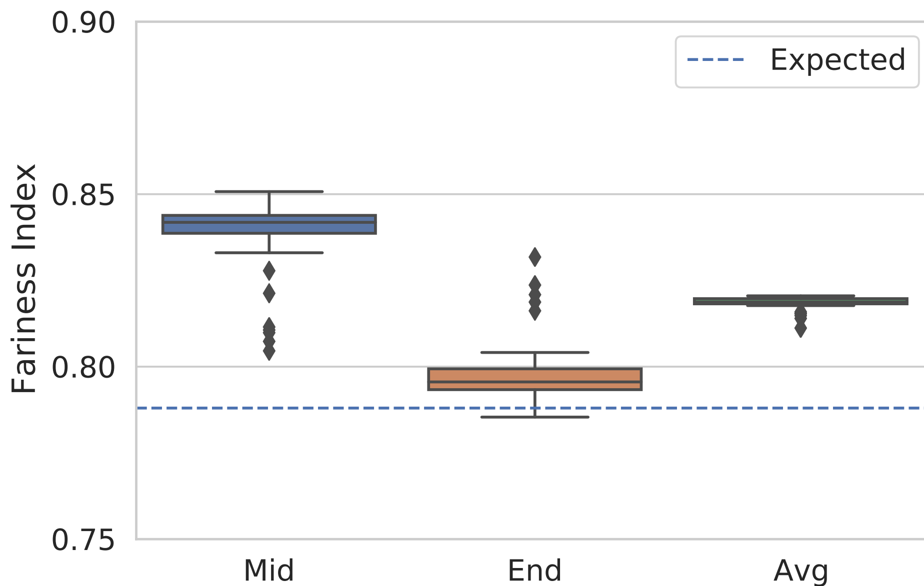


Figure 6.9: Fairness indexes for Vegas based DA-LBE flows using PID as the weight policy. The box expends from the Q1 to Q3; the middle line represents the median (Q2); the whiskers extend to the $1.5 \times (Q3 - Q1)$; and outliers are represented as dots beyond the whiskers.

I achieved these times by sampling the time each flow enters its initiation and release function (see figure 4.1), which are called at connection establishment and tear down. The blue line represents the deadline set for the flows, which was at 600 seconds.

Expectations

From the results of the fairness experiments I have seen that for, both MBC and PID, the overall average of the DA-LBE flows are very close to each other. However I have seen that MBC acts a little less aggressive than PID as its average send rate often lays below the target send rate, while the PID based flows tends to stay very close to the target rate. With this in mind I would expect the results to show that PID flows would finish closer to the deadline, while the MBC flows may overshoot by a couple of seconds. This expectation is further strengthened by the findings during the network efficiency experiments, where it was clear that the MBC flow missed the deadline by some seconds due to it being too passive during the connection, while the PID flow kept a very stable rate during the entire connection and was able to finish just in time.

Further, I must emphasize that the set deadline is considered as *soft*, meaning that the DA-LBE flows do not have to finish in perfect time to be

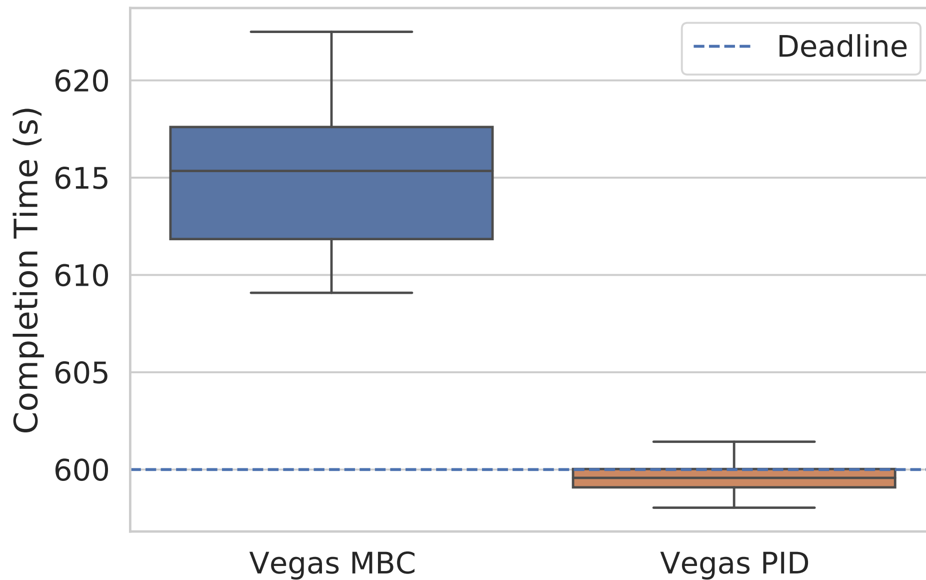


Figure 6.10: Completion times for Vegas based DA-LBE flows using MBC and PID as the weight policies. The box expends from the Q1 to Q3; the middle line represents the median (Q2); the whiskers extend to the $1.5 \times (Q3 - Q1)$; and outliers are represented as dots beyond the whiskers.

acceptable. However, concern should arise if the flows repeatedly overshoot by a considerable amount of time, or if the flows repeatedly undershoot by a considerable amount of time. These scenarios suggest a miss behaving DA-LBE flow which is either too passive or too aggressive, respectfully.

Results

Once again, I display the results in box and whiskers format, where the representation is the same as was used for the fairness experiments.

Figure 6.10 shows that the MBC flows there are not able to finish within the given deadline. This was close to what I expected, as I knew the MBC flow's average sending rate generally would stay a little below the target send rate followed by a rapid increase towards the end of the transmission. However, this increase does not seem to be rapid enough to finish within the deadline.

For the PID flows, the results are quite different. Very noticeable is the fact that the box and whiskers plot is very compact, which suggests that the PID flows are maintaining a good stability between sending rate and target rate. It also seems that on average the PID flows are able to finish very close the deadline, with only a few samples finishing a couple of seconds late.

Table 6.6, which shows the completion time relative to the deadline,

Relative Completion Times					
Experiment	Mean	Median	Stddev.	Max	Min
Vegas MBC	15.143	15.346	3.429	22.496	9.083
Vegas PID	−0.373	−0.427	0.774	1.431	−1.960

Table 6.6: Table of values related to the completion times for Vegas based DA-LBE flows using MBC and PID as the weight policies.

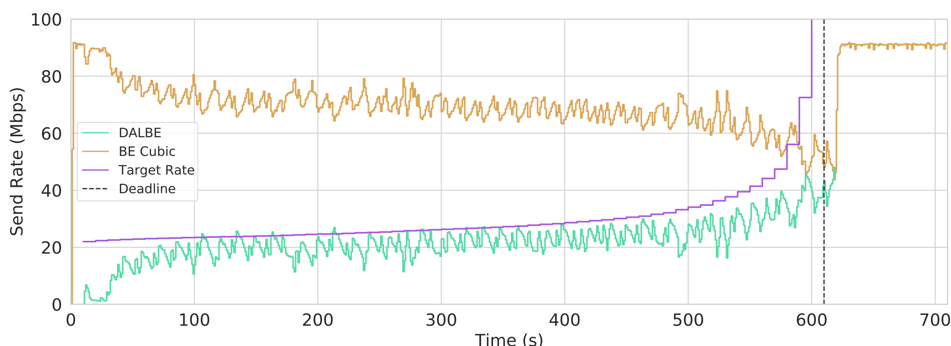


Figure 6.11: One second averages of the throughput of the earliest completing run for the fairness and completion times experiment for a Vegas based DA-LBE flow using MBC as the weight policy.

confirms my assumptions about the two flow types. I can see that the MBC flows on average overshoot by approximately 15 seconds, within a standard deviation from the mean of about 3.4 seconds. The PID flows, however, are on average able to finish before the deadline with approximately 0.4 seconds to spare. I can also see that the standard deviation is below one seconds which explains why the box and whiskers are so compact.

Earliest vs. Latest for Model Based Vegas

From table 6.6 it is evident that the earliest completion time by a MBC flow is approximately 9 seconds past the deadline. Figure 6.11 illustrates the sending rate of this flow, in terms of throughput, averaged over once second intervals. This run is able to achieve a very stable rate throughout the entire connection, and there is nothing very noticeable about this figure. However, quite noticeable here is the repeating behaviour of the MBC flows by achieving an average sending rate just below the target sending rate, and an increase towards the end of the connection in an attempt to finish in time.

From the same table it is apparent that the latest finishing flow is approximately 22.5 seconds past the deadline. Figure 6.12 shows that the latest contestant has a very similar behaviour to the earliest contestant, however

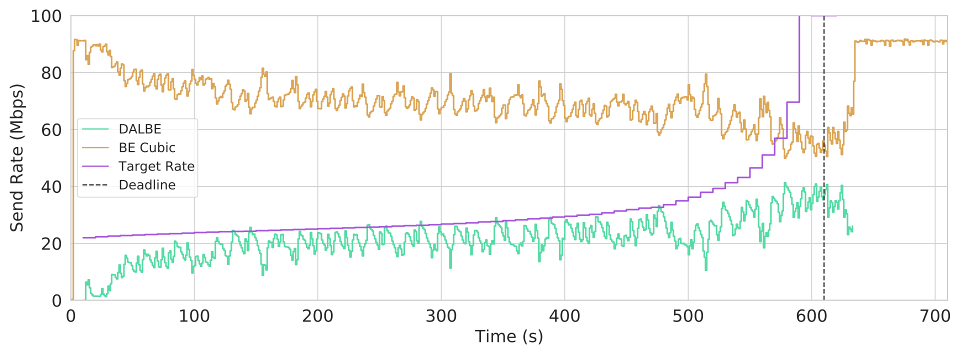


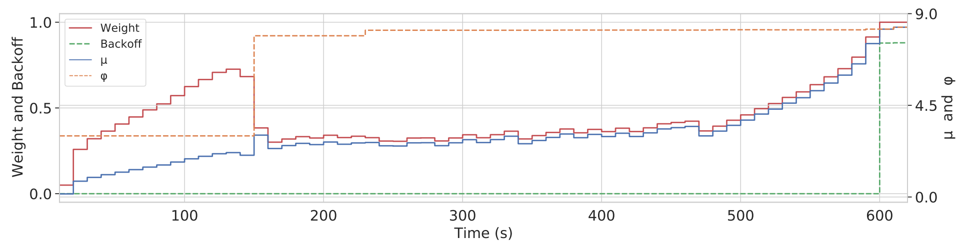
Figure 6.12: One second averages of the throughput of the latest completing run for the fairness and completion times experiment for a Vegas based DA-LBE flow using MBC as the weight policy.

towards the end there is a noticeable drop of the sending rate, which may explain why this flow is overshoots by this much. Taking a close look at the debugging graphs in figure 6.13, at around 500 seconds, there does not seem to be any apparent cause for this drop. The only noticeable factor, though minimal, and which can be seen in figure 6.13c is the change in measured queuing delay right before the t = 500 second mark. This, together with the BE flow probing for more capacity may cause the DA-LBE flow to back off as it reacts to the increase in queuing delay in the network, and will interpret this as if it is the cause of the congestion.

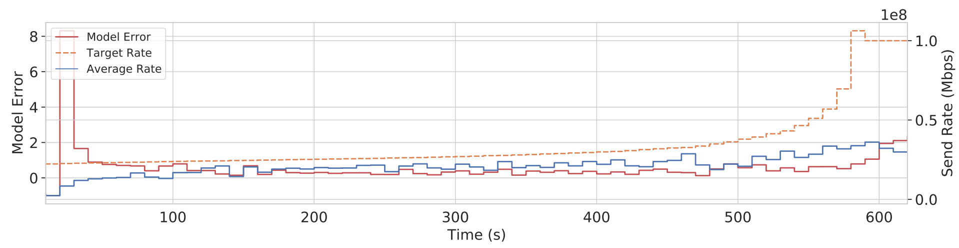
Earliest vs. Latest for PID Based Vegas

From table 6.6 it is evident that the earliest completion time by a PID flow is approximately two seconds ahead of the deadline. By looking closer at the throughput graph for this run in figure 6.14 it is apparent that there is a distinctive spike in the send rate at about t = 300 seconds, which may explain why this run is the earliest to finish. Figure 6.15a shows that right before t = 300 seconds there is a significant increase in ϕ, which suggests that the threshold for loss events was not reached before this time, and that there was a significant amount of loss events registered at this interval. This, together with $w == 1$ and *backoff* being close to one explains why this spike in the send rate occurs.

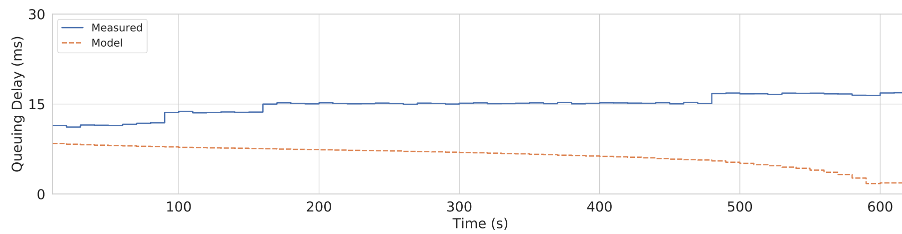
The latest contestant overshoots by about 1.5 seconds, which is not a significant amount of time. Noticeable for this run is that the DA-LBE flow is able to quite quickly reach the target rate, and maintain a stable sending rate throughout the connection. There are no significant spikes or dips which suggests that there is very little interference during the approximate 600 second connection, thus allowing it to finish this early.



