Libdalbe

A library for developing Deadline-Aware Less-than Best Effort transport services

Hugo Matthijs Harstad Wallenburg

Thesis submitted for the degree of
Master in Programming and Networks
(Faculty of mathematics and natural sciences)
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2018

Libdalbe

A library for developing Deadline-Aware Less-than Best Effort transport services

Hugo Matthijs Harstad Wallenburg

Libdalbe

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

Printed: Reprosentralen, University of Oslo

Acknowledgements

I would first and foremost like to extend my sincere gratitude to my supervisors, David Hayes, David Ros, and Andreas Petlund, for their immense help and guidance throughout this undertaking. Also deserving of thanks are my friends from the Simula Master's lab, all of us toiling on our theses, my family and friends who haven't the faintest clue what "an Internet" is but still helped me in my writing, and my roommates who had to endure me coming home late every night. Thank you all.

Lastly, I absolutely could not have finished this project without ridiculous amounts of coffee and, of course, Vim.

Abstract

Some data transfers on the Internet today don't need to complete their transmission as early as possible, yet they may still benefit from a loose guarantee of timeliness. Examples include online backups, non-critical system updates, and large file downloads. Such transfers can make use of a Deadline Aware Less-than Best Effort (DA-LBE) transport to prevent degrading the quality of service for competing traffic while also upholding a loose notion of timeliness. In this thesis we present Libdalbe, a library that provides this functionality in an accessible and robust manner.

Libdalbe facilitates the development of *metacontrollers*, which adapt an underlying TCP congestion control in the Linux kernel. The metacontrollers use various congestion event statistics to calculate appropriate control parameters for the DA-LBE kernel component created by Lars Erik Storbukås. With Libdalbe we provide two example metacontrollers which use a Model-Based Controller to adapt TCP Cubic and TCP Vegas respectively.

We tested Libdalbe and the metacontrollers both on an emulated network and on the Internet to verify that our solutions uphold all qualities required for DA-LBE service. Through a quantitative evaluation we found that our metacontrollers succeed in attaining all behaviors associated with DA-LBE service on the Internet, but our metacontroller for turning TCP Vegas into a DA-LBE transport fails to meet the timeliness criterion on heavily congested network.

Contents

List of Figures

List of Tables

Acronyms

ACK Acknowledgement

AIMD Additive Increase/Multiplicative Decrease

API Application Programming Interface

AQM Active Queue Management

BBR Congestion-Based Congestion Control

BDP Bandwidth Delay Product

BE Best Effort

BIC Binary Increase Congestion control

BaseRTT Base RTT

CC Congestion Control

CPU Central Processing Unit

Cubic TCP Cubic

D-ITG Distributed Internet Traffic Generator

DA-LBE Deadline Aware Less-than Best Effort

DHCP Dynamic Host Configuration Protocol

DupACK Duplicate Acknowledgement

ECN Explicit Congestion Notification

EWMA Exponentially Weighted Moving Average

FIFO First In First Out

FLOWER Fuzzy Lower-than-Best-Effort Transport Protocol

FTP File Transfer Protocol

HFSC Hierarchical Fair Service Curve

HTB Hierarchical Token Bucket

HTTP Hypertext Transfer Protocol

ICMP Internet Control Message Protocol

IETF Internet Engineering Task Force

IFB Intermediate Functional Block

IPC Inter-Process Communication

IP Internet Protocol

IQR Inter-Quartile Range

ISOC Internet Society

ISP Internet Service Provider

LBE Less-than Best Effort

LEDBAT Low Extra Delay Background Transport

MBC Model-Based Controller

MSS Maximum Segment Size

MTU Maximum Transmission Unit

NEAT New, Evolutive API and Transport-Layer Architecture

NTP Network Time Protocol

NUM Network Utility Maximization

OWD One-Way Delay

PID Proportional-Integral-Derivative

POSIX Portable Operating System Interface

RED Random Early Detection

RFC Request For Comments

RTT Round Trip Time

Reno TCP Reno

SMTP Simple Mail Transfer Protocol

TBF Token Bucket Filter

TCP/IP Internet Protocol Suite

TCP Transmission Control Protocol

TC Traffic Control

TFTP Trivial File Transfer Protocol

TTL Time To Live

Tahoe TCP Tahoe

UDP User Datagram Protocol

VPN Virtual Private Network

Vegas TCP Vegas

WAN Wide Area Network

WWW World Wide Web

`cwnd` Congestion Window

`rwnd` Receiver Window

sACK Selective Acknowledgement

`ssthresh` Slow Start Threshold

Chapter 1

Introduction

A lot of data being transferred on the Internet can be categorized as *quality constrained*, that is transfers for which rapid and reliable delivery are considered valuable traits. Content delivery services where the end user can immediately notice latency or bandwidth problems, such as media streaming or web browsing, require these qualities. File sharing applications may consider all user-generated traffic quality-constrained as the timeliness of data delivery is easily discernible to the users.

This is where *Less-than Best Effort* (LBE) services, also called *scavenger services*, shine: Such services give precedence to other traffic by measuring the impact of their own traffic upon the network and limiting their own sending rate accordingly. This naturally results in lower expected sending rates, meaning they are unsuited for applications that require speedy or consistent timely delivery. Conceivable use cases for such LBE services include online backups, non-critical system updates, and large file downloads. The two latter cases have already seen LBE services being used, with Apple using this kind of service for system updates [2], and BitTorrent using an LBE service for massive peer-to-peer data transfers [66].

Some of these examples include services where one might want to favor user traffic, but still be able to guarantee timeliness within reason. Online backup services may prefer to let the user have network priority so that they perceive the backup service to not be intruding upon their usage, but concurrently the very core of their service requires timeliness lest the user's backups run stale. An operating system manufacturer, e.g. Apple, wishing to maintain software quality through frequent updates while not incurring traffic interruptions upon their users, may want to push updates to their users using LBE services while also setting a deadline for those updates to finish in order to maintain a level of quality in their products.

This type of service requires a method of transfer that includes both the concept of transferring data in a LBE manner as well as being able to enforce a deadline. Meeting both of these requirements necessitates using a mode of transport, called *Deadline Aware Less-than Best Effort* (DA-LBE), that can scale its own aggression depending on the sending rate required to meet the deadline while also measuring its own impact on the network and scaling back when aggression is not required. To the best of our knowledge this kind of transport has not yet been implemented.

In this thesis we introduce Libdalbe, the DA-LBE library: A framework for providing this set of functionalities to application developers. Libdalbe allows for developers to create their own DA-LBE controllers that modify an existing TCP in the kernel, called *metacontrollers*. The metacontrollers are given statistics about the underlying congestion control and a DA-LBE kernel component, to allow for estimating network congestion and modeling one's own impact on the network. The DA-LBE transfers provided with Libdalbe are implemented as

custom metacontrollers themselves, serving as working and tested example metacontrollers.