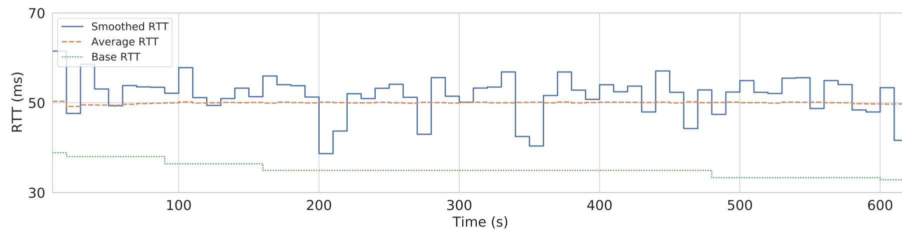
(a)



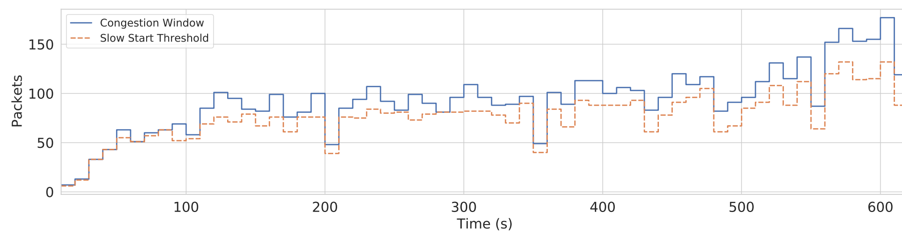
(b)



(c)



(d)



(e)

Figure 6.13: Debug graphs produced for the latest completion run for the fairness and completion times experiment. A Vegas based DA-LBE flow using MBC as the weight policy. 10 seconds increments.

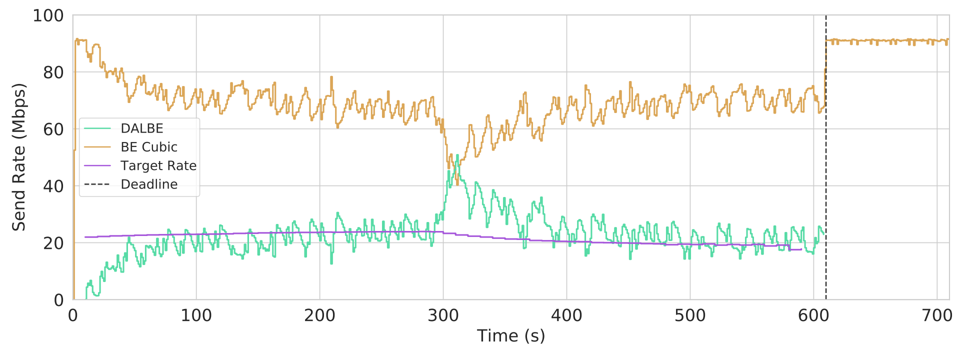
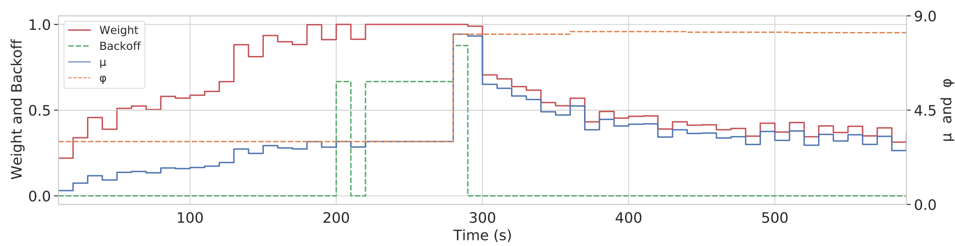
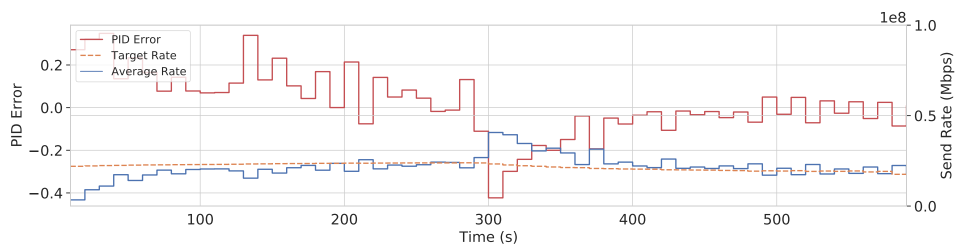


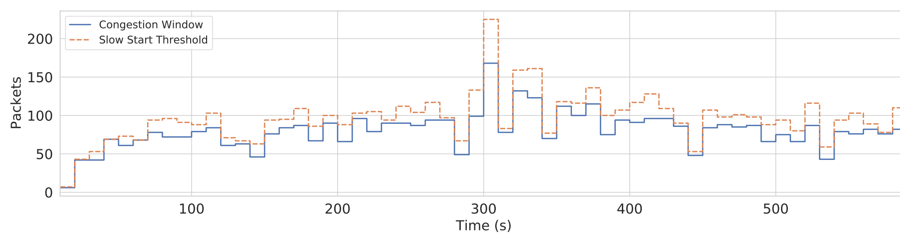
Figure 6.14: One second averages of the throughput of the earliest completing run for the fairness and completion times experiment for a Vegas based DA-LBE flow using PID as the weight policy.



(a)



(b)



(c)

Figure 6.15: Debug graphs produced for the latest completion run for the fairness and completion times experiment. A Vegas based DA-LBE flow using PID as the weight policy. 10 seconds increments.

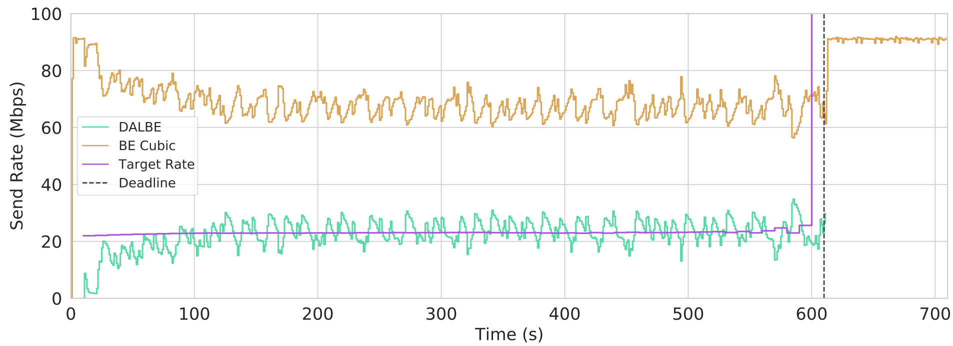


Figure 6.16: One second averages of the throughput of the latest completing run for the fairness and completion times experiment for a Vegas based DA-LBE flow using PID as the weight policy.

6.4 Summary

This chapter presented a set of experiments that were used to evaluate the network performance of the DA-LBE meta congestion controller, utilizing Vegas as the underlying congestion controller, for both MBC and PID based control. To achieve this, a set of requirements, defined by Hayes *et al.* , were used as the basis for evaluation the results.

The first experiment, targeting the network efficiency of the meta congestion controller, showed that both MBC and PID were able to utilize most of the capacity when it became available. For the MBC flow, it was also clear that it was able to follow the target deadline quite well, however, due to its average rate being just below the target rate for the entire connection, it overshot the deadline by a couple of seconds. The PID flows average rate, on the other hand, was very close to the target rate during the entire connection, thus allowing it to finish just in time.

In addition to this, I investigated what impact the use of Q16.16 *fixed point* numbers had on the calculation of w, by comparing the actual values to the expected value using *floating point* operations. This revealed that though there was some error in the calculations, however, it did not seem to be enough to have an impact of the performance of the DA-LBE meta congestion controller. This also gave an indication that the error could become a concern if the amount of variables used for the calculations increased.

In the second experiment I computed *Jain's Fairness Index* on fifty runs of an approximately 10 minute experiment, as well as recording the completion time for each run, for both MBC and PID. With this information it was possible to get an indication of how well the flows were performing in terms of their LBEness. The outcome was that both MBC and PID were showing LBE-like characteristics, however, the aggressiveness of the PID flows were a little bit higher than the MBC, as the PID flow's average rate was in general

closer to the target rate, while the MBC would stay just below. From this experiment it was also possible to see that the PID flows on average provided more stability, and thus overshooting the deadline by less than the MBC flows, yet they were both achieving expected LBE characteristics.

Chapter 7

Load and Overhead Experiments

In this chapter I present a set of experiments, aimed at providing some insight of the load and overhead introduced meta congestion controller module. In addition to this, I determine the amount of memory allocated by the meta congestion controller.

7.1 Requirements for Load and Overhead Experiments

Focusing on the fourth and final research question (RQ4);

> *How can the additional memory usage and computation load be evaluated for a DA-LBE meta congestion controller module?*

Rather than define a set of requirements from this question, I divide the question into two parts;

- How much memory is allocated on a per-connection basis, and does it scale well with the increase in connections?
- How much overhead is introduced by the major functions, and does it pose a major threat to the overall performance of the meta congestion controller?

By finding an answer to these two functions, I intend to be able to define a set of adequate requirements for the meta congestion controller, in terms of memory usage and computational load.

7.2 Memory Usage

At this point my experiments have given insights to how the DA-LBE meta congestion controller performs in terms of network efficiency, fairness and completion times. However, certain aspects of the internal performance have not yet been discussed. In this section I wish to address the the additional memory consumption introduced by implementing a meta congestion controller in the Linux kernel. More precisely I am interested in how much additional memory is allocated by the meta congestion controller per connection.

7.2.1 Reasoning about Memory Usage

The additional memory needed by each connection can be hard to measure with available tools in the Linux Kernel. However, as I know that the meta congestion controller module does not allocate memory, other than what I am aware about from the source code, it becomes quite trivial to reason about the additional usage. In this case I am mostly interested in how much memory is allocated for each connection. Each connection requires a structure, ***struct*** *dalbe* [71, *dalbe.h*], that holds required values for the DA-LBE framework as well as pointers to the connection socket and underlying congestion controller. This structure contains a decent amount of fields, and is allocated using *kmalloc* [73, *include/linux/slab.h*] when the connection is established. In addition to the allocation of the structure, an array of ten, 32 bit integers is allocated which is used for calculating the moving average of the average RTTs sampled each update interval for w. In total this comes to 336 bytes per connection. No additional memory is allocated beyond this by the DA-LBE congestion module, which suggests that the memory usage stays linear with an increasing amount of DA-LBE connections.

7.3 Function Frequency

As part of the analysis of the internals of the DA-LBE meta congestion module, I wanted to get an overview of the frequency in which the congestion controller functions, implemented by the DA-LBE meta congestion module, were invoked. I believe this knowledge is valuable as it will give a clear indication of where most of the function load is located, in terms of computation time. In addition to this I believe that this knowledge will be valuable for further work if optimization of the code becomes a concern, as these results give a good indication for where code optimization should be focused.

7.3.1 Setup

In this experiment I used *perf-stat* [50, *perf-stat(1)*] to produce a statistical function frequency profile of the DA-LBE flow. The only requirement for the profiler to produce reasonable results was that the functions which the DA-LBE meta congestion controller module implemented were called in a somewhat realistic manner. This required me to generate an environment where certain events and state changes in the TCP connection would occur, that triggered the set of implemented functions. Examples of such events are loss, fast re-transmit and recovery *etc.* , and state changes that may be triggered by these events. I came to the conclusion that this would be the case if I re-used the fairness experiment setup, as the DA-LBE flow would be in the presence of one competing BE flow as well as a specified amount of background traffic, thus creating a suitable environment for analysing the frequency the functions were invoked. I only ran the experiment for ten iterations, as the purpose of this experiment was mostly to get a quick overview of the function frequency.

For the profiler I configured it to only profile the client process, and to sample at a rate of 1000Hz. This sampling rate translates to the profiler waking up 1000 times per second to collect samples about function events from our client process.

To avoid sampling events which where not of interest, such as functions invoked by the scheduler, interrupts, etc, I set up the profiler to only follow certain events. These events were registered to the profiler by using the event probing, *perf-probe* [50, *perf-probe(1)*][1].

7.3.2 Results

The results from the statistical profiler can be seen in 7.1, and show the amount of counted events for each function. Notice that the four last func-

[1] The process of doing this for a loadable kernel module was a little challenging, and for this reason I have added this as part of the documentation.

Perf Event Count	
Function	Average Event Count
dalbe_in_ack_event	560765
dalbe_pkts_acked	560764
dalbe_cong_avoid	560269
dalbe_set_state	70
dalbe_cwnd_event	25
dalbe_ssthresh	24
dalbe_setsockops	6
dalbe_alloc	1
dalbe_release	0
dalbe_getsockops	0
dalbe_get_info	0
dalbe_undo_cwnd	0

Table 7.1: Table of the event counts produced by the function frequency experiment.

tions *dalbe_release*, *dalbe_undo_cwnd*, *dalbe_getsockops* and *dalbe_get_info* are never called. The first function, *dalbe_release*, which is not captured by the profiler, I suspect is not counted as it was missed by the profiler sampling period. The main reason why I suspect that it is missed by the profilier is the fact that it is only called once, and that I know it had been called from looking through the debugging logs. The reason why the second function, *dalbe_undo_cwnd*, is not counted is a little more confusing. I presume this has to do with the intricate chain of events that have to occur for this function to be triggered, however, I was not quite able to determine just why it was so rarely invoked. For the two latter functions, *dalbe_getsockops* and *dalbe_get_info*, it makes perfect sense why they are not counted as they are connected to system calls which the client never invokes. I argue that though these functions, *dalbe_undo_cwnd* included, may be called, their presence is usually very minimal, and thus do not pose a large threat to the performance of the DA-LBE meta congestion module.

More concerning, in terms of overhead, is the quantity of calls to the functions *dalbe_in_ack_event*, *dalbe_pkts_acked*, and *dalbe_cong_avoid*. From my knowledge of the TCP architecture in the Linux kernel I know that these functions are called on an reception-of-ACK basis. This explains why these functions are the major events captured by the profiler, and the fact that the amount of events are very similar suggests that they should be all considered as equal threats to the performance. Thus they should also be the first candidates for code optimization by the software developer.

The remaining functions are in comparison not of any major concern, as they are seldom called. However, their count may be more frequent in

a more unstable network link, where TCP events are more frequent, thus causing more frequent changes to the TCP state machine. For this reason, they should still be considered as potential candidates for optimization if their code increases drastically in complexity.

7.4 Function Overhead

The task of calculating function overhead is more intricate than showing the function frequency, and to some degree it is very difficult to give exact numbers. In this section I analyze the results from the average time and total time spent in the main functions of the meta congestion controller, with the purpose of seeing how much impact a meta congestion controller has on the system compared to a general congestion controller. I also take a closer look into the time spent inside the function where most of the DA-LBE related calculations take place. The purpose of this is to see how much impact the framework has on the meta congestion controller.

7.4.1 Setup

For this experiment I used the *function-graph* tracer of *ftrace* to capture every absolute time spent inside certain functions. As I had seen that the setup for the fairness experiments worked quite well for the function frequency experiment, I reused the same setup for this experiment. I also focused on the MBC weight policy, as the function of adapting the weight is more computationally heavy than the function used for PID. I ran the experiment for fifty iterations to average out any noise from the results, as the absolute times could easily be affected by unforeseen events in the OS.

Selecting an Appropriate Buffer Size

One major challenge I faced while setting up this experiment, which had clearly been seen in the function frequency experiment, was the fact that some of the functions that were traced were called much more frequently than others. For *ftrace*, in contrast to *perf*, this becomes quite difficult to handle, as the tracer will try to capture everything. If the buffers allocated by the tracer, one for each CPU, are not large enough, certain events will be overwritten by new events. To combat this problem, I experimented with several buffer sizes, and even made an attempt to split the traced functions into different runs. However, to be able to get a proper estimation of the total time spent in each function, all (or almost all) events should be traced. This way the total time spent inside the function can easily be compared to the time the connection is alive.

For this reason I resulted in trying a set of different sizes until I found a buffer size which allowed me to capture all the events. The size of the buffer

ended up being 320 MB in total, spread among four CPUs, which allowed to capture every event. This, however, introduced another complication; each trace log produced by one run would end up being $\approx$ 300 MB in size. As this trace was performed on one of the edge nodes, which had very limited storage, I had to be very careful of running too many iterations at the time.

Specific Tracer Options

The tracer used for this experiment, *function-graph*, may be configured in specific ways to serve a specific purpose. Following are a set options which I applied to the tracer such that it would produce results adequate for this experiment.

- **nooverwrite**: This tells the tracer that it should not overwrite the buffers when filled up, but instead ignore new trace events. This was useful, as it would imply that the buffers were filled with events from the beginning of the connection, giving more control over what was captured by the tracer.
- **funcgraph-abstime**: This allows for storing the absolute time spent inside the functions. The time is calculated as the difference between entry and exit of the function.
- **funcgraph-proc**: This option, when enabled, adds the name of the function that triggered the event, making it possible to disinguish between events.
- **nosleep**: This tells the tracers to not save the events which include scheduling events. This removes bias from the average as the time spent in context switches is not included.

Choice of Trace Events

The choice of trace events came down to; (1) what trace events were already available in the kernel, and (2) what trace events were available for the meta congestion controller module. As a result of this work using Vegas as the underlying congestion controller, the selection of trace events had to be a subset of events which were available from both the meta congestion controller module and the underlying congestion controller. This was achieved by performing a union of the available trace events from both congestion controllers, resulting in a subset of events which where present in both.

Table 7.2 shows the subset of trace events which was used for these experiments. Notice that the two last events in the table for Vegas are from the Reno congestion controller. This is because Vegas does not implement these

[2] Especially high when the interval for ϕ and/or w elapses.

Subset of Trace Events				
	Dalbe	Code Complexity	Vegas	Code Complexity
Common	dalbe_cong_avoid	Medium	tcp_vegas_cong_avoid	Medium
	dalbe_pkts_acked	Med/High[2]	tcp_vegas_pkts_acked	Low
Uncommon	dalbe_cwnd_event	Low	tcp_vegas_cwnd_event	Low
	dalbe_set_state	Low	tcp_vegas_state	Low
	dalbe_ssthresh	Low	tcp_reno_ssthresh	Low
	dalbe_undo_cwnd	Low	tcp_reno_undo_cwnd	Low

Table 7.2: Table with the subset of trace events specified for *ftrace*, distinguishing between the most common and uncommon events. The table also shows the code complexity of each function.

functions, but rather reuses the functions declared by Reno, by pointing directly to them. For this reason there are no explicit trace events available for Vegas for these functions, however tracing events of Reno will give the same result, as long as the tracer is set to only follow one process. In this case, the process which the tracer follows is the DA-LBE client available as part of the test bed software [72].

By tracing both the function of the DA-LBE meta congestion controller and corresponding function of the underlying congestion controller which the DA-LBE meta congestion controller invokes, the time spent in both functions is recorded. This allows for an estimation of how much time is spent in the DA-LBE meta congestion controller module in relation to the time spent in the Vegas module.

7.4.2 Expectations

My expectations for this experiment was mostly based on knowledge of the code for both the meta congestion controller, and the underlying congestion controller. From this I could make an assumption about how computationally heavy each traced function would be, based on the code complexity of each function.

Table 7.2 shows the code complexity perceived from analysing the source code of both congestion modules. *Low* means that there is a very minimal set of instructions inside the functions, and no evidence of looping code or other performance threatening code. *Medium* indicates that there is a intermediate set of instructions inside the function and/or some looping code which may pose some threat to the performance. *High* indicates that there is both a substantial set of instructions and/or looping code which may be performance threatening. From this table it is quite evident that the code in the common functions of the DA-LBE congestion controller module are expected to be more heavy than the code inside the Vegas module. This table suggests that there should be most overhead introduced by the common functions, and that the uncommon functions should not pose a huge threat to the performance.

It should also be mentioned that even though the DA-LBE congestion controller module is a loadable kernel module, in comparison to the Vegas congestion controller module, which is in this case built into the kernel, this does not introduce any performance penalty once the loadable kernel module has been successfully loaded into the kernel [38, *x73.html*].

Sepecific Expectations for Total Time

It is apparent that the common functions (see table 7.2) are invoked more frequently than the uncommon functions. From the profiling of function frequency performed in the previous experiment, the count of the common

events was noticeably larger than the uncommon. This leads me to believe that the time spent inside these functions will scale similarly to the results from the profiling.

7.4.3 Results

In the following sections I present the results from the average overhead per function, total time spent in each function, and the impact the DA-LBE framework has on the implementation.

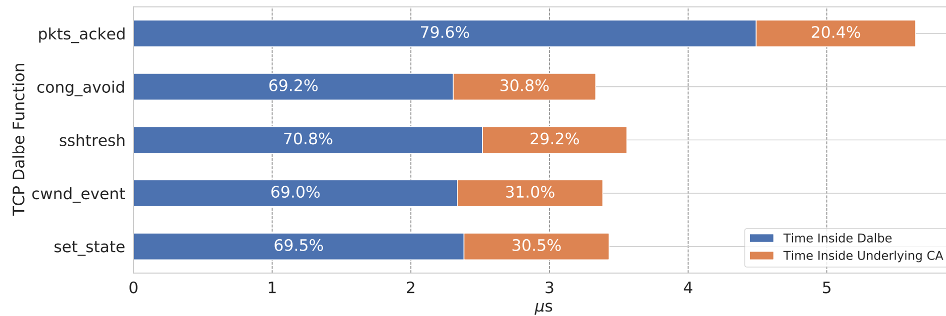
Figure 7.1a and 7.1b show bar charts of the average time and median time spent in each function. Each bar is divided into two parts; the first part (blue) shows the amount of time spent inside the DA-LBE meta congestion controller module excluding the time spent inside the underlying congestion controller. The the second part (orange) shows the amount time spent inside the, underlying congestion controller, which in this case is Vegas. The blue and orange bars stacked together represents the time actual time spent in each function, *i.e.* combining both the meta congestion controller and the underlying congestion module. The percentage is relative to the actual time spent inside the function.

Figure 7.1c shows a bar chart of the total time spent inside each of the functions. It has been scaled to logarithmic times, as the difference between total times of the events was so substantial.

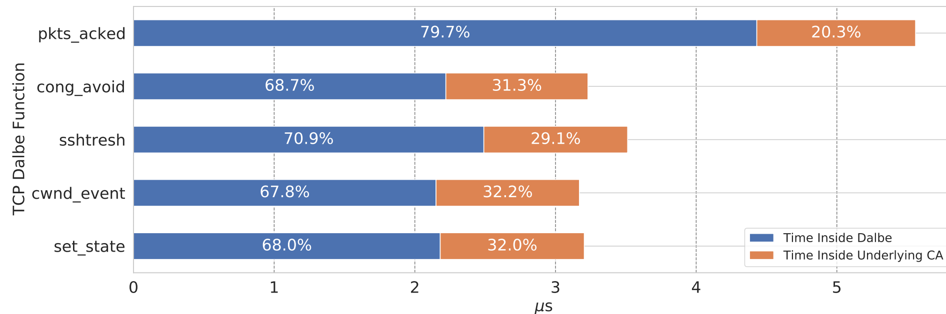
Function Overhead Introduced by Common Events

The average overhead introduced by the common events, seen in figure 7.1a, is somewhat close to the expected outcome. The *pkts_acked* function introduces the most overhead, at about 80% on average, which I suspect is due to the heavy calculations performed within the Dalbe meta congestion controller module for this function. For the *cong_avoid* the introduced average overhead is about 70%. I expected that the amount of work would be quite even between the two, based on the knowledge from the source code, however, it seems to be that the DA-LBE congestion controller module is still the culprit in terms of overhead.

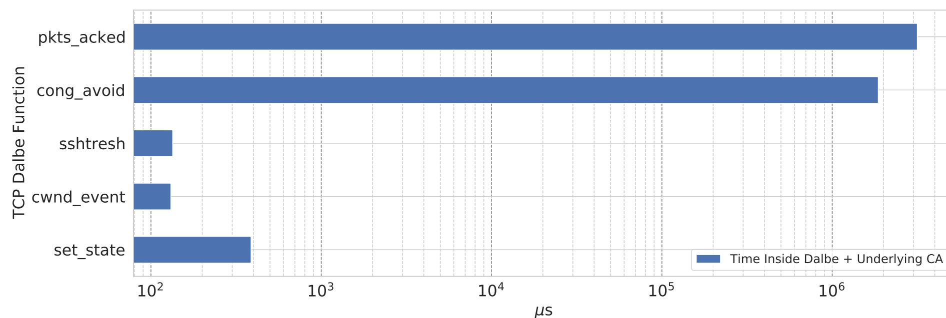
By looking at table 7.3, there are three factors that may suggest that even though the overhead introduced by the meta congestion module is quite substantial most of the values are quite small. The first noticeable factor is that both the *mean* and *median* value are quite similar. The second noticeable factor is that both the *maximum* value for these common events are significantly larger than the *median* value. Finally, the third noticeable factor is that the standard deviation is quite low, where only the *dalbe_pkts_acked* has a standard deviation above one. This would suggest that the average overhead may be increased by a range of substantially large outliers.



(a) Average function overhead introduced by the meta congestion controller from 50 iterations.



(b) Median function overhead introduced by the meta congestion controller from 50 iterations.



(c) Total time spent in each function by the meta congestion controller.

Figure 7.1: A set of figures that illustrate the function overhead introduced by the DA-LBE meta congestion controller, averaged over 50 iterations of an approximately 600 second experiment with one DA-LBE flow competing with a BE flow.

Ftrace Times From 50 Iterations							
Function	Count	Total Time	Median	Mean	Stddev	Min	Max
dalbe_pkts_acked	560142	3159447μs	5.56μs	5.64μs	1.57μs	3.19μs	154.00μs
tcp_vegas_pkts_acked	560142	644002μs	1.13μs	1.15μs	0.29μs	0.66μs	40.13μs
dalbe_cong_avoid	559013	1863550μs	3.23μs	3.33μs	0.58μs	1.96μs	52.54μs
tcp_vegas_cong_avoid	559013	574276μs	1.01μs	1.03μs	0.30μs	0.61μs	42.02μs
dalbe_set_state	113	387μs	3.20μs	3.43μs	0.62μs	2.77μs	7.56μs
tcp_vegas_state	113	118μs	1.02μs	1.05μs	0.19μs	0.89μs	2.81μs
dalbe_cwnd_event	39	131μs	3.17μs	3.38μs	0.93μs	2.87μs	8.39μs
tcp_vegas_cwnd_event	39	41μs	1.02μs	1.05μs	0.16μs	0.92μs	1.91μs
dalbe_ssthresh	38	134μs	3.51μs	3.56μs	0.39μs	3.03μs	5.24μs
tcp_reno_ssthresh	38	39μs	1.02μs	1.04μs	0.14μs	0.90μs	1.78μs

Table 7.3: A table with values related to the average function overhead for both common and uncommon trace events.

Overhead Introduced by Uncommon Events

The average overhead introduced by the uncommon events is a little more unexpected. From my reasoning about the expected overhead, based on the complexity within each function, it would have been reasonable to expect very little overhead introduced by the DA-LBE meta congestion controller module for these functions. However, it is clear that this is not the case, as all the uncommon events have $\approx 70\%$ introduced overhead by the meta congestion controller. Even the simplest function in terms of code complexity, being *dalbe_ssthresh* in this case, introduces more overhead than the least intrusive function of the common events.

By looking at table 7.3, the first noticeable factor is that the amount of samples on average per iteration is substantially lower than that of the common events. This could suggest that the amount of samples are not enough to get a proper estimate of these functions. However, it could also suggest that since this code is seldom used throughout the connection the CPU may not have the code stored in memory, causing a page fault. The time penalty of the page fault may be what is causing these functions to introduce substantial overhead, however, this can only be speculated about as there is no evidence that this is actually the case.