The basis for this framework is a set of modifications to the Linux kernel, the DA-LBE kernel component, provided by Lars Erik Storbukås [70]. Our library and the DA-LBE kernel component make up the DA-LBE framework. His work provides an interface into the statistics and inner workings of data transfers in the kernel, allowing both for gathering of data and manipulation of the data flow during operation. The gathering and manipulation of this data is for Linux-based operating systems protected by the system kernel, making it cumbersome to access them. Libdalbe facilitates the use of these kernel modifications from user space, allowing developers easier access.

Libdalbe was supposed to be included in the NEAT project [5], but the implementation was not ready in time. New, Evolutive API and Transport-Layer Architecture (NEAT) is a *transport system* that seeks to allow Internet applications to, instead of specifying a protocol to use, specify a type of service. The NEAT system would then choose from an available set of protocols and choose the best suited one for the requested service, based on hardware, network conditions, and local policy. Libdalbe would have fit into a category of services that could provide DA-LBE or just LBE service, and is structured in such a way as to allow the easy control of many types of protocols.

1.1 Problem Statement

Large data transfers do not necessarily require the type of service offered by Transmission Control Protocol (TCP), namely the guarantee that data will be transmitted as quickly as possible given the network conditions. Applications that do not require these qualities, but still choose to use standard TCP for data transfers, may degrade the service offered to other services that might have a more pressing need for timely service. These large data transfers can, if they do not require timeliness guarantees, employ an LBE transfer to limit their impact on other traffic. However, many require some notion of timeliness, a quality that could be provided by a Deadline Aware Less-than Best Effort (DA-LBE) service. In this thesis we aim to provide this functionality in an accessible and robust manner.

1.2 Goals of this Thesis

In this thesis we aim to achieve the following:

1. Develop functionality that allows for the implementation of custom DA-LBE metacontrollers that access and act upon an underlying TCP directly in the kernel using kernel modifications by Lars Erik Storbukås [70].

2. Develop two DA-LBE metacontrollers for TCP Cubic and TCP Vegas, available as default metacontroller choices in the library.

3. Test the implemented library and metacontrollers to verify their performance in real environments.

1.3 Methodology

Our methodology for implementing the functionality described in section 1.1 was as follows:

1. We examined the feasibility of several possible approaches to providing an accessible and robust DA-LBE service. We initially imagined our service as an application in the Linux tool-chain that would either read its data from a file or from standard input and transfer this data using some DA-LBE enabled transport. Another approach we considered was creating a DA-LBE *tunnel* which would have involved creating a virtual networking interface, similarly to how VPNs operate, through which new connections would be given DA-LBE properties. We settled on creating a library, offering more control to application programmers in exactly how they want to apply DA-LBE services to each of their distinct connections while remaining accessible.

2. The correct operation of our solutions relies on functionality in the kernel done by Lars Erik Storbukås [70]. In order to first verify that the DA-LBE kernel component in the kernel functioned correctly, we started by creating a minimal version of our library with accompanying simple test application which we tested on virtual machines.

3. A proper test bed was created to further aid in development of our metacontrollers when Libdalbe behavior on our virtual machines was not matching up with our expectations. We spent a considerable amount of time verifying that the mechanisms chosen for emulation were as accurate to a real network as reasonably possible.

4. We focused on implementing one DA-LBE congestion metacontroller at a time and started with the one for TCP Cubic, which required the fewest changes from the theory in the paper by Hayes *et al.* [24]. Each congestion metacontroller was developed using a loop roughly consisting of the following steps:

   (a) Implement new functionality or subroutine.

   (b) Run test on the test bed designed to touch on most areas our DA-LBE metacontroller should affect.

   (c) Identify issues and discover their source, be it Libdalbe, the DA-LBE component, or test misconfiguration.

   (d) Repeat until desired functionality is achieved without errors.

5. For evaluation, perform experiments similar to those of Hayes *et al.* and verify similar (or detect and rationalize or correct dissimilar) behavior in comparison with their simulations.

6. Final evaluation of real world performance is carried out on a real Internet connection with real cross traffic and noise, taking care to run experiments multiple times and rely on the data set as a whole for conclusions.

1.4 Structure

The rest of this thesis is structured as follows:

Chapter 2, Background: In the Background chapter we explain fundamentals of the issues that the work presented in this thesis aims to help solve. We explain the mechanisms that we are implementing, what they do, how they work, and why they work. Subsequently we will provide some context on what kind of situation and system this solution is meant for, an example of recent development in the field, and the thesis on which a lot of this work is based.

Chapter 3, Libdalbe: This chapter will present the library that we are implementing. We will cover choices made in the design of the library, what functionality is available, and how custom meta-protocols are handled. How the library is interfacing with the kernel component is explained in detail, with a section on changes that needed to be made for the proper performance of our library. Lastly, we will point out some shortcomings in our final product and how changes could be made to remedy these issues.

Chapter 4, Testing Setup: In chapter 4 we explain how we will verify that our implementation works as specified in the background theory in chapter 2 and in the application description in chapter 3. We go over setting up a highly configurable test bed that can emulate a real network topology with reasonable accuracy, explaining the choices behind setup specifics and possible pitfalls. We describe the setup for testing the actual real world performance of our solution on the Internet. To facilitate the running and analyzing of our experiments we developed a set of tools based on open technologies which is described in the last sections of this chapter.

Chapter 5, Results and Analysis: The Results and Analysis chapter describes the experiments that we performed to verify the functionality of our library along with the results of said experiments and analysis of the data they show. We will examine the differences in performance between the local test bed setup and the Internet setup and analyze how their idiosyncrasies affect the behavior of our library. The analysis is backed by graphs of both the throughput of our implementations and detailed output from our testing application for each test. We verify that our library performs as specified through the use of a fairness measure and transmission completion times.

Chapter 6, Conclusion: For the Conclusion chapter we will quickly summarize this thesis as a whole, reiterate our findings from chapter 5, and draw final conclusions from our results. We will relate our results to the problem statement from section 1.1. Lastly is a section on future work; how our library can be improved, what other experiments can be run, and finally some ideas for further research on topics touched, or skipped, in this thesis.

Chapter 2

Background

This chapter will introduce the basic concepts of how computers communicate on the Internet, how networking equipment remains functional under heavy load, and finally how we can design and implement a type of data transport that gives way to competitors while remaining capable of competing in order to reach a deadline. We will also touch on previous work that enables the contributions made in this thesis as well as give examples of recent similar efforts.

2.1 Transmission of Data on the Internet

Modern developers need not worry much about the intricacies of Internet communication when developing their applications; as long as their data arrive intact and in a timely manner, how it is transferred does not matter. This notion of sending data from an application to be received by an application at the other end, e.g. a web browser sending a request to a server for a web page, can be envisioned as a *layer* of communication. In this multi-layered protocol setup, the developer might only need to know to send a HTTP request, and the HTTP implementation knows which lower-layer protocol to employ to perform its request.

2.1.1 Separation of Concerns

The separation of functionality into different layers makes it so that the implementation of the functionality within a layer can stay minimal; the layer does not need to interface with functionality that is far removed from its own and can thus focus on simplifying its interface. An application needs not implement its own protocols to transfer data over Wi-Fi or physical wires as this mechanism is hidden behind several layers of abstraction, requiring the application only to communicate with the layer directly underneath it for data transport. For the Internet, these layers are defined in RFC 1122 [47] and RFC 1123 [48] as the Internet Protocol Suite or, more commonly known due to the two main protocols described therein, TCP/IP.