Total Time Spent In Functions

At this point it is clear that the meta congestion controller introduces approximately 70% to 80% overhead. However, by looking at figure 7.1c, it may be possible to see how much impact this overhead has on the actual connection. What is clear from this figure, is that most of the time is spent inside the common functions, while the very little time is spent inside the uncommon functions. From table 7.3 it is apparent that in total, about five seconds is spent in the common functions combined, over the approximately 600 second connection. While approximately only 550μs is spent in total for the uncommon functions combined.

From this observation it is apparent that even though the overhead may be quite large at times, the total time spent inside the functions is still minimal. For the common functions, the combined total time is still below 1% of the entire connection, and for the uncommon events it is almost not recognizable. This, together with the promising results from the network performance experiments, suggest that the overhead is still not enough to have a noticeable effect on the overall performance of the meta congestion controller.

Impact of the Update Intervals for ϕ and w

Lastly, I take a closer look at the impact that the update intervals for ϕ and w have on the meta congestion controller module. The function responsible for

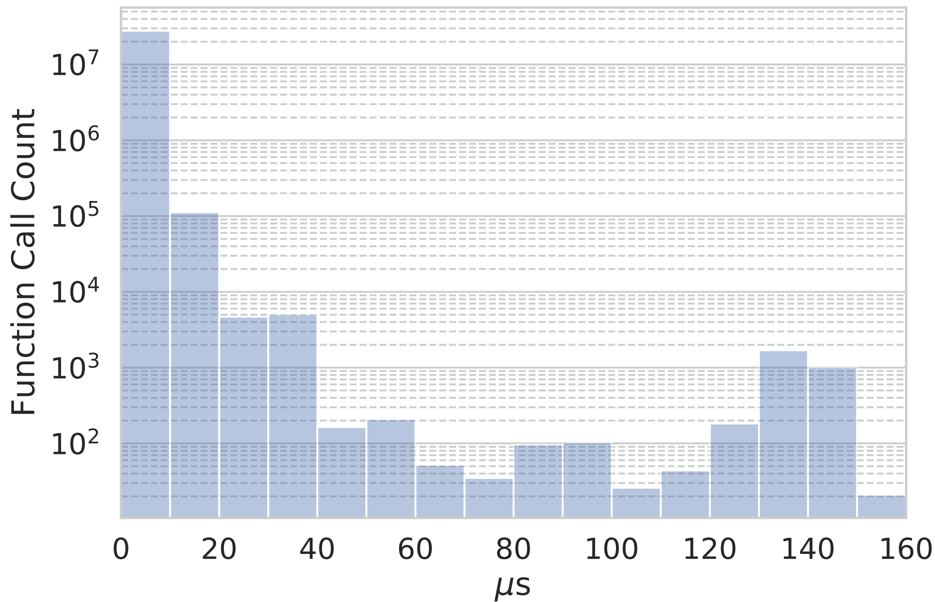


Figure 7.2: Histogram showing the samples for the *pkts_acked* function.

these update intervals is *dalbe_pkts_acked*. For this function, the maximum recorded time was 154μs, seen in table 7.3, which is quite far above the average of approximately 5μs.

From the approximately 600 seconds experiment, with an update interval of 10 seconds, I would expect about 60 updates of ϕ and w. As the start times for their periodic updates is the same, we can assume that both are triggered at the same time *i.e.* in the same function call. Considering that there is about 60 updates per run for 50 iterations, we would expect $60 \times 50 = 3000$ of the samples to have a substantially overhead due to the update of these values.

Figure 7.2 illustrates the samples for *dalbe_pkts_akced* from the fifty runs. From this figure, it is apparent that most of the values are skewed to the left, suggesting that they are achieving a sub 20μs overhead. Noticeably, at 120μs to 150μs the samples reach approximately 10^3. This fits quite well with the expectation for events that would be extra computational heavy due to the update of ϕ and w. From this observation, it would also be safe to assume that the spike seen around this area, in fact is caused by these updates. This would also suggest that the effect these intervals have on the meta congestion controller is not substantial. However, as the values seem to be very predictable, it would also suggest that if the interval times were to be shortened, the overhead for this function would increase.

7.5 Summary

In this chapter chapter I presented a set of experiments aimed at providing some insight to the memory usage, and computational load and overhead introduced by the meta congestion controller.

I first reasoned about the memory usage of the meta congestion controller, and how it scaled linearly with the increase of DA-LBE connections. Following this, I ran a simple experiment, which gave an indication of the frequency at which the functions implemented by the meta congestion controller were invoked. This gave some good insight into where the most of the function overhead would be expected.

Secondly, I ran an experiment, based on the fairness experiment in chapter 6, which gave an insight into how much function overhead was introduced by the meta congestion controller in comparison to the underlying congestion controller, and how much time was spent in each function in total during an approximately 600 second experiment. From this experiment it was apparent that the meta congestion controller was responsible for approximately 70% to 80% of the absolute time spent in the functions, however, the considering the frequency of some of these functions, the overhead did not pose a large threat to the performance of the meta congestion controller.

Lastly, I took a closer look at the function responsible for updating w and ϕ. From this it was quite clear that the largest recorded overheads could be tied to the update intervals for these values. This suggested that even though these updates increased the average overhead, they had little effect on the overall performance of the meta congestion controller.

Chapter 8

Conclusion

This thesis aimed to produce a stable, long-term solution for DA-LBE transport services, by creating an implementation for the Linux OS.

An *agile inspired* development process was used to develop the DA-LBE framework as a meta congestion controller, *Dalbe*, for the Linux OS. Dalbe is implemented as a loadable kernel module, following common conventions for development in the Linux kernel. It requires minimal changes to the kernel, uses the existing socket API, and can be easily augmented with congestion control models.

Two test environments were built, each serving a different purpose. The first test environment, a virtual machine based setup, was constructed for early testing on the kernel module, allowing Dalbe to be thoroughly checked for memory leaks, bugs and errors. The second test environment, a hardware based setup constructed together with a colleague, consisted of five connected machines, which emulated a bottleneck link using common tools available for the Linux OS. This allowed for running experiments on the meta congestion module in a near-realistic, stable environment.

By recreating a set of experiments based on previous work by Hayes *et al.* [35] and Wallenburg [74], the network performance of Dalbe was analysed, using Vegas as the underlying congestion controller for both MBC and PID based weight control. These experiments demonstrated that the meta congestion controller was able to utilize the available capacity in the network. They also demonstrated that in this case PID based control would tend to be a little more aggressive than MBC, however, both were able to maintain their desired LBE characteristics.

The impact of using Q16.16 *fixed point* numbers had on the calculation of the weight, w, was investigated. This revealed that the maximum error observed was less than 1% with respect to the expected value, which suggested that in this case it was not enough to have any large impact on the quality of the calculations. The investigation also demonstrated that increasing the amount of variables in the calculations could potentially increase the

overall error.

The memory consumption of the meta congestion controller was reasoned about, based on the knowledge of its internal structure. From this it was clear that the amount of memory consumed by the meta congestion controller was quite insignificant, only 336 bytes per connection, and that the allocation of memory scales linearly with the increase of DA-LBE connections.

Finally, a set of tracing tools available for the Linux OS were used to investigate the function frequency and the function overhead introduced by the meta congestion controller. This indicated quite clearly which functions were the major concerns in terms of overhead. It also revealed that the function overhead introduced by the meta congestion controller was between 70% and 80%, which could seem concerning in terms of performance. However, the total time spent in each function revealed that even for the most frequently called functions, the total time spent in each function was quite insignificant, as combined they were responsible for less than 1% of the entire time spent in the connection. It was also observed that the largest overheads recorded for the function responsible of updating w and ϕ, could be tied to the amount of expected update intervals for the connection, suggesting that in this case these updates had little effect on the overall performance of the meta congestion controller.

Dalbe provides a thoroughly tested and debugged meta congestion controller for DA-LBE transport services, utilizing Vegas as the underlying congestion controller for both MBC and PID based control, which makes it a good start in the direction of providing a stable, long-term solution.

8.1 Future Work

Finally, I want to discuss what would be a reasonable direction for future work on the meta congestion controller, based on my own opinions and the findings of this thesis.

8.1.1 Improvements to the Meta Congestion Controller

The meta congestion controller does not come without some short comings, which were pointed out it chapter 4. In this section I wish to emphasize the shortcomings which, in my opinion, should be addressed first.

Adjusting the aggressiveness of loss-based control should be considered as the first priority, as this mechanism does not require too much additional code, and when accomplished it will allow for testing the meta congestion controller with most underlying congestion controllers.

Sadly, I was not able to finish implementing the mechanism for adjusting the aggressiveness for underlying, loss-based, congestion controllers, which limits the current implementation to being solely delay-based. However, I did make an attempt of getting this feature in before the deadline, which can be found in the *git* repository for the meta congestion controller [70] under the branch named *"loss-based"*, together with some notes on how this functionality may be accomplished.

Improving the architecture for better modularity is in my opinion a good step in the direction for expanding the meta congestion controller with more functionality, beyond what is present in the current implementation.

As I stated in chapter 4, the current architecture of the meta congestion controller, is serving its purpose. However, it could be made more modular by *e.g.* taking inspiration from the pluggable congestion controller interface in the Linux kernel, and providing a similar pluggable interfaces for adding additional models to the meta congestion controller.

Model based control for Cubic should eventually be implemented. This should make it more appealing for the users, as Cubic is the default congestion controller in Linux.

For the time being there is only one available model for the meta congestion controller, and that is for Vegas. It was never a major goal for this thesis to implement more than one model, however, at one point during this project I did make an attempt at understanding how a model for Cubic could be implemented using *fixed point* operations, which resulted in some code that successfully performs a *cubic root* using *fixed point* operations, also located in the test repository [70]. As the major difficulty of implementing a model for Cubic for the DA-LBE meta congestion controller is to correctly and efficiently compute this *cubic root* with *fixed point* operations, this code may be very useful for future work.

8.1.2 Improvements to Testing and Experimentation

I also wish to emphasize some improvements that could be done to the testing and experimentation, based on observations during this work, and my own opinions.

Enhancing the Function Frequency Experiment

One thing I realized from the results of the function frequency experiment was that some of the functions were very rarely invoked during the approximately 600 second connection. At the time being I did not consider the fact that the experimental setup could be changed in a way that would possibly have caused more frequent calls to these uncommon functions. What would

be interesting would to re-construct this experiment in such way that the frequency of the uncommon functions would be increased, to investigate if this has any effect on sampled overhead for these functions.

Further Investigation of Fixed Point Operations

While investigating the impact of using *fixed point* operations, it was quite clear that using more variables in the calculations would increase the overall error. As the *fixed point* operations implemented in the meta congestion controller allow for using both Q16.16 and Q2.30, it would be interesting to investigate further if combining the two representations would be beneficial for the meta congestion controller. What would also be very interesting during this investigation, would be to map out the *fixed point* variables, and determine how prone to overflows they are. Determining these factors should, in my opinion, contribute to increasing the robustness of the meta congestion controller.

Network Performance Experiments over the Internet

Finally, the test environment used for evaluating the meta congestion controller was based on emulation, using tools available for Linux. What would be a reasonable step in the direction of making the meta congestion controller even more robust, would be to run experiments over the Internet.

Bibliography

[1] Jeff Ahrenholz. *CORE Documentation*. 2018. URL: http://coreemu.github.io/core/ (visited on 04/03/2020).

[2] M. Allman. *TCP Congestion Control with Appropriate Byte Counting (ABC)*. RFC 3465. http://www.rfc-editor.org/rfc/rfc3465.txt. RFC Editor, Feb. 2003. URL: http://www.rfc-editor.org/rfc/rfc3465.txt.

[3] M. Allman, S. Floyd and C. Partridge. *Increasing TCP's Initial Window*. RFC 3390. RFC Editor, Oct. 2002.

[4] M. Allman, V. Paxson and E. Blanton. *TCP Congestion Control*. RFC 5681. http://www.rfc-editor.org/rfc/rfc5681.txt. RFC Editor, Sept. 2009. URL: http://www.rfc-editor.org/rfc/rfc5681.txt.

[5] Rex Black. *Foundations of software testing : ISTQB certification*. eng. 2012.

[6] Bob Braden et al. *Recommendations on Queue Management and Congestion Avoidance in the Internet*. RFC 2309. http://www.rfc-editor.org/rfc/rfc2309.txt. RFC Editor, Apr. 1998. URL: http://www.rfc-editor.org/rfc/rfc2309.txt.

[7] R. Braden. *Requirements for Internet Hosts - Application and Support*. STD 3. RFC Editor, Oct. 1989.

[8] Robert Braden. *Requirements for Internet Hosts - Communication Layers*. STD 3. http://www.rfc-editor.org/rfc/rfc1122.txt. RFC Editor, Oct. 1989. URL: http://www.rfc-editor.org/rfc/rfc1122.txt.

[9] Lawrence Brakmo, Sean O'Malley and Larry Peterson. 'TCP Vegas: new techniques for congestion detection and avoidance'. eng. In: SIGCOMM '94. ACM, 1994, pp. 24–35. ISBN: 0897916824.

[10] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159. http://www.rfc-editor.org/rfc/rfc7159.txt. RFC Editor, Mar. 2014. URL: http://www.rfc-editor.org/rfc/rfc7159.txt.

[11] B. Cain et al. *Internet Group Management Protocol, Version 3*. RFC 3376. http://www.rfc-editor.org/rfc/rfc3376.txt. RFC Editor, Oct. 2002. URL: http://www.rfc-editor.org/rfc/rfc3376.txt.

[12] Brian E. Carpenter. *Architectural Principles of the Internet*. RFC 1958. http://www.rfc-editor.org/rfc/rfc1958.txt. RFC Editor, June 1996. URL: http://www.rfc-editor.org/rfc/rfc1958.txt.