The communication layers, as described in RFC 1122 [47] and RFC 1123 [48] are shown in figure 2.1. The two *stacks* on each side of the figure encompass the protocol suite implemented on the host denoted by the surrounding grey box. Hosts communicate, shown by the dashed horizontal arrows in figure 2.1, by passing data down through their own network stack until the data reaches the physical medium through which it is sent to another host, or all hosts, connected to the same medium. Networking equipment along the route between the communicating hosts implements parts of the protocol stack needed to forward the data to the correct receiver.

From the bottom up, the layers are:

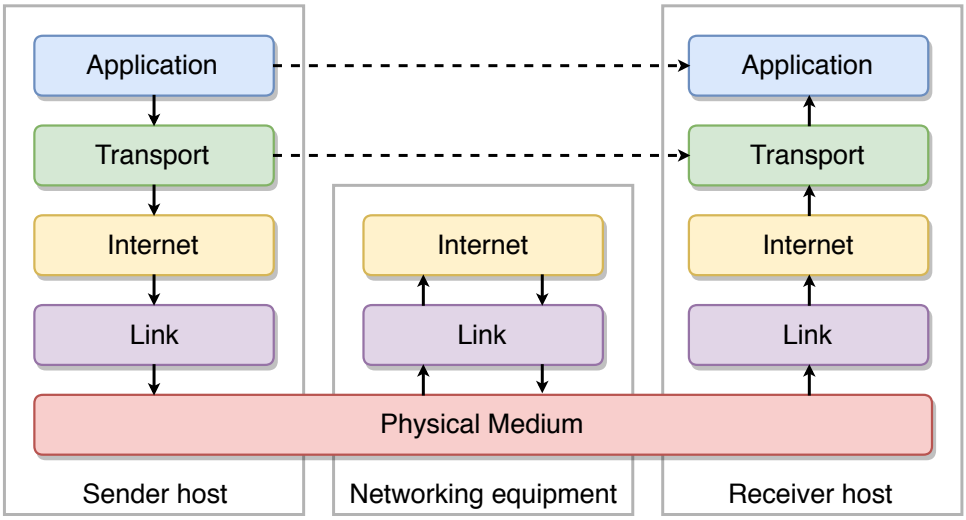


Figure 2.1: The TCP/IP Network Stack

- The **Physical layer** is the actual physical medium through which raw data propagates. Ethernet [25] and Wi-Fi [26] are examples of technologies for communicating through copper (or fiber) wires and the radio spectrum, respectively.

- The **Link layer** is responsible for ensuring communication between hosts connected to the same physical medium. This includes, for example, the protocol determining how two hosts communicate and how to ascertain for which host a data packet, an *Ethernet frame* in the case of Ethernet, is destined.

- The **Internet layer** enables inter-network communication between hosts. The primary protocol of this layer is the Internet Protocol (IP), which describes the transmission of addressable datagrams between hosts on separate networks. IP is connectionless and provides no guarantees as to the delivering of packets. As a result, further protocols are required to ensure the correct delivery of data.

- The **Transport layer** provides data transport between processes by addressing ports, tied to applications, on the receiving host. User Datagram Protocol (UDP) is a datagram service acting as a simple wrapper around IP, supplying only data checksumming and port-addressing as extra features. Many custom protocols are implemented on top of UDP as it functions almost identically to IP and is directly available from the Application layer, making it easy for developers to experiment with. On the other hand, TCP, further described in section 2.2, transports a reliable byte-stream of data, shielding the user from the duplication, damage, reordering, and dropping of packets that affect IP.

- The **Application layer** defines protocols for applications to communicate between each other. RFC 1123 describes the protocols Telnet, FTP, TFTP, and SMTP. HTTP, the defining protocol of the World Wide Web used for fetching and submitting web content, belongs in the Application layer.

Each protocol in the TCP/IP network stack reserves some of the transmission bandwidth for metadata such as addresses, checksums, and payload length by encapsulating the payload data in a larger construct specific to that protocol. Ethernet on the Physical layer sends payloads by way of Ethernet frames, IP wraps its payload in an IP packet, and TCP sends

data using TCP segments. Consequently, the actual data transmitted in the physical medium is larger than the data transmission initiated on the Application layer.

2.1.2 IP

Transmission of data between hosts on the Internet is done through the Internet Protocol [61] in the Internet layer. IP is responsible for routing data through potentially multiple networks to the receiving host based on its IP address. No active error mitigation is performed. IP packets may contain corrupt data, they may arrive twice, or in the wrong order, or not at all. A checksum field guards against errors in the packet header, the metadata, — errors which could risk the packet arriving at the wrong destination — but further errors must be caught by other means such as employing a higher-layer protocol like TCP. This type of service is termed *Best Effort* (BE); IP will make an effort to deliver packets, but makes no other assurances.

Sources of packet loss other than being discarded as a result of corrupt metadata depend on the networking equipment connecting the two hosts. Routers forward IP packets between networks and use shared knowledge about the network state to determine the correct destination network for each packet. Each IP packet carries with it a Time To Live (TTL) counter that limits the number of times it can be forwarded to a new network[1]; packets that have been in the network long enough for their data to be considered stale are dropped, a situation which may occur if changing network conditions causes routers to forward packets in a loop. The most common cause of lost packets in typical operation however is congestion in the network, which occurs when shared resources are experiencing heavy load.

2.1.3 Shared Resources

Communication within a network relies on shared infrastructure in that network. A typical home networking solution might have a single Wi-Fi enabled router connecting several family members and their devices to the Internet. In this scenario all Wi-Fi devices share the medium in which they communicate, the frequency band the router operates on, but more importantly for our work, they must all share the one router and its connection to the Internet. Our hypothetical family most likely is not connected to the Internet with the same link capacity as the one supported locally on their own network [28] and so their devices must compete for the limited bandwidth available to them; they share a bottleneck. Though the equipment in the Internet supports much higher bandwidth rates than our hypothetical family network, the problem of multiple hosts communicating over one link remains; the outgoing link might be able to transmit at rates of 10 Gbit/s, but if the router is receiving traffic from 11 other links supporting 1 Gbit/s each there is not enough bandwidth for all of them to transmit at once. In both the case of the home network and the core Internet equipment, the routers are congested.

2.1.3.1 Buffers, Queues, and Explicit Congestion Notification

To mitigate the effects of the target link not supporting the needed bandwidth, packet queues are employed. Routers deliver packets to the network interface[2] corresponding to the target network of each packet. The network interfaces themselves have *buffers*, packet storage areas, into which the packets are queued to await transfer onto the physical network. When there are more packets being enqueued than the interface can transfer, its packet queue

[1] Termed *network hops*.

will eventually fill the buffer; when the buffer reaches capacity there is no option but to drop packets. Historically, routers would drop the incoming packet [51, p. 3], known as *tail drop*, and some routers would drop the oldest packet in the queue, *drop from front* [38], but more recent development sees interfaces actively dropping packets from their queues early to maintain network stability [16], [51], [63], [71], a mechanism termed Active Queue Management (AQM).

When AQM solutions are employed there is also the possibility of notifying the sender of congestion early without dropping packets through the use of Explicit Congestion Notification (ECN), a functionality in IP [53]. ECN allows routers to set an ECN mark in the packet to explicitly signal congestion; the receiver of an ECN-marked packet will echo the congestion mark to the other host in the connection pair by setting an ECN mark of its own, resulting in both hosts being notified of congestion early. In section 2.6.3 we describe how we employ ECNs to artificially limit the sending rate of our own application.

2.2 Transmission Control Protocol

With IP offering a Best Effort (BE) service, higher level data transfer protocols must be defined to facilitate reliable communication. TCP [62] is one of these protocols, fitting into the Transport layer of TCP/IP and providing a connection oriented transport between processes running on Internet hosts. TCP transmits a stream of data, a flow, using uniquely identifiable TCP segments that are reassembled on the receiver host to reconstruct the byte-stream that was given to the TCP by the sender side application. For every segment received, TCP will answer with an Acknowledgement (ACK) packet stating which segment it expects next; if the sender receives an ACK that it has already sent it knows that there has been either loss or reordering, and that the missing packet might need to be retransmitted. If a set amount of time passes without an expected ACK being received, the *retransmission timeout*, the packet is considered lost and will be retransmitted; if the packet was simply reordered, the ACK will probably arrive not too long after it was expected. This mechanism of sending an ACK for every segment received can be described as a *self-clocking* signal, an ACK-clock, with new segments being released into the network for every segment that leaves.

There is a real possibility that the recipient of a TCP data stream cannot consume data at the same rate it is received; to counteract this problem, TCP on the receiver side must buffer segments until the recipient application consumes their data. To prevent the receiver running out of buffer space for the incoming byte-stream, the TCP receiver determines the amount of segments it can buffer for the consumer application and advertises the size of their buffer, their Receiver Window (`rwnd`), to the sender TCP on every ACK. The sender TCP uses a *sliding window* marking the range of segments that are in flight and unacknowledged; this window is kept in sync with the advertised `rwnd` from the receiver. This is *flow control*.

Like a TCP receiver may not be able to consume data fast enough, the network may not be able to process packets at the rate they are received. As briefly introduced in section 2.1.3, if a router is connected to two network links supporting different maximum transfer speeds and the faster link is sending packets at maximum capacity to be forwarded to the slower link, the router may need to drop packets as new ones arrive; the network has become *congested*. In the worst case this can lead to significant drops in total network throughput, a *congestion collapse*, as noted by Jacobson [29]. As flow control exists not to overwhelm the receiver, congestion control must be employed to ensure the stability of the whole network.

[2]The physical interface through which a network link connects the router to a network.

2.3 Congestion Control

To prevent incurring unnecessary congestion on the network, TCP relies on Congestion Control (CC) mechanisms that probe the network for available bandwidth and back off upon indication of congestion. TCP must infer the state of congestion based on information such as: Packets being lost [29]; a higher propagation delay signifying building router queues (e.g. Vegas [6]); or ECNs, through which routers can alert end hosts of congestion early.

The first congestion control algorithms that were proposed used packet loss as a signal of congestion, with the following reasoning: " If packet loss is (almost) always due to congestion and if a timeout is (almost) always due to a lost packet, we have a good candidate for the 'network is congested' signal. Particularly since this signal is delivered automatically by all existing networks, without special modification [...]. " ([29])

After Jacobson developed his loss-based congestion control algorithms, other mechanisms have been suggested and implemented, two examples being Vegas and BBR. Vegas [6] uses the Round Trip Time (RTT) to estimate the current sending rate, where the congestion signal is the ratio between actual and estimated sending rate. Additionally, Vegas treats loss as a congestion signal in order to ensure TCP fairness (described in section 2.3.1.4). Vegas is more thoroughly described in section 2.4.2. BBR [8], a congestion control recently developed by researchers at Google, "reacts to actual congestion" by using sending rate and RTT measurements to establish a network model and probing for RTT and bandwidth measurements with the model as a base. Notably, BBR disregards packet loss as a congestion signal and thus runs the risk of sending faster than competing non-BBR TCPs in situations with abundant loss. The dominant congestion control of the future might use hitherto unthought of signals of congestion.

2.3.1 Congestion Control Algorithms

RFC[3] 5681 [54] defines four Congestion Control algorithms that most TCP implementations use today: Slow Start, Congestion Avoidance, Fast Retransmission, and Fast Recovery. The RFC also prohibits any TCP algorithm from pushing packets to the network any faster than these four algorithms would allow; an algorithm exceeding this limit may cause unnecessary network congestion and can, if widely employed, lead to congestion collapse.

2.3.1.1 Slow Start & Congestion Avoidance

Slow Start and Congestion Avoidance together estimate the capacity of the network and attempt to find an equilibrium point, the point at which all senders in the network can transfer data at the same rate. Because TCPs cannot communicate between each other, the algorithms used for congestion control should provably converge on an equilibrium state, given that they are employed on all network hosts. A behavior termed Additive Increase/Multiplicative Decrease (AIMD) describes how a TCP sender increases its sending rate additively, by a constant amount of data, while backing off multiplicatively, by a factor of its current rate, upon receiving a congestion signal. AIMD is proven to converge to a fair allocation of bandwidth for all users [30] and functions as follows.

Congestion window Each TCP connection maintains a Congestion Window (`cwnd`) on the sender side that limits the maximum number of segments a connection may have in flight at

[3] RFCs are publications by the Internet Engineering Task Force (IETF) and the Internet Society (ISOC). They are typically submitted for peer review and some of them become Internet standards.

any one time. TCP will attempt to expand this window in order to probe for more bandwidth, backing off on any signal of congestion so as to not overload the network. Two modes of `cwnd` expansion exist: A mode where the sending rate increases exponentially, called Slow Start, to quickly establish an initial measurement of the maximum capacity, and a mode in which the sending rate increases slowly to carefully check for more bandwidth.

Slow Start Slow start is a mechanism to jump-start the ACK-clock by attempting to quickly reach a point where the clock is self-supporting. The congestion window is first initialized to a small size, typically allowing only 1 [49] or 2 [52] or, more recently, 10 segments [60] in flight at the start of the connection. Subsequently the window is expanded by one segment for each received ACK, thus if there is no loss the congestion window will double in size every RTT; an exponential increase. Slow Start is used during the start of a flow, when the link capacity is unknown, and potentially after a retransmission if the packet loss in Congestion Avoidance mode cause the `cwnd` to drop below Slow Start Threshold (`ssthresh`) meaning the sending rate is far off equilibrium. The Slow Start mechanism is essentially probing for the maximum capacity of the network, more specifically the network path between the two hosts of the connection, by deliberately causing congestion until the momentary capacity is reached. Three events will cause TCP to exit the Slow Start state and enter Congestion Avoidance:

1. **Packet loss** - TCP assumes there is congestion on the network.
2. **`rwnd` limit reached** - The congestion window cannot be extended further, as we cannot overload the receiving host.
3. **`ssthresh` reached** - Upon exceeding the *Slow Start Threshold* (`ssthresh`), the flow must enter Congestion Avoidance mode. `ssthresh` acts as a reasonable upper bound on the currently estimated network performance; it is initially set to a maximum value, "The initial value of ssthresh SHOULD[4] be set arbitrarily high (e.g., to the size of the largest possible advertised window)" ([54]), so as to not have an arbitrary implementation specific setting controlling the probing phase. Upon packet loss, `ssthresh` is set to half the current amount of data in flight, resulting in the threshold being set to the active `cwnd` when the packet loss occurred (one RTT ago, when the window was half as big).