[13] J. Chu et al. *Increasing TCP's Initial Window*. RFC 6928. http://www.rfc-editor.org/rfc/rfc6928.txt. RFC Editor, Apr. 2013. URL: http://www.rfc-editor.org/rfc/rfc6928.txt.

[14] Jonathan Corbet, Alessandro Rubini and Greg Kroah-Hartman. *Linux Device Drivers*. eng. Sebastopol: O'Reilly Media, Incorporated, 2005. ISBN: 0596005903.

[15] corbnet. *Pluggable congestion avoidance modules*. Mar. 2005. URL: https://lwn.net/Articles/128681/ (visited on 17/07/2020).

[16] coreemu. *Common Open Research Emulator*. https://github.com/coreemu/core. 2020.

[17] Intel Corporation. *Intel Ethernet Server Adapter I210-T1*. URL: https://www.intel.com/content/www/us/en/products/network-io/ethernet/gigabit-adapters/server-i210-t1.html (visited on 05/07/2020).

[18] Intel Corporation. *Intel® Ethernet Server Adapter I350-T4V2*. URL: https://www.intel.com/content/www/us/en/products/network-io/ethernet/gigabit-adapters/server-i350-t4v2.html (visited on 05/07/2020).

[19] Oracle Corporation. *Oracle VM VirtualBox*. http://virtualbox.org/.

[20] Oracle Corporation. *Oracle VM VirtualBox User Manual*. URL: https://www.virtualbox.org/manual/ (visited on 05/07/2020).

[21] D. Cuomo. *Support for LEDBAT: Public Service Announcement*. Oct. 2018. URL: https://techcommunity.microsoft.com/t5/networking-blog/support-for-ledbat-public-service-announcement/ba-p/339796 (visited on 17/07/2020).

[22] Autotest developers. *Autotest*. 2012. URL: http://autotest.github.io/ (visited on 24/07/2020).

[23] LTP developers. *Testing Linux, one syscal at the time*. 2012. URL: http://linux-test-project.github.io// (visited on 24/07/2020).

[24] Fabric. *Simple, Pythonic remote execution and deployment*. https://github.com/fabric/fabric. 2020.

[25] S Floyd and V Jacobson. 'Random early detection gateways for congestion avoidance'. eng. In: *IEEE/ACM Transactions on Networking* 1.4 (1993), pp. 397–413. ISSN: 1063-6692.

[26] Python Software Foundation. *Python*. http://python.org/.

[27] Wireshark Foundation. *Wireshark*. http://wireshark.org/.

[28] Inc Free Software Foundation. *An Inline Function is As Fast As a Macro*. 2020. URL: https://gcc.gnu.org/onlinedocs/gcc-7.1.0/gcc/Inline.html (visited on 29/07/2020).

[29] Google. *Google Test*. https://github.com/google/googletest. 2020.

[30] Brendan Gregg. *Systems performance : enterprise and the cloud*. eng. Upper Saddle River, NJ, 2013.

[31] The Tcpdump Group. *Tcpdump & Libcap*. URL: http://tcpdump.org/ (visited on 05/07/2020).

[32] WAND Network Research Group. *Libtrace*. URL: https://research.wand.net.nz/software/libtrace.php (visited on 05/07/2020).

[33] Sangtae Ha, Injong Rhee and Lisong Xu. 'CUBIC: a new TCP-friendly high-speed TCP variant'. eng. In: *Operating systems review* 42.5 (2008), pp. 64–74. ISSN: 0163-5980.

[34] G Hasegawa, K Kurata and M Murata. 'Analysis and improvement of fairness between TCP Reno and Vegas for deployment of TCP Vegas to the Internet'. eng. In: IEEE, 2000, pp. 177–186. ISBN: 9780769509211.

[35] David A Hayes et al. 'A framework for less than best effort congestion control with soft deadlines'. eng. In: IFIP, 2017, pp. 1–9. ISBN: 9783901882944.

[36] C. Hedrick. *Routing Information Protocol*. RFC 1058. http://www.rfc-editor.org/rfc/rfc1058.txt. RFC Editor, June 1988. URL: http://www.rfc-editor.org/rfc/rfc1058.txt.

[37] Stephen Hemminger. *TCP infrastructure split out*. Mar. 2005. URL: https://lwn.net/Articles/128626/ (visited on 17/07/2020).

[38] Bryan Henderson. *Linux Loadable Kernel Module HOWTO*. Sept. 2006. URL: https://tldp.org/HOWTO/Module-HOWTO/ (visited on 23/08/2020).

[39] T. Henderson et al. *The NewReno Modification to TCP's Fast Recovery Algorithm*. RFC 6582. http://www.rfc-editor.org/rfc/rfc6582.txt. RFC Editor, Apr. 2012. URL: http://www.rfc-editor.org/rfc/rfc6582.txt.

[40] 'IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems'. In: *IEEE Std 1588-2019 (Revision ofIEEE Std 1588-2008)* (2020), pp. 1–499.

[41] 'IEEE Standard for Ethernet'. In: *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)* (2018), pp. 1–5600.

[42] Apple Inc. *tcp_ledbat.c*. 2010. URL: https://opensource.apple.com/source/xnu/xnu-1699.32.7/bsd/netinet/tcp_ledbat.c (visited on 21/05/2019).

[43] V Jacobson. 'Congestion avoidance and control'. eng. In: SIGCOMM '88. ACM, 1988, pp. 314–329. ISBN: 9780897912792.

[44] V. Jacobson et al. *TCP BBR congestion control comes to GCP - your Internet just got faster*. July 2017. URL: https://cloud.google.com/blog/products/gcp/tcp-bbr-congestion-control-comes-to-gcp-your-internet-just-got-faster (visited on 17/07/2020).

[45] Van Jacobson, Bob Braden and Dave Borman. *TCP Extensions for High Performance*. RFC 1323. http://www.rfc-editor.org/rfc/rfc1323.txt. RFC Editor, Apr. 1992. URL: http://www.rfc-editor.org/rfc/rfc1323.txt.

[46] R Jain, D Chiu and W Hawe. 'A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems'. eng. In: (1998).

[47] jbucar. *D-ITG, Distributed Internet Traffic Generator*. https://github.com/jbucar/ditg. 2020.

[48] Project Jupyter. *Jupyter Notebook*. URL: https://jupyter.org/ (visited on 05/07/2020).

[49] The Linux Kernel. *The Linux Kernel documentation*. URL: https://www.kernel.org/doc/html/v5.4/ (visited on 05/07/2020).

[50] The Linux Kernel. *The Linux man-pages project*. URL: https://man7.org/linux/man-pages (visited on 05/07/2020).

[51] M. Larabel. *The Linux Kernel Enters 2020 At 27.8 Million Lines In Git But With Less Developers For 2019*. Jan. 2020. URL: https://www.phoronix.com/scan.php?page=news_item&px=Linux-Git-Stats-EOY2019 (visited on 21/07/2020).

[52] Robert Love. *Linux kernel development*. eng. Place of publication not identified, 2010.

[53] D. S. Miller. *Netdev Group's -next networking tree*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/netdev/net-next.git (visited on 05/07/2020).

[54] D. Mills et al. *Network Time Protocol Version 4: Protocol and Algorithms Specification*. RFC 5905. http://www.rfc-editor.org/rfc/rfc5905.txt. RFC Editor, June 2010. URL: http://www.rfc-editor.org/rfc/rfc5905.txt.

[55] Jeffrey C. Mogul and K. K. Ramakrishnan. 'Eliminating Receive Livelock in an Interrupt-Driven Kernel'. In: *ACM Trans. Comput. Syst.* 15.3 (Aug. 1997), pp. 217–252. ISSN: 0734-2071. DOI: 10.1145/263326.263335. URL: https://doi-org.ezproxy.uio.no/10.1145/263326.263335.

[56] J. Moy. *OSPF specification*. RFC 1131. http://www.rfc-editor.org/rfc/rfc1131.txt. RFC Editor, Oct. 1989. URL: http://www.rfc-editor.org/rfc/rfc1131.txt.

[57] Jon Postel. *Internet Protocol*. STD 5. http://www.rfc-editor.org/rfc/rfc791.txt. RFC Editor, Sept. 1981. URL: http://www.rfc-editor.org/rfc/rfc791.txt.

[58] Jon Postel. *Transmission Control Protocol*. STD 7. http://www.rfc-editor.org/rfc/rfc793.txt. RFC Editor, Sept. 1981. URL: http://www.rfc-editor.org/rfc/rfc793.txt.

[59] K. Ramakrishnan, S. Floyd and D. Black. *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC 3168. http://www.rfc-editor.org/rfc/rfc3168.txt. RFC Editor, Sept. 2001. URL: http://www.rfc-editor.org/rfc/rfc3168.txt.

[60] I. Rhee et al. *CUBIC for Fast Long-Distance Networks*. RFC 8312. RFC Editor, Feb. 2018.

[61] Rami Rosen. *Linux Kernel Networking : Implementation and Theory*. eng. Berkeley, CA, 2014.

[62] D. Rossi et al. 'LEDBAT: The new BitTorrent congestion control protocol'. In: *Proceedings - International Conference on Computer Communications and Networks,ICCCN*. May 2010.

[63] J Saltzer, D Reed and D Clark. 'End-to-end arguments in system design'. eng. In: *ACM Transactions on Computer Systems (TOCS)* 2.4 (1984), pp. 277–288. ISSN: 0734-2071.

[64] seladb. *PcapPlussPluss*. URL: https://pcapplusplus.github.io/ (visited on 05/07/2020).

[65] *Selected chapters from Software engineering : compiled from Software engineering, 9th edition, Ian Sommerville*. eng. Harlow, 2014.

[66] S. Shalunov et al. *Low Extra Delay Background Transport (LEDBAT)*. RFC 6817. http://www.rfc-editor.org/rfc/rfc6817.txt. RFC Editor, Dec. 2012. URL: http://www.rfc-editor.org/rfc/rfc6817.txt.

[67] Tom Shanley. *Protected mode software architecture*. eng. Reading, Mass, 1996.

[68] L. Storbukås. 'Implementing Less than Best Effort with Deadlines'. MA thesis. University of Oslo - Department of Informatics, May 2018.

[69] H. P. Tandberg. *DALBE Analysis*. https://bitbucket.org/henningtandberg/tcp-dalbe-analysis/src/master/. 2020.

[70] H. P. Tandberg. *DALBE Test Suite*. https://bitbucket.org/henningtandberg/tcp-dalbe-test/src/master/. 2020.

[71] H. P. Tandberg. *TCP DALBE*. https://bitbucket.org/simula-mosaic/mosaic-students-henning/src/master/. 2020.

[72] H. P. Tandberg and M. Bratland. *DALBE Test Bed*. https://bitbucket.org/simula-mosaic/testbed/src/master/. 2020.

[73] Torvalds. *Linux kernel source tree*. https://github.com/torvalds/linux/tree/v5.4. 2020.

[74] H. Wallenburg. 'Libdalbe - A library for developing Deadline-Aware Less-than Best Effort transport services'. MA thesis. University of Oslo - Department of Informatics, Aug. 2018.

[75] Lisong Xu, K Harfoush and Injong Rhee. 'Binary increase congestion control (BIC) for fast long-distance networks'. eng. In: vol. 4. IEEE, 2004, 2514–2524 vol.4. ISBN: 0780383559.

Appendices

Appendix A

Architecture and Internals

A.1 Pluggable Congestion Controller Interface

This section defines a synopsis for each function which Dalbe adopts from the TCP
congestion controller interface *tcp_congestion_ops* [73, *include/net/tcp.h*].

A.1.1 Initialize Private Data

Signature

```
static void dalbe_init(struct sock *sk)
```

Synopsis

This functions is called at the start of the connection, and used for allocating the *dalbe* structure with its initial values and adding the structure to a hashtable which keeps track of the data for each connection. Also calls upon the underlying congestion controllers initiation function if it is implemented.

A.1.2 Cleanup Private Data

Signature

```
static void dalbe_release(struct sock *sk)
```

Synopsis

This function is called at the end of the connection, and is used to clean up any dynamically allocated memory and removes the DA-LBE structure from the hashtable used to keep track of connection data. Also calls underlying congestion controller cleanup function if it is implemented.

A.1.3 Calculate New Slow Start Threshold

Signature

```
static u32 dalbe_ssthresh(struct sock *sk)
```

Synopsis

This function is used to calculate a new slow start threshold. DA-LBE does not do any additional calculation of the slow start threshold, so in our case it is just a pass-through to the underlying congestion controller *ssthresh* function.

A.1.4 Inform About State Change

Signature

```
static void dalbe_set_state(struct sock *sk, u8
    new_state)
```

Synopsis

This function is called in the event of a TCP state change. It is used to inform the congestion controller about a state change. DA-LBE does not require to keep track of the state explicitly, so in our case it is just a pass-through to the underlying congestion controller *set_state* funciton.

A.1.5 Calculate New Congestion Window

Signature

```
static void dalbe_cong_avoid(struct sock *sk, u32
     ack, u32 acked)
```

Synopsis

This function is called during the congestion avoidance phase on the reception of an ACK. It allows DA-LBE to keep track of the proportion of how much the congestion window is reduced during the congestion avoidance phase. It achieves this by first letting the underlying congestion controller calculate the new sending window size, followed by accumulating the proportion of how much it was reduced given the type of congestion event that triggered the change. If the cause of reduction was due to loss, $w == 1$, and backoff > 0, it may randomly skip the loss event by resetting the window to its previous value.

A.1.6 Inform About New Congestion Event

Signature

```
static void dalbe_cwnd_event(struct sock *sk,
    enum tcp_ca_event ev
```

Synopsis

This function is called when a congestion controller event occurs (see [73, *include/net/tcp.h:964*]. It is used for counting ECN events as well as calculating the time between congestion events for loss based controllers. Also calls the underlying congestion controller *cwnd_event* if implemented.