Congestion Avoidance Where as in the Slow Start state the Congestion Window is doubled in size every RTT, in Congestion Avoidance mode the `cwnd` may only increase by one segment every RTT. After exiting Slow Start, the `cwnd` will be set to a value that is a decent estimation of the network capacity, however, network conditions change; the congestion on the network the moment the flow exited Slow Start is not indicative of the level of network congestion for the whole flow. TCP will therefore continually probe the network for more bandwidth at a much slower rate of increase than in Slow Start, and on the first reception of a congestion signal reduce `cwnd` by a constant factor. If, as a result, the new `cwnd` is lower than `ssthresh`, TCP will return to the Slow Start state to quickly get back to the equilibrium.

The first TCP implementation to make use of Slow Start and Congestion Avoidance has since come to be known as TCP Tahoe.

[4] "SHOULD" is an RFC "key word" meaning "that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course." See RFC 2119 [50]. Throughout the explanation of TCP concepts, some of terms outlined in RFC 2119 [50] will be used with the meaning defined therein.

2.3.1.2 Fast Retransmit & Fast Recovery

The basis for the Fast Retransmit and Fast Recovery algorithms is that the receiver end of the TCP connection sends Duplicate Acknowledgement (DupACK) packets for every packet received out of order; packets arriving in the wrong order can be a sign of reordering or duplication of packets on the network, but is usually a signal of a packet having been lost. According to [54], the TCP sender SHOULD use the Fast Retransmit algorithm to detect and repair loss, based on incoming duplicate ACKs: receiving three DupACKs causes the immediate retransmission of the lost segment, without waiting for the retransmission timer to expire.

After having retransmitted the lost segment, the TCP sender enters the Fast Recovery state, in which the TCP will artificially inflate the `cwnd` to allow the continued steady transfer of data. The packets that triggered the DupACKs have left the network and are stored in the receiver's buffer; they are not contributing to network congestion and so the network has room for as many packets as have been correctly received. Inflating the `cwnd` ensures that the sending rate is not impaired and avoids the TCP having to reenter the Slow Start state. After having received an ACK for the segment that was lost, TCP sets $cwnd := ssthresh$, thereby deflating the artificially inflated window, and continues in Congestion Avoidance mode.

This improvement was first implemented in TCP Reno.

2.3.1.3 TCP Cubic

First presented by Ha and Rhee [22] in 2008, TCP Cubic [22], [65] is a congestion control scheme in which the growth of the congestion window is governed by a cubic function in terms of the elapsed time since the last loss event. It is an enhancement of Binary Increase Congestion control (BIC) [80], which uses a similar though more involved growth algorithm, that aims to improve on fairness both in relation to TCP in general and flows with different RTTs.

The congestion avoidance scheme of Cubic is an improvement over BIC in the following two ways: First, Cubic retains the advantages of BIC's `cwnd` growth directly following a loss event; the window is quickly expanded up to the size of the previous equilibrium point, where it stays for a prolonged period of time. Ha and Rhee [22] note that their growth function is easier to analyze than the multi-staged growth function of BIC. Second, Cubic grows its `cwnd` as a function of *real time*, as opposed to BIC which grows its window in terms of the RTT. Ha and Rhee further show that their approach yields better TCP friendliness in both long- and short-RTT networks.

TCP Cubic has been the default congestion control algorithm in Linux for a long time already [65], replacing BIC; being the default congestion control in Linux, it is safe to say that it is widely deployed in the Internet today. In our experiments Cubic is being used as the baseline of TCP performance on the assumption that an arbitrarily chosen competing TCP on the Internet will be Cubic.

2.3.1.4 Fairness

The mechanisms described in section 2.3.1 are all described using the key word *SHOULD*, meaning they are not deemed mandatory. RFC 5681 [54] advises however that "a TCP MUST NOT be more aggressive than [these algorithms] allow". Given that this is the standard for TCP implementations it must be assumed that other hosts on the network are using the mechanisms set forth by the standard; if a new congestion control would allow a higher

sending rate the hosts using this new algorithm would attain an unfair share of network bandwidth, and likely starve competing hosts.

This fairness can be quantified using a fairness measure, such as *Jain's Fairness Index* [31]. In their paper, Jain *et al.* equate fairness in throughput to fairness in `cwnd` size, which relates to the recommendations in section 2.3.1 for TCP implementations to never scale their `cwnd`s past what a baseline TCP would.

Jain's fairness index is defined as follows:

$$\mathbb{J}(x_1, x_2, \ldots, x_n) = \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n \times \sum_{i=1}^{n} {x_i}^2} \tag{2.1}$$

where x denotes the allocation of the given resource for every source n, normalized by the expected fair share. If the resource is not to be shared equally, e.g. one user is allocated 20 % of the resource and another gets 80 %, then the values must be normalized according to what is considered a fair allocation of the resource. When measuring the fairness of greedy TCPs competing against each other no normalization is needed, as they are each expected to achieve an equal share of the resource, or $\frac{1}{n}$. Jain's fairness index gives a value $\frac{1}{n} \leq x \leq 1$, where the resource allocation is fairest when the result is 1 and most unfair when it is $\frac{1}{n}$.

Any two TCPs, regardless of implementation, competing against each other on the same route should ideally result in a fairness index of 1, though the actual situation is more complicated. Ha and Rhee [22] show that five different TCP implementations, including Cubic which they are presenting, show little fairness towards a reference "regular long-term TCP" on a 220 ms RTT link at speeds of 100 Mbit/s or more.

2.3.2 Bufferbloat

In the loss-based schemes for congestion control described thus far, the TCP sender is continually probing the network for more bandwidth by attempting to send more and more data. This mechanism of detecting congestion, not taking `rwnd` into account, relies on the fact that network equipment has a finite amount of storage; the filling of these storage buffers is equated with network congestion through tail drop. As far back as 1985, in RFC 970 [64], Nagle was able to show in experiments that sufficiently large network buffers would cause TCP to time out. In a more recent scenario, Beijnum [3] describes how, when downloading a large file on a router with a queue of about 80 packets connected to a 128 kbit/s network, web browser traffic became practically unresponsive owing to being forced to compete with a standing queue of 10 seconds of packets in the router. The situation in which one flow (or a few flows) monopolizes the queue is called *Lock-Out*; in the case of [3] the browser traffic is disadvantaged because its `cwnd` is never allowed to grow sufficiently to compete with the flow currently inhabiting the queue. This problem of network equipment coming equipped with excessively large buffers than is generally warranted is called *Bufferbloat*.

Limiting the buffer size is not a viable long-term solution, as a main benefit of large buffers is that they absorb bursts of packets that can be forwarded in *silent periods* between bursts [51]. Decreasing the size of network buffers in order to improve the steady-state performance of TCP would be counter-productive, as these flows too exhibit bursty behavior at times and benefit from having large buffers to absorb these bursts. Braden *et al.* [51] note that the size of network buffers should reflect the intended maximum supported burst size. A possible solution would be one that could signal congestion on a router to TCP senders before the buffer fills up completely.

2.3.3 Active Queue Management

Tail drop is a *passive* queue management scheme, in that it takes no steps towards early notification of congestion; as long as the queue is not full, nothing is done. Two other possible approaches to passive queue management are *random drop on full* and *drop from front when full*, neither of which solve the problem described at the end of section 2.3.2[5].

AQMs *actively* drops packets from the existing queue, even when not full, so that the TCP senders currently inhabiting the queue are signaled of congestion before it becomes a serious threat. In this way, a TCP that reacts appropriately to such loss will not lose out on their fair share, while not excessively congesting the network. Braden *et al.* offer that the active management of router queues would lead to fewer packets being dropped in routers, lower delays for interactive services through smaller queues on average, and less severe lock-out behavior. Gettys and Nichols [21] warn that, even today, networks without an effective AQM may be vulnerable to congestion collapse. RED [19] is an example of an AQM which drops or ECN-marks a percentage of packets when the queue reaches a threshold.

2.4 Delay-Based congestion control

Packet loss as a congestion signal is reactive in nature; it implies that the network is already congested. Delay-based congestion control schemes use *propagation delay* to approximate network congestion, a signal that can indicate congestion by way of increasing RTTs. The delay is measured by the difference in time between when a segment was sent and the receive time of the corresponding ACK; the idea is that the network implicitly signals increasing congestion by virtue of expanding router queues, resulting in higher round-trip times. In this way, delay-based congestion control schemes may react to congestion before it becomes a problem, while a loss-based congestion control would delay backing off until the network is already congested which can potentially exacerbate the situation.

2.4.1 Underlying Mechanisms

Generally, one of two possible delay-based congestion signals are used: RTT or One-Way Delay (OWD). RTT is the easiest to use in already established networks as it requires changes only in the implementation of the TCP sender; the sender only needs to keep track of the timestamps of packets sent and compare them to the timestamps of their corresponding ACKs. The downside to using RTT as a measure of delay is that any measurement made this way will include the delay for both the outbound route and the inbound one. If the return path is more congested, the sender TCP will overestimate network congestion. The route between the hosts is additionally not guaranteed to be equivalent for both directions, in which case the RTT-based delay measurement will not be representative of the congestion on the outbound path.

Conversely, OWD measures the time between the packet being sent and the packet being received on the opposing end of the connection. The measurement does not consider the return path, consequently it is not affected by congestion on the return path. A requirement for OWD-based delay measurement is that the TCP on the receiving end must support the mechanism as it is not supported in TCP by default [54]. Subsequently, most algorithms in use today that feature OWD are implemented as custom protocols over UDP, such as the first implementation of LEDBAT [66].

[5]Though they both solve the Lock-Out problem [51].

The receiver calculates OWD as the difference between a timestamp received with each incoming TCP segment and the receiver's own system clock, which it then communicates back to the sender. Because the system clocks of the two hosts will never be exactly equal[6], the delay calculated using this method can never be the absolute OWD on the outbound path, but rather an estimation can be made based on the relative differences in the measurements [78]. Low Extra Delay Background Transport (LEDBAT), for example, uses a simple OWD measurement to establish a base delay by taking the minimum of all measured OWDs and subsequently using this base OWD measurement as a base for other measurements — thus all delay measurements done in LEDBAT are relative to the minimal OWD seen[7].

Getting a reliable estimate of the base delay is paramount, as underestimating the network congestion may lead to erroneously adding packets to an already congested network. Upon entering an already congested network, getting an accurate measurement of the base delay is nearly impossible as this measurement requires a packet to traverse an empty router queue, which in a congested network may never happen. LEDBAT as specified in RFC 6817 [59] famously suffers from *latecomer advantage* [9], in which a flow arriving at a congested link includes the existing queue in its base delay causing it to underestimate its congestion contribution and causing the existing connections to see ever-increasing queues, which is even acknowledged in the specification of LEDBAT itself. FLOWER, a newer development in LBE transport protocols, aims to solve the aggressiveness and latecomer issues in LEDBAT by employing what they call a *fuzzy* congestion controller.

An additional problem for delay-based congestion controls is the fact that the delay signal is noisy. As described in section 2.3.2, Internet traffic typically occurs in bursts which naturally leads to bursts in the delay signal as well. Any significant use of delay as a congestion signal must therefore smooth this value somehow.

2.4.2 TCP Vegas

Vegas [6], [40, Version 4.16] uses RTT for its delay measurements, specifically the delay as given by the minimum RTT measured during the previous RTT. The minimum delay measurement seen for the current flow is used for the base RTT measurement, consequently Vegas may potentially exhibit latecomer unfairness similar to that of LEDBAT.

The congestion control of Vegas operates by calculating the expected sending rate, more specifically the `cwnd`, given the base delay and the current `cwnd`, and subtracting this from the actual sending rate including queuing delay. The derived value represents the difference between actual and expected sending rate as segments in the `cwnd`. The parameters α, β, and γ are boundary values that govern the behavior of Vegas based on the previously calculated difference in expected and actual `cwnd`, essentially the number of queued segments Vegas is allowed to incur on the network. In Slow Start, identical to that of Reno, γ decides how much congestion Vegas is allowed to induce during Slow Start and functions as an additional trigger point for Congestion Avoidance. Outside of Slow Start, α and β together define three zones of behavior depending on the segments of extra queuing, `diff`, Vegas infers it is putting on the network:

`diff` $< \alpha$**:** Allowed to put more packets in the network; increase `cwnd`.

[6]The system clock is affected by external sources, thus no two system clocks will ever be exactly synchronized or advance at exactly equal rates. An attack has even been demonstrated using CPU heat to produce extra clock skew in order to identify a hidden service on the Tor network [44].

[7]Using only relative values eliminates the need to counteract clock offset, where the difference between two system clocks is constant. In RFC 6817 [59, appendix A.2], clock skew is described as being an insignificant issue for LEDBAT.

diff $> \beta$**:** Incurring too much congestion; decrease `cwnd`.

$\alpha \leq$ **diff** $\leq \beta$**:** Sending at an acceptable rate; change nothing.

Brakmo *et al.* claim that Vegas "is able to achieve between 40 % and 70 % better throughput than Reno" ([6]), yet further analysis has revealed that Vegas performs significantly worse than competing loss-based traffic for tail drop routers with large buffers [23]. Vegas is available in Linux as a kernel module.

2.5 Less-than Best Effort (LBE) Transport Protocols

Typical transport protocols attempt to utilize as much of the network bandwidth as possible, while remaining fair in relation to its peers. In an ideal situation, ten TCP flows sharing a link would each be guaranteed at least 10 %, or $\frac{1}{n}$ for n flows, of that link's capacity; this would yield the maximum utilization of available resources and would be considered *fair*.