A.1.7 Upon Arrival of an ACK

Signature

```
static void dalbe_in_ack_event(struct sock *sk,
    u32 flags)
```

Synopsis

This function is always called upon the arrival of an ACK. DA-LBE uses this function explicitly to count the amount of bytes sent during the connection. Also calls the underlying congestion controller *in_ack_event* if implemented.

A.1.8 Calculate New Window in the Event of Loss

Signature

```
static u32 dalbe_undo_cwnd(struct sock *sk)
```

Synopsis

This function is called in the event of a loss to recalculate the congestion window. DA-LBE does not perform any additional calculation of the congestion window, so in our case it is just a pass-through to the underlying congestion controller *undo_cwnd*.

A.1.9 Packet Accounting in the Event of an ACK

Signature

```
static void dalbe_pkts_acked(struct sock *sk,
    const struct ack_sample *sample)
```

Synopsis

This function is called if the received ACK removes anything from the re-transmit queue. This i where the main functionality of DA-LBE lays, and it implements the following:

- If the update interval for ϕ has expired, an attempt to update ϕ is made. This however, depends on the amount of loss that has occurred during the interval.
- If the update interval for w has expired, the following is done in the order presented.
  - The average byte rate is updated.
  - The amount of outstanding data is updated.
  - The target deadline is updated.
  - The target byte rate is updated.
  - A new value of w is calculated depending on the weight update policy (PID or MBC.
  - μ and the *backoff* values are updated accordingly.

  Note that this is done in the order presented by Hayes *et al.* [35].
- At the end the very end of the ACK processing, one of two methods for adjusting the aggressiveness may be performed depending on the type of underlying congestion controller.
  - If the underlying congestion controller is loss-based, DA-LBE randomly generates a fake loss.
  - Else, if the underlying congestion controller is delay-based, DA-LBE alters the RTT sample which is passed on to the underlying congestion controller.

Also it calls upon the underlying congestion controller *pkts_acked* if implemented.

A.1.10 Get Information About the Congestion Controller

Signature

```
size_t dalbe_get_info(struct sock *sk, u32 ext,
    int *attr, union tcp_cc_info *info)
```

Synopsis

This function is usually called from user space to return information about the congestion controller. DA-LBE does not return any specific information, but rather acts as a pass-through to the underlying congestion controller *get_info* if implemented.

A.1.11 Set Custom Socket Options

Signature

```
int dalbe_setsockops(struct sock *sk, int optname
    , char __user *optval, unsigned int) optlen
```

Synopsis

This function is called from user space, and allows for passing data directly to the DA-LBE congestion controller module. The accepted socket options can be seen in appendix B.2.
This function handles errors according to the convention used by *setsockopt*[73, *setsockopt(2)*], allowing for proper error handling.

A.1.12 Get Custom Socket Options

Signature

```
int dalbe_getsockops(struct sock *sk, int optname
    , char __user *optval, int __user *optlen)
```

Synopsis

This function is called from user space, and allows for retreiving data directly from the DA-LBE congestion controller module. The accepted socket options can be seen in appendix B.2.
This function handles errors according to the convention used by *getsockopt*[73, *getsockopt(2)*], allowing for proper error handling.

A.2 DALBE Math

This section describes the core math functions used by Dalbe.

A.2.1 Multiplication between two unsigned fixed point numbers

Signature

```
static inline u32 uq_mul(u32 a, u32
    a_scale_factor, u32 b, u32 b_scale_factor)
```

Parameters

```
u32 a
```

Fixed point number.

```
u32 a_scale_factor
```

Corresponding scale factor for *a*.

```
u32 b
```

Fixed point number.

```
u32 b_scale_factor
```

Corresponding scale factor for *b*.

Return Value

Returns a fixed point number scaled by the largest of the two scale factors.

Synopsis

This function returns the product of two unsigned fixed point numbers.
The function rounds up to avoid truncating the lowest bits.
To save the integer part, the number with the largest integer part will
be the return type.

A.2.2 Multiplication between two signed fixed point numbers

Signature

```
static inline s32 sq_mul(s32 a, s32
    a_scale_factor, s32 b, s32 b_scale_factor)
```

Parameters

```
s32 a
```

Fixed point number.

```
s32 a_scale_factor
```

Corresponding scale factor for *a*.

```
s32 b
```

Fixed point number.

```
s32 b_scale_factor
```

Corresponding scale factor for *b*.

Return Value

Returns a fixed point number scaled by the largest of the two scale factors.

Synopsis

This function returns the product of two signed fixed point numbers.
The function rounds up to avoid truncating the lowest bits.
To save the integer part, the number with the largest integer part will be the return type.

A.2.3 Division between two unsigned fixed point numbers

Signature

```
static inline u32 uq_div(u32 a, u32
    a_scale_factor, u32 b, u32 b_scale_factor)
```

Parameters

`u32 a`

Fixed point number used as numerator.

`u32 a_scale_factor`

Corresponding scale factor for *a*.

`u32 b`

Fixed point number used as denominator.

`u32 b_scale_factor`

Corresponding scale factor for *b*.

Return Value

Returns a fixed point number scaled by the largest of the two scale factors.

Synopsis

This function returns the quotient of two unsigned fixed point numbers.
To save the integer part, the number with the largest integer part will
be the return type.

A.2.4 Division between two unsigned fixed point numbers

Signature

```
static inline s32 sq_div(s32 a, s32
    a_scale_factor, s32 b, s32 b_scale_factor)
```

Parameters

```
s32 a
```

Fixed point number used as numerator.

```
s32 a_scale_factor
```

Corresponding scale factor for *a*.

```
s32 b
```

Fixed point number used as denominator.

```
s32 b_scale_factor
```

Corresponding scale factor for *b*.

Return Value

Returns a fixed point number scaled by the largest of the two scale factors.

Synopsis

This function returns the quotient of two signed fixed point numbers.
To save the integer part, the number with the largest integer part will
be the return type.

A.2.5 Support Macros for Fixed Point Operations

The following sections describe some useful macros utilized by Dalbe.

Conversion Between Integer and Fixed Point

```
int_to_q16(n)
```

Returns a fixed point number scaled with a 16 bit fraction part. Can be either signed or unsigned.

```
int_to_q30(n)
```

Returns a fixed point number scaled with a 30 bit fraction part. Can be either signed or unsigned.

Simplification of *uq_mul* and *sq_mul*

```
mul_uq16_uq16(a, b)
```

Returns the product of two unsigned fixed point numbers with a fraction part of 16 bit. Results is stored in an unsigned fixed point number with a fraction part of 16 bits.

```
mul_uq16_uq30(a, b)
```

Returns the product of two unsigned fixed point numbers. One with a fraction part of 16 bits, the other with a fraction part of 30 bits (the order is not important). Results is stored in an unsigned fixed point number with a fraction part of 16 bits.

```
mul_uq30_uq30(a, b)
```

Returns the product of two unsigned fixed point numbers with a fraction part of 30 bit. Results is stored in an unsigned fixed point number with a fraction part of 30 bits.

```
mul_sq16_sq16(a, b)
```

Returns the product of two signed fixed point numbers with a fraction part of 16 bit. Results is stored in an signed fixed point number with a fraction part of 16 bits.

```
mul_sq16_sq30(a, b)
```

Returns the product of two signed fixed point numbers. One with a fraction part of 16 bits, the other with a fraction part of 30 bits (the order is not important). Results is stored in an signed fixed point number with a fraction part of 16 bits.

```
mul_sq30_sq30(a, b)
```

Returns the product of two signed fixed point numbers with a fraction part of 30 bit. Results is stored in an signed fixed point number with a fraction part of 30 bits.

Simplification of *uq_div* and *sq_div*

```
div_uq16_uq16(a, b)
```

Returns the quotient of two unsigned fixed point numbers with a fraction part of 16 bit (a / b). Results is stored in an unsigned fixed point number with a fraction part of 16 bits.

```
div_uq16_uq30(a, b)
```

Returns the quotient of two unsigned fixed point numbers (a / b). One with a fraction part of 16 bits, the other with a fraction part of 30 bits (the order is not important). Results is stored in an unsigned fixed point number with a fraction part of 16 bits.

```
div_uq30_uq30(a, b)
```

Returns the quotient of two unsigned fixed point numbers with a fraction part of 30 bit (a / b). Results is stored in an unsigned fixed point number with a fraction part of 30 bits.

```
div_sq16_sq16(a, b)
```

Returns the quotient of two signed fixed point numbers with a fraction part of 16 bit (a / b). Results is stored in an signed fixed point number with a fraction part of 16 bits.

```
div_sq16_sq30(a, b)
```

Returns the quotient of two signed fixed point numbers (a / b). One with a fraction part of 16 bits, the other with a fraction part of 30 bits (the order is not important). Results is stored in an signed fixed point number with a fraction part of 16 bits.

```
div_sq30_sq30(a, b)
```

Returns the quotient of two signed fixed point numbers with a fraction part of 30 bit (a / b). Results is stored in an signed fixed point number with a fraction part of 30 bits.

Integer Fraction to Fixed Point

`fract_to_uq16(a, b)`

Returns the quotient of two integer unsigned numbers (a / b). Results is stored in an unsigned fixed point number with a fraction part of 16 bits.

`fract_to_uq30(a, b)`

Returns the quotient of two integer unsigned numbers (a / b). Results is stored in an unsigned fixed point number with a fraction part of 30 bits.

`fract_to_sq16(a, b)`

Returns the quotient of two integer signed numbers (a / b). Results is stored in an signed fixed point number with a fraction part of 16 bits.

`fract_to_sq30(a, b)`

Returns the quotient of two integer unsigned numbers (a / b). Results is stored in an unsigned fixed point number with a fraction part of 30 bits.

A.3 Pluggable Congestion Control Architecture

This section illustrates some of the triggers for the function hooks implemented by DA-LBE. Be aware that these are very simplified, but should be helpful to understand the flow of the architecture which DA-LBE is part of.

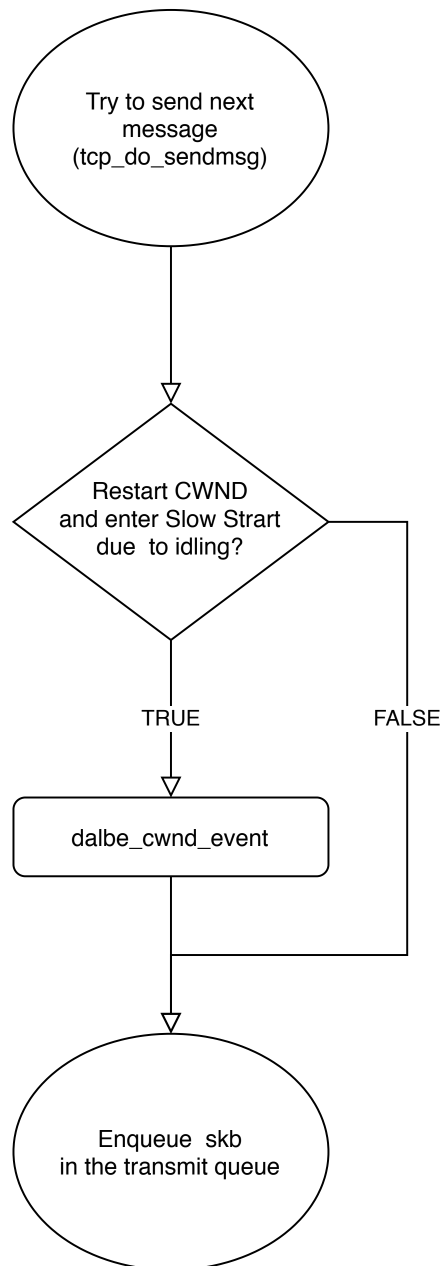


Figure A.1: Flow chart showing one possible scenario where *dalbe_cwnd_event* may be invoked during the TCP connection due to data being sent.

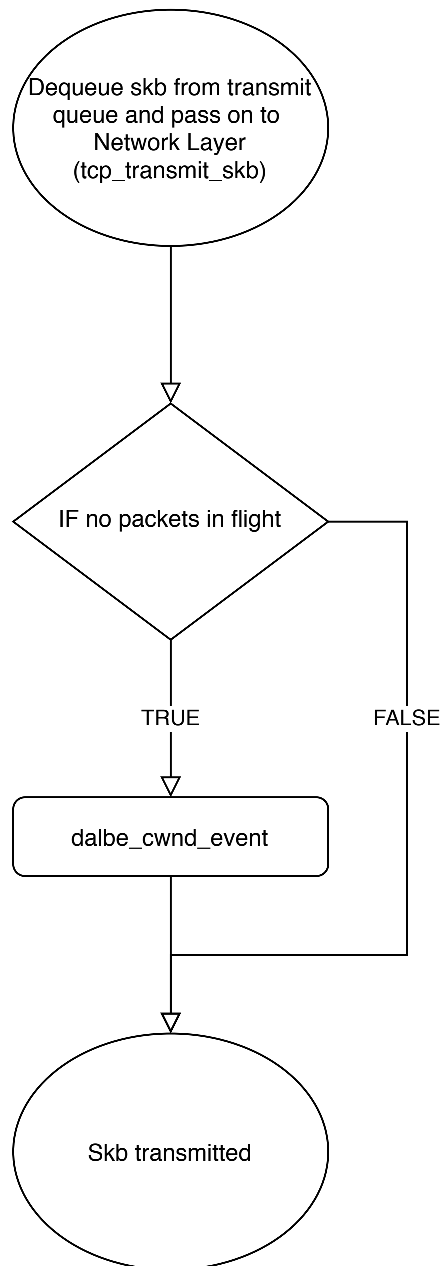


Figure A.2: Flow chart showing another possible scenario where *dalbe_cwnd_event* may be invoked during the TCP connection due to data being sent.