A LBE protocol strives to remain unobtrusive to other, potentially more aggressive, TCP flows. It will, in the same way as typical TCPs, attempt to utilize as much of the available network resources as possible, but when it shares a link with a more aggressive TCP flow, LBE will yield its share of the network bandwidth for the benefit of its competitors. As such, a LBE protocol is a scavenger protocol, taking what available bandwidth it can but never directly competing. The most prevalent use of these kinds of algorithms in the real world comes from BitTorrent (which uses LEDBAT) [66] and Apple [2]; LEDBAT was thought to carry 13 % to 20 % of internet traffic in 2013 [67]. LEDBAT [59], similarly to Vegas, adjusts its sending rate according to how much extra delay it perceives it is incurring in the network, but while Vegas targets a (very low) maximum number of extra segments of queuing, LEDBAT instead aims for a maximum latency, at most 100 ms [59].

2.6 Deadline Aware Less-than Best Effort (DA-LBE)

A DA-LBE service is one that aims to provide LBE service while also including a notion of timeliness. Meant for applications that do not require assurances of capacity or latency yet would still like to provide a loose timeliness guarantee, a DA-LBE transport service can transfer data in the background, not impacting quality-bound competing traffic, while providing the loose timeliness guarantee of a *soft deadline*. This soft deadline target allows a data transfer to arrive after the given deadline if the network conditions prohibited it reaching the deadline in time. When needed, the DA-LBE service may transmit with aggressiveness on par with standard TCP.

In this section we will explore a way to achieve DA-LBE service by adjusting an existing TCP for our goals. This is done through the use of a DA-LBE congestion *metacontroller*, a service on top of the existing TCP, which periodically adjusts parameters on the TCP flow while it is transmitting.

2.6.1 Modeling the Price of Network Congestion

Tuning the perceived congestion in a congestion controller requires the mapping of congestion signals, along with knowledge about one's own impact upon the network, to a universal *price* of congestion. For a loss-based congestion control, the kind of which is described in section 2.3.1.1, the loss of packets is feedback from the network about the congestion resulting from one's own sending rate. With a solid understanding of the mechanisms of one's own

congestion controller along with the congestion controllers in use in the network it is possible to use the aforementioned feedback to put a price on the congestion a congestion controller is creating in the network.

This thesis uses work by Hayes *et al.* [24], wherein the authors use Network Utility Maximization (NUM) [33], [34], [68] to map congestion control specific prices to a universal price [72] for congestion measurement. NUM frames the network congestion control problem as an optimization problem, where the aim is to maximize the bandwidth utilization for each traffic source. The Lagrangian Dual is used to solve this optimization problem, and it is specified that the Lagrange multiplier can be used as a *price* of congestion for each link. Congestion controllers could *artificially* scale these congestion prices to change their relative perceived fair share of the network capacity.

Trichakis *et al.* [76] investigate a situation in which certain flows, for certain periods, may have a minimum sending rate requirement. To conform to the set rate, senders may deflate the measured congestion price, allowing them to send faster than their fair share warrants. In the case of a DA-LBE service, Hayes *et al.* state that flows would usually need to *inflate* the measured congestion in order to offer parts of their fair share to competing flows, but may also choose to deflate the measured congestion when ramping up aggressiveness to reach a deadline. Section 2.6 elaborates on exactly how this is done for the parts of [24] that are implemented in this work.

2.6.2 DA-LBE metacontrollers

The default congestion controllers, the *metacontrollers*, provided with this thesis are based on work by Hayes *et al.* [24]. Hayes *et al.* describe possible metacontroller implementations using both a Model-Based Controller (MBC) and a PID controller, of which we choose to focus on the MBC given that its tuning is easier.

2.6.2.1 Weight

The LBE sender is modeled as a traffic source that inflates its measured network price, q, by some weight $w \in [w_{min}, w_{max}]$. The limits are defined such that when $w = w_{min}$ the service is transmitting at its lowest possible level of aggressiveness, as *LBE* as it can be; and when $w = w_{max}$, it is sending at rates equivalent to a BE sender. Throughout the lifetime of the flow this weight is changed in relation to the network conditions so as to reach $\hat{q}$, the price for the sending rate required to meet the deadline exactly. It is adjusted at regular intervals, designated t_n, of time T_w apart, with an additional longer term adjustment at time intervals of T_ϕ. This longer interval is meant to adjust for long term trends in network congestion, the general traffic on the network, while the shorter interval accounts for short term more volatile changes.

The weight in the short term is determined mostly by the actual measured rate in proportion to the target rate. The target rate, the lowest required rate that will reach the deadline, is defined as:

$$\zeta(t_n, t_D) = \frac{\text{data remaining}}{t_D - t_n} \tag{2.2}$$

where t_n is the n^{th} time interval since the start of the flow, and t_D is the deadline.

Equation (2.2) is used to derive an *error* term: ϵ_{n-1}, where ϵ_{n-1} is the error for the preceding interval to the one being predicted; the error in the previous interval is used to

extrapolate network conditions for the next interval. This error term is calculated as follows:

$$\epsilon_{n-1} = \frac{\zeta(t_{n-1},\, t_D) - \bar{x}(t_{n-1},\, t_n)}{\bar{x}(t_{n-1},\, t_n)} \tag{2.3}$$

where $\bar{x}$ designates the measured sending rate for the given interval. Note that $\zeta(t_{n-1},\, t_D)$ is the target rate of the *previous* interval; we can now compare the target to the actual measured rate for that interval, deriving an error term by which the price of the next interval can be corrected.

The error term from equation (2.3) is then used to derive the weight $\hat{w}$ needed to reach the desired price $\hat{q}$ for the next interval t_n:

$$w_n = \left\lceil \frac{q_{n-1}}{\hat{q}_n}(1 + \epsilon_{n-1}) \right\rceil_{w_{min}}^{w_{max}} \tag{2.4}$$

where q_{n-1} is the measured congestion price for the last interval T_ϕ.

Additionally, a growth limit is imposed on the weight calculation such that the model never experiences upward jumps in aggressiveness; the model errs on the side of caution and assumes increased congestion in one interval is indicative of congestion in the next one. The weight growth limit is defined as follows:

$$\hat{w}_n = \begin{cases} w_{n-1} + l_w w_n & \text{if } (w_n - w_{n-1}) > l_w w_n \\ w_n & \text{otherwise} \end{cases} \tag{2.5}$$

where l_x is the limit defined for growth of x, in this case the weight w.

Equation (2.5) limits the growth such that the value still increases in relation to the *actual* new value, but never excessively. If the new value is lower than the limit however, it is accepted; it follows that sudden drops in aggressiveness are allowed, which results in the model favoring LBE behavior when adapting to network congestion.

2.6.3 Model-Based Controller for TCP Cubic

Hayes *et al.* suggest two ways of altering the perceived network congestion for a loss-based congestion control: Generating *phantom* ECN signals, or adjusting the *decrease factor*. The former is a fairly straight forward way to signal congestion to the controller without causing actual loss, the primary congestion signal, which also means that this method does not trigger unwarranted retransmissions. However, as we can only alter the chance of generating phantom ECNs on every interval T_w, the controller is stuck in that specific mode of aggressiveness scaling for the entire interval; it is not able to take advantage of small periods of lower network congestion.