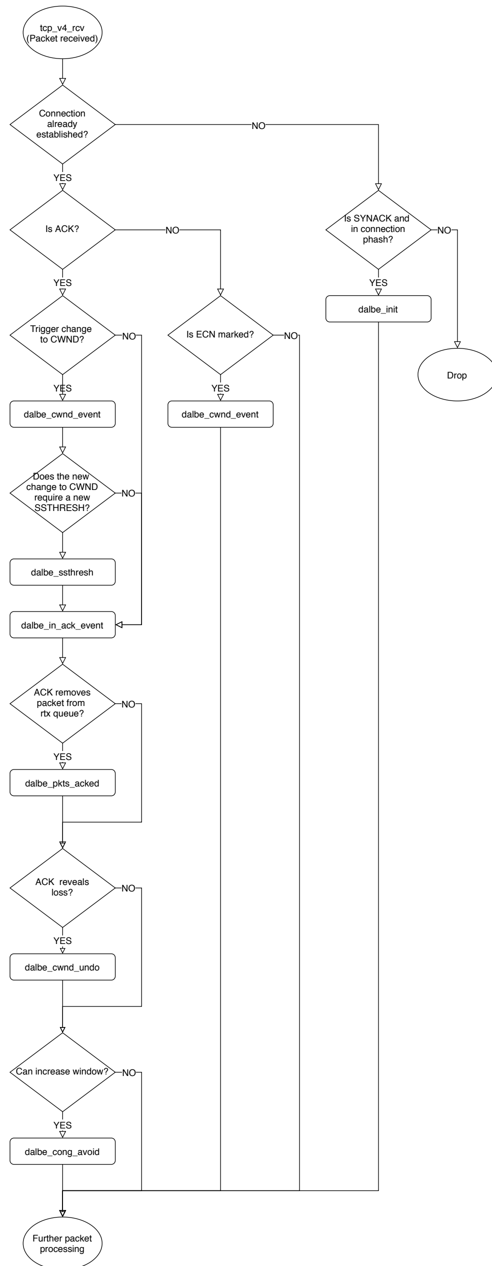


Figure A.3: A comprehensive flowchart showing what DA-LBE meta congestion controller functions may be invoked by the reception of an ACK.

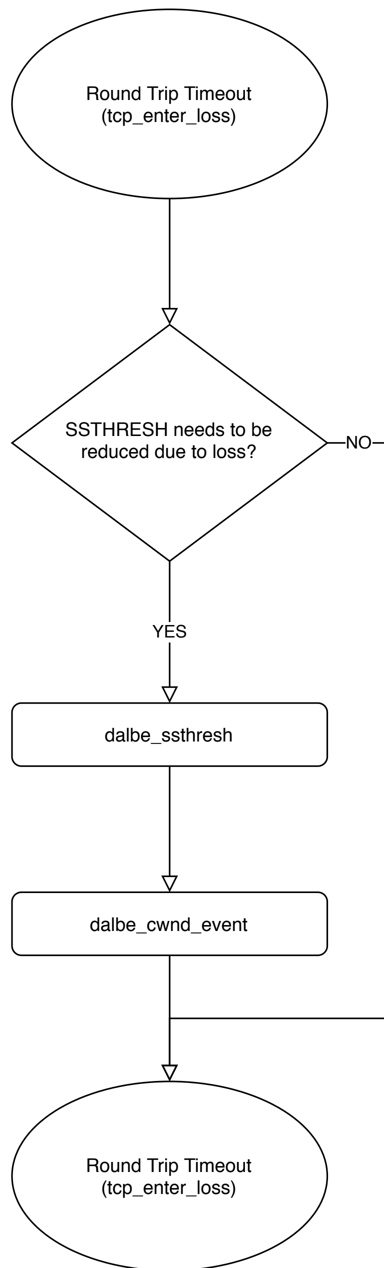


Figure A.4: Flow chart showing a possible scenario where both *dalbe_ssthresh* and *dalbe_cwnd_event* may be invoked during the TCP connection due to an RTO.

Appendix B

Parameters and Socket Options

B.1 Module Parameters

The following section describes the module parameters that Dalbe accepts when the module is loaded using *insmod*.

B.1.1 Module Parameters

```
int default_t_w
```

Default update period for w in milliseconds.

```
int default_t_phi
```

Default update period for ϕ in milliseconds.

```
s32 default_pid_gain_p
```

Proportional gain K_p (see equation 2.8). Defaults to 0.5.

```
s32 default_pid_gain_i
```

Integral gain K_i (see equation 2.8). Defaults to 0.03.

```
s32 default_pid_gain_d
```

Differential gain K_d (see equation 2.8). Defaults to 0.05.

```
int vegas_alpha
```

Corresponds to the α value of Vegas[73, *net/ipv4/tcp_vegas.c*][9]. Make note that if this is not set to the same as Vegas the model based controller for Vegas will not work correctly. It is also critical that $\alpha == \beta$ for Vegas.

B.2 Custom Socket Options

The following section describes the custom socket options available for Dalbe. These can be set by using [50, *setsockopt(2)*] and fetched using [50, *getsockopt(2)*], at the TCP layer.

B.2.1 Custom Socket Options

`DALBE_UNDERLYING_CA`

Used to set the underlying congestion controller. Represented as a string with a maximum length of *TCP_CA_NAME_MAX*[73, *include/net/tcp.h*].

`DALBE_INTERVAL_W`

The interval time for calculating w in milliseconds. Represented as an *int*.

`DALBE_INTERVAL_PHI`

The interval time for calculating ϕ in milliseconds. Represented as an *int*.

`DALBE_DEADLINE`

The target deadline for Dalbe in milliseconds. Represented as an *unsigned long*.

`DALBE_DATA_SIZE`

The size of the data to be transmitted by the means of DA-LBE. Represented as an *u64*.

`DALBE_W_POLICY`

The policy to be used for updating the weight w. Can be either be 0 for PID or 1 for MBC. Any on recognized values will default it to PID. Represented as an *s32*.

`DALBE_PID_GAIN_P`

Proportional gain K_p (see equation 2.8). Represented as an *s32*.

`DALBE_PID_GAIN_I`

The value of K_i (see equation 2.8). Represented as an *s32*.

`DALBE_PID_GAIN_D`

The value of K_d (see equation 2.8). Represented as an *s32*.

Appendix C

Source Code and Raw Data

Source code and raw data can be found at the following locations.

C.1 Source Code

C.1.1 mosaic-students-henning

This repository contains the loadable kernel module with, a patch to the Linux kernel, documentation and some scripts. Available by request [71].

C.1.2 TestBed

This repository contains the scripts and configuration files used for the test bed together with documentation on how to set up and use it. Available by request [72].

C.1.3 tcp-dalbe-test

This repository contains a set of tests used for testing the API and the mathmatical functions defined by Dalbe. Available by request [70].

C.1.4 tcp-dalbe-analysis

This repository contains the scripts for parsing and plotting the results. Available by request [69].

C.2 Raw Data

All data can be available by request.

Appendix D

Documentation

D.1 Meta Congestion Controller Documentation

This section includes the documentation for the meta congestion controller [71].

TCP DALBE

A Meta Congestion Controller forDeadline-Aware Less than Best EffortDelivery in the Linux Operating System.

Authors

- Henning Parratt Tandberg
- Vivian Band
- David Hayes

Related Work

This work is based on the following article:

- D. A. Hayes, D. Ros, A. Petlund and I. Ahmed, "A framework for less than best effort congestion control with soft deadlines," 2017 IFIP Networking Conference (IFIP Networking), Stockholm, 2017, pp. 1-9. doi: 10.23919/IFIPNetworking.2017.8264853

Supported Kernel Version

Linux 5.4

Module Paramenters

The module can be configured to some degree when it is inserted to the kernel. Following are the available module parameters:

Parameter Name	Parameter Type	Description
default_t_w	int	Default update period for w (ms)
default_t_phi	int	Default update period for phi (ms)
default_pid_gain_p	s32	Proportinal gain. Defaults to 0.5
default_pid_gain_i	s32	Integral gain. Defaults to ~0.03
default_pid_gain_d	s32	Differential gain. Defaults to 0.05
vegas_alpha	u32	Alpha = Beta for TCP Vegas

Socket Options

The module comes with a patch to the kernel which creates a hook to set custom socket options at the `SOL_TCP` level. Following are the available socket options:

Option Name	Option Type	Placement	Description
`DALBE_UNDERLYING_CA`	char *	underlying_ca	Underlying congestion controller
`DALBE_INTERVAL_W`	int	t_w	Interval for calculating `w`
`DALBE_INTERVAL_PHI`	int	t_phi	Interval for calculating `phi`
`DALBE_DEADLINE`	unsigned long	target_deadline	Deadline for data delivery
`DALBE_DATA_SIZE`	u64	data_to_send	Amount of data to be sent
`DALBE_W_POLICY`	s32	w_policy	Weight control policy
`DALBE_PID_GAIN_P`	s32	pid_gain_p	PID gain value for `p`
`DALBE_PID_GAIN_I`	s32	pid_gain_i	PID gain value for `i`
`DALBE_PID_GAIN_D`	s32	pid_gain_d	PID gain value for `d`

Installation

The module comes with a patch to the Linux kernel. This patch is located in `./dalbe.patch`, and must be applied to the correct version of the Linux kernel. The following steps show how to apply the patch, install the custom kernel, and load the kernel module:

1. Clone the correct kernel repository to the VM if it is not allready there.

```
git clone https://kernel.googlesource.com/pub/scm/linux/kernel/\
git/davem/net-next && \
git clone git@bitbucket.org:simula-mosaic/mosaic-students-henning.git
```

2. Enter the root directory of the cloned kernel.

```
cd net-next/
```

3. Checkout the correct version/tag.

```
git checkout tags/<TAG_OF_SUPPORTED_KERNEL>
```

4. Apply the patch:

```
git apply <PATH_TCP_DALBE_REPO>/dalbe.patch
```

5. Create a local module config based on the bare minimum of the running system.

```
make localmodconfig
```

6. Install required modules that are not included in the local module config. This is done by either utilizing the config editing tool `menuconfig` or by editing the `.config` file directly.

```
make menuconfig          # Easiest way to add a module.

# OR

vim net-next/.config     # Not as easy...
```

7. Build the kernel with all available threads, as this may take some time.

```
make -j$(nproc)
```

NOTE: Frome here all steps have to be performed as a `sudo` user.

8. Install the modules specified in the config created in step 5.

```
sudo make modules_install
```

9. Install the kernel to the running system.

```
sudo make install
```

10. Install the correct headers

```
sudo make headers_install INSTALL_HDR_PATH=/usr
```

11. Reboot the system.

```
sudo reboot
```

When the system reboots, verify that the kernel was properly installed by running `uname -r` whcih should output `5.4.0+`.

Build, Install and Remove Module

Once the custom kernel has successfully been built and installed the module can be installed:

1. Navigate to the root directory of the module repository.

```
cd <PATH_TO_TCP_DALBE_REPO>
```

2. Build the module:

```
make
# Or if you want debugging enabled:
make debug
```

3. Insert the module into the kernel:

```
sudo insmod dalbe.ko
# Or with parameters:
sudo insmod <PARAMETER_NAME>=<PARAMETER_VALUE> dalbe.ko
```

4. Verify that the modules has been inserted:

```
lsmod
```

The module is now part of the kernel and can be tested.

5. To remove the module from the kernel:

```
sudo rmmod dalbe
```

Load the module at startup

After the module has been built it can be set to load at startup.

NOTE: - This is based on a setup for Ubuntu 18.04, and thus the setup may differ on another system. - All opartions in the following setps require `sudo`.

1. Make the module availabe at boot by adding the name of the module (without `.ko`) to `/etc/modules-load.d/modules.conf`.

```
echo dalbe >> /etc/modules-load.d/modules.conf
```

2. Add the module to `modprobes` database by copying the kernel module object to the networking driver directory of the running kernel.

```
cp dalbe.ko /etc/modules/$(uname -r)/drivers/net/
```

3. Register dependencies for the module.

```
depmod
```

Reboot the system.

4. Verify that the module has been loaded.

```
lsmod | grep dalbe
```

Register Trace Events

If tracing the module comes of interest, its trace events may be registered as followed using `perf`:

Skip step 1 - 4 if perf is allready installed.

1. Enter the directory of the perf tool set in the linux soruce code.
   ```
   cd net-next/tools/perf/
   ```
2. Build `perf`.
   ```
   make
   ```
3. Install `perf`.
   ```
   sudo make install
   ```
4. Reboot to make the installation take proper effect.
   ```
   sudo reboot
   ```

At this point, `perf-probe` can be used to register trace events.

5. List the available trace events for the kernel module, which can be probed by the tracer.
   ```
   perf probe -F -m /lib/modules/$(uname -r)/kernel/drivers/net/dalbe.ko
   ```
6. Register a trace event for the kernel module.
   ```
   perf probe -m /lib/modules/$(uname -r)/kernel/drivers/net/dalbe.ko\
   -a 'dalbe_in_ack_event'

   #Or register all trace events prefixed with dalbe_

   perf probe -m /lib/modules/$(uname -r)/kernel/drivers/net/dalbe.ko\
   -a 'dalbe_*'
   ```
7. At this point the given trace events are registered and can be probed by the tracer.
   ```
   perf record -e 'probe:dalbe_*' -aRg <EXECUTABLE_UTILIZING_DALBE>
   ```

Testing

For testing and analysis please refer to the following repositories:

dalbe-testbed: A testbed desinged for running experiments on *tcp-dalbe* in an emulated environment.

tcp-dalbe-test: Unit tests targeting the API and fixed point operations utilized by *tcp-dalbe*.

tcp-dalbe-analysis: A set of tools for parsing and plotting data produced by *tcp-dalbe* during testing.

Contribution

Recent status and todo-list of the implementation can be found here.

License

GPL-2.0 AND BSD-3-Clause.

D.2 Test Environment Documentation

This section includes the documentation for the test environment [72].

DALBE Test Bed

A testbed for running various experiments on *tcp-dalbe* and *mptcp-dalbe*.

Authors

- **Henning Parratt Tandberg**
- **Mattis Bratland**

Architecture

This script suite is designed for setting up various test-topologies for both a virtual based and hardware based environment.

It runs experiemnts on the given nodes by adding them to the config (./configs/experiemnt), and requires that the node has been added to the ssh config on the system that orchatrastes the experiments.

Platform and Word-Of-Caution

We have only run the following scripts on a **Debian** based platform. More precisely, Ubuntu Server 18.04 LTS. We can not guarantee that the following scripts work on any other given platform.

Requirements

- Python3.8 (pluss dev and venv package)
- Pip3
- Fabric
- pyinstaller
- colorlog
- GNU make.

Installing Python3.8 etc. and pip3

```
sudo apt-get install python3.8 python3.8-dev python3.8-venv python3-pip
```

Installing Python libraies

When the python3.8 etc. has been installed, we advise you use a virtual environment for the scripts. This is to isolate the project configuration from any other global setup and make sure that the correct libraries and versions are used.

1. Enter the root testbed directory.

```
cd testbed/
```

2. Set up virtual environment.

```
python3.8 -m venv testbed-venv
```

3. Activate the virtual environment.

```
source testbed-venv/bin/activate
```

4. Install the required libraries using pip.

```
pip install -r requirements.txt
```

Building and Installing

The scripts can be installed as binaries after intallation of all requiremts are complete.

1. Enter the root testbed directory.

```
cd testbed/
```

2. Activate the virtual environment if it's not already active.

```
source testbed-venv/bin/activate
```

3. Build binaries from the scripts.

```
make # or make all
```

4. Install the binaries on the system.

```
sudo make install
```

At this point you can deactivate the virtual environment and run the scripts as if they are bash commands. If you do not wish to install every script, you can simply enter the directory of the script you wish to install and proceede from step 3. and 4.

Cleanup and Uninstalling

All scripts can be uninstalled by doing the following:

```
sudo make uninstall
```

If you wish to remove build files and binaries from the testbed direcotry do the folling:

```
make clean
```

Test Bed Setup

The following sections describe how the testbed should/could be set up.

SSH

We have set up all the nodes in the testbed with an ssh key pair and created an entry for them in our local ssh config. This makes for an easy, secure, login.

And - this is what is expected by the provided scripts that attempt to connect to the nodes in the test bed using **fabric**.

Also worth mentioning that the **edge nodes**, which in our testbed have less computing power than the **router**, may have their performance degraded by running heavy security software. If this is the case, they should only be accessible via the **router** to avoid them from being directly exposed to the Internet.

Following is an example of an ssh config:

```
Host router
    User dalbe
    Hostname dalbe.router.hostname / address
    Port 22
    IdentitiesOnly yes
    IdentityFile ~/.ssh/dalbe-router/id_rsa

Host node-a-0
    User dalbe
    Hostname 172.16.0.2
    Port 22
    IdentitiesOnly yes
    IdentityFile ~/.ssh/node-a0/id_rsa
    ProxyCommand ssh hw-router nc %h %p

Host node-b-0
    User dalbe
    Hostname 172.16.0.3
    Port 22
```

```
    IdentitiesOnly yes
    IdentityFile ~/.ssh/node-b0/id_rsa
    ProxyCommand ssh hw-router nc %h %p

Host node-a-1
    User dalbe
    Hostname 172.16.0.4
    Port 22
    IdentitiesOnly yes
    IdentityFile ~/.ssh/node-a1/id_rsa
    ProxyCommand ssh hw-router nc %h %p

Host node-b-1
    User dalbe
    Hostname 172.16.0.5
    Port 22
    IdentitiesOnly yes
    IdentityFile ~/.ssh/node-b1/id_rsa
    ProxyCommand ssh hw-router nc %h %p
```

Network Interfaces

Each node in the testbed needs to be configured with the correct interface config. This can easily be done by editing */etc/network/interfaces* on each node. A set of pre-written interface configs can be found in ./network/interfaces.

NOTE - If the nodes are running **Ubnutu 18.04** or later, netplan must be deactivated by installing **ifupdown**:

```
sudo apt install ifupdown
```

Visudo

Some commands may require sudo permissions. We suggest, for scurity reasons, that only the specific commands run are given access without prompting for password. Following is an example of how this can be done using `visudo`:

```
# Cmnd alias specification
Cmnd_Alias ETHTOOL_PAUSE = /sbin/ethtool --pause *
Cmnd_Alias ETHTOOL_COALESCE = /sbin/ethtool --coalesce *
Cmnd_Alias ETHTOOL_OFFLOAD = /sbin/ethtool --offload *
Cmnd_Alias TCP_NO_METRICS_SAVE = /sbin/sysctl \_
-w net.ipv4.tcp_no_metrics_save\=*
```

```
Cmnd_Alias IP_TCP_METRICS_FLUSH = /sbin/ip tcp_metrics flush
Cmnd_Alias TCPDUMP = /usr/sbin/tcpdump
Cmnd_Alias DALBE_TRANSFER = /usr/bin/dalbe-transfer

# includedir /etc/sudoers.d
dalbe ALL=(ALL) NOPASSWD:ETHTOOL_PAUSE
dalbe ALL=(ALL) NOPASSWD:ETHTOOL_COALESCE
dalbe ALL=(ALL) NOPASSWD:ETHTOOL_OFFLOAD
dalbe ALL=(ALL) NOPASSWD:TCP_NO_METRICS_SAVE
dalbe ALL=(ALL) NOPASSWD:IP_TCP_METRICS_FLUSH
dalbe ALL=(ALL) NOPASSWD:TCPDUMP
dalbe ALL=(ALL) NOPASSWD:DALBE_TRANSFER
```

CORE

CORE Network Emulator is intented to run on certain notes to emulate a specific type of network to provide some type of specific testing scenario. Where this may be needed, we suggest this be installed manually. The *core-session-xml* script has been created to easily run specific configurations on a given node.

The installiation guides for CORE can be found here.

Software

The test bed comes with a variety of scripts and software. Their usage is documented in the following sections.

CORE Session XML

A script that passes an XML config to a `core-daemon` using gRPC. The daemon initiates a network emulation (referred to as `core-session`) based on the XML-file it recieves.

The XML configs

The XML configs are genereated with the `core-gui`, and should be placeed in the `./configs/core/` directory.

Usage

```
core-session-xml [-h] --xml_config XML_CONFIG
    -h, --help           Show this help  message and exit
    --config CONFIG      XML config to open
```

Important Note

The script is meant to run on the same host as where the `core-daemon` is running. Before running the script, the `core-cleanup` should be run to make sure that there are no other conflicting core-sesions running.

Experiment Script

A script for running experiments on a testbed.

```
experiment [-h] --config_path CONFIG_PATH

optional arguments:
  -h, --help            show this help message and exit
  --config_path CONFIG_PATH
                        Path to experiment config file
```

Config

Each experiment is configured with a `JSON` file, which is structured as follows:

```
{
    "name": "<NAME_OF_EXPERIMENT>",
    "duration": <DURATION_OF_EXPERIMENT_IN_SECONDS>,
    "nodes|router" : [
        "<NODE_NAME>" : {
            "ssh" : {
                "config_entry" : "<ENTRY_FOR_NODE_IN_SSH_CONFIG>"
            },
            "network": {
                "core_config": "<ABSOLUTE_PATH_TO_XML_CONFIG_PATH>",
                "interfaces": ["<LIST_OF_INTERFACES_TO_USE>"]
            },
            "commands" : {
                {
                    "time" : <TIME_TO_START_COMMAND_IN_SECONDS>
                    "command": "<NAME_OF_COMMNAD>",
                    "parameters": ["<LIST_OF_PARAMETERS_FOR_COMMAND>"],
```

```
                    "detached": <true|false>,
                    "sudo": <true|false>
                },
            }
        }
    ]
}
```

Transfer Script

A script that starts either a server or a cliend, and has the posibility to use the MPDALBE/DALBE meta congestion controller (for TCP) and socket options.

Requirements

To use the MPDALBE/DALBE socket options, the correct kernel must be installed.

Usage

```
usage: main.py [-h] --config CONFIG

optional arguments:
  -h, --help       show this help message and exit
  --config CONFIG  Config file to parse
```

Config

```
{
    "client" or "server" : {
        "host" : <SERVER_ADDRESS>,
        "port" : <SERVER_PORT> (DEFAULT: 15000),
        "data_size" : <AMOUNT_OF_DATA_TO_SEND>,
        "tcp_ca" : <NAME_OF_CONGESTION_ALGORITHM> (OPTIONAL),
        "dalbe_options" (OPTIONAL) : {
            <DALBE_OPTIONS>,
        }
    }
}
```

Config in Detail

By spesicying “client” the transferscript will start up in client mode, and will attempt to connect to the server address. If “server” is specified, the script will accept connections on the address-port. **NOTE** - the server must be started before the client starts.

The “data_size” field specifies, in bytes, how much data the client will attempt to transfer to the server. The server will ignore this field from the config.

If the “tcp_ca” field is set, the client or server will try to set the congestion algorithm for the TCP connection. Else if the “dalbe_options” has been set, congestion algorithm will be set to “dalbe”, Else, it will use the default congestion controller set by the running kernel.

DALBE Config

```
"dalbe_options" (OPTIONAL) : {
    "underlying_ca" : <UNDERLYING_CONGESTION_ALGORITHM> (DEFAULT "vegas")
    "interval_w" : <UPDATE_INTERVAL_W_...> (DEFAULT: 13000),
    "interval_phi" : <UPDATE_INTERVAL_PHI_...> (DEFAULT: 13000),
    "target_deadline" : <TARGET_DEADLINE_...> (DEFAULT: 10),
    "data_size" : <DATA_SIZE> (DEFAULT: 1000000),
    "w_policy" : <WEIGHT_CALCULATION_POLICY> (DEFAULT: "MBC"),
    "pid_gain_p" : <PID_GAIN_PARAMETER_P> (DEFAULT: 3),
    "pid_gain_i" : <PID_GAIN_PARAMETER_I> (DEFAULT: 1),
    "pid_gain_d" : <PID_GAIN_PARAMETER_D> (DEFAULT: 2)
}
```

More datail about these parameters can be found here.

Shortcomings of DALBE Transfer Script

- There is at the moment no correlation between the “data_size” field in the “dalbe_options”-object and the one in the “server”-/“client”-object, so the user has to make sure these are the same.
- The data size must be set for the server even though it is ignored.

Transfer v2

A client and server for DA-LBE transfers, written in C++.

Server Usage

The server is just a simple TCP listening server, which will receive all data from the connected clinet, followed by terminating.

```
dalbe-server -p <PORT>
```

Client Usage

The client connects to a server and sends the specified amount of bytes.

```
dalbe-client -a <ADDRESS> -p <PORT> [DALBE_OPTIONS]
DALBE OPTIONS:
-b <UNDERLYING_CA>    (string)          The underlying CA.
-c <INTERVAL_W>       (unsigned long)   The update interval for w.
-d <INTERVAL_PHI>     (unsigned long)   The update interval for phi.
-e <TARGET_DEADLINE>  (unsigned long)   The specified soft deadline.
-f <DATA_SIZE>        (unsigned long)   The amount of data to send.
-g <W_POLICY>         (0=MBC, 1=PID)    Weight policy to use.
-i <PID_GAIN_P>       (float)           Proportional gain.
-j <PID_GAIN_I>       (float)           Integral gain.
-k <PID_GAIN_D>       (float)           Differential gain.
-t <ALT_TCP_CONG>     (string)          Alternative CA.

The option -t allows the client to use an alternative congestion
contrller such as TCP-Cubic, TCP-Vegas, etc., and when used all
DA-LBE related options are ignored.
```

CPU Load and Memory Monitor

Software that monitors the average CPU load and memory usage on the system, by sampling at random periods.

Usage

The monitor is meant to run in the background, and generates a CSV of the CPU load and memory usage.

```
monitor <OUT_FILE>
```

Additional Scripts

In addition to the software presented above, some additional “simple” scripts are availabe, and provide the followng:

network/network-up.sh

Meant for the **router node**, when run it sets up a dumbell topology for experimentation. It uses Intermediate Function Blocks (IFB) which emulates network traits using Traffic Control queuing disciplines. For it to work, the script must be modified to target the correct Network Interfaces. See the `EASTBOUND` and `WESTBOUND` variables at the beginning of the script.

trace/dalbe_ftrace.sh

This script provides an example of how `ftrace` can be configured to capture the absolute time used inside certain functions of the meta congestion module.

trace/dalbe_perf_stat.sh

This script provides an example of how `perf` can be used to create a statistical function frequency profile of an executable which utilizes the meta congestion module.

D.3 Unit Tests and Benchmarks Documentation

This section includes the documentation for the unit tests and benchmarks [70].

TCP DA-LBE Unit Tests and Benchmarks

A set of unit tests for TCP DALBE specific code. It currently consits of:

- Tests for socket API
- Tests for fixed point calculations and functions that involve fixed point maths.

Requirements

- GCC
- CMake >= 5.10

Setup

1) Clone repository and change to the directory.

```
git clone https://henningtandberg@bitbucket.org/henningtandberg/\
tcp-dalbe-test.git && cd tcp-dalbe-test
```

2) Create build directory and rund CMake.

```
mkdir build && cd build && cmake ..
```

3) Build the code.

```
make
```

This should generate all the needed exectuables, but **note** that builing the API tests requires the system to have built and loaded the meta congestion module. If only test tests that do not target the meta congestion module directly are to be built and run just enter the directory of the specific test and run `make`.

Running the exectuables

For running the **Socket API Tests**:

```
./api-test/APITests
```

For running the **Fixed Point Tests**:

```
./fixed-point-test/FixedPointTests
```

For running the **Fixed Point Benchmarks**:

```
./fixed-point-Bench/FixedPointBench
```

Author

- Henning Parratt Tandberg