The latter of the two mentioned methods of adjusting the controller aggressiveness is the adjustment of the Cubic decrease factor, or β. This parameter is the factor by which the controller reduces its `cwnd` in response to congestion; a lower β results in backing off more on each congestion event. As this method of scaling aggressiveness is tied to actual loss events, it follows that for periods of little or no congestion — the aforementioned periods of low congestion — the model behaves as aggressively as it would have without LBE adjustments. Adjusting β is in principle the most suited method of tuning the aggressiveness of loss-based TCP flows like Cubic.

However, due to how Linux handles kernel modules, the β parameter can not be changed once the connection has been set up, which makes adjusting β in the middle of a connection impossible in practice[8]. Furthermore, tampering with module parameters changes the

behavior of all other processes that depend on that module; seeing as Cubic is the default congestion controller for Linux[8], this means that all TCP flows would share the β parameter, effectively negating our efforts in making just *one* specific flow LBE. Phantom ECNs remain the preferred choice for adjusting the perceived network congestion of loss-based congestion controllers.

The algorithm for adapting TCP Cubic to become a Deadline Aware Less-than Best Effort controller using phantom ECNs as an extra congestion indication is as follows:

Every T_ϕ***:*** Update q
$\quad q \leftarrow \mathbb{P}[\text{loss}]$

Every T_w***:*** Update $\mathbb{P}(\text{Phantom ECN})$

$$\widehat{\text{cwnd}} \leftarrow \zeta(t_n, t_D) \times \overline{\text{RTT}}$$

$$\hat{q}_n \leftarrow \frac{\text{RTT}_{\min}}{\widehat{\text{cwnd}}^{\frac{4}{3}} \left(\frac{4(1-\beta)}{(0.4(4-(1-\beta)))}\right)^{\frac{1}{3}}}$$

$$\epsilon_{n-1} \leftarrow \text{equation (2.3)}$$

$$w_n \leftarrow \text{with } \epsilon_{n-1},\ q, \text{ and } \hat{q}_n\text{, calculate a weight using equation (2.4)}$$

$$\hat{w}_n \leftarrow \text{limit } w_n \text{ using equation (2.5)}$$

$$\mathbb{P}(\text{Phantom ECN}) \leftarrow q\left(\frac{1}{\hat{w}_n} - 1\right)$$

Algorithm 1: Model-Based Controller for DA-LBE Cubic, adapted from Hayes *et al.* [24, Fig. 3]

$\mathbb{P}[\text{loss}]$ is the probability of a loss event, or more precisely a *general* congestion event, being generated in the network, interpolated from the network state in the previous interval, defined as:

$$\mathbb{P}[\text{loss}] = \frac{I}{N} \tag{2.6}$$

Where I is the number of loss events for the previous interval and N is the number of packets acknowledged for the previous interval.

An additional control is introduced that attempts to allow DA-LBE Cubic to make use of periods of low congestion: The model tracks the time between *real* congestion events, τ_{cong}, and prevents triggering *phantom* ECNs if it detects that a significant amount of time has passed since real congestion was detected. This time between congestion events is averaged over the previous few events using Exponentially Weighted Moving Average (EWMA) and, in deciding whether to inhibit phantom ECNs, is scaled by a factor v so as to err on the side of acting Less-than Best Effort rather than Best Effort.

In the DA-LBE kernel component, the phantom ECNs are applied in the following manner:

[8] As of Linux version `4.13.0`.

On every received ACK:

$t_{\text{cong}} \leftarrow$ time since the last *real* congestion indication

$\tau_{\text{cong}} \leftarrow$ average time between *real* congestion events

if $t_{cong} > v \times \tau_{cong}$ **then**

  if `rand()` $< \mathbb{P}(\textit{Phantom ECN})$ **then**

    Trigger a phantom ECN

Algorithm 2: Algorithm for triggering phantom ECNs in the DA-LBE kernel component.

2.6.4 Model-Based Controller for TCP Vegas

The DA-LBE Vegas controller described by Hayes *et al.* [24] does not map directly to the supported mechanisms in this work, or that of the DA-LBE kernel component. As described in section 2.6.3, the adjustment of kernel module parameters is not reflected in the code until the module has been reloaded; as the β approach was rendered invalid, so too is the α-adjustment approach outlined by Hayes *et al.* unfit [24, Fig. 7]. Using similar devices for determining the price of network congestion, we can adjust the perceived load the congestion control is putting on the network by modifying the congestion signal, instead of adjusting the α parameter. For DA-LBE Vegas, the congestion being put on the network is measured as the difference between the average RTT and the base RTT, known as the *queuing delay*.

The method for determining the price of DA-LBE Vegas congestion is more involved than that of DA-LBE Cubic. Vegas reacts both to delay and loss, and so both must be considered for the model to be correct.

The network price calculation from equation (2.6) must be expanded as follows:

$$q \leftarrow \frac{W(\text{loss-reno})\mathbb{P}(\text{loss-reno})[\text{cong}] + W(\text{delay})\mathbb{P}(\text{delay})[\text{cong}]}{W(\text{loss-cubic})\mathbb{P}(\text{loss-cubic})[\text{cong}]} \tag{2.7}$$

For the delay-based TCP we define the I for $\mathbb{P}(\text{delay})$ to be the number of times the `cwnd` has been reduced as a result of the delay congestion signal, that is every time the Vegas perceives that it is incurring more than β segments of queuing delay in the network. $W(z)$ is the reduction in `cwnd` for the congestion controller z. For the loss-based congestion controllers Reno and Cubic, their W is the factor by which they reduce their congestion windows on loss, respectively 0.5 and $1.0 - \beta \approx 0.3$. For Vegas the amount by which its `cwnd` is reduced varies with the size of the congestion window, as it is adjusted up and down by just one segment in congestion avoidance mode. We use the mean factor by which the Vegas's congestion window is reduced as a result of the delay congestion signal, as reported by the DA-LBE kernel component. The network price calculation outlined in equation (2.7) is thus the chance for Vegas to experience a congestion event of any kind, normalized by the chance of a congestion event for the type of congestion controller that is active in the network, in our case defined to be Cubic.

The algorithm for adapting TCP Vegas to become a DA-LBE controller using queuing delay adjustment is as follows:

Every T_ϕ***:*** Update q

$q \leftarrow$ equation (2.7)

Every T_w***:*** Update μ

$$\hat{d}_n \leftarrow \frac{\alpha}{\zeta(t_n,\, t_d)}$$

$$d_n \leftarrow \overline{\mathrm{RTT}} - \mathrm{RTT}_{\mathrm{base}}$$

$$w_{\mathrm{base}} \leftarrow \left(\frac{d_n}{\hat{d}_n}\right)^{-q}$$

$\epsilon_{n-1} \leftarrow$ equation (2.3)

$$w_n \leftarrow w_{\mathrm{base}} \times (1 + \epsilon_{n-1})$$

$\hat{w}_n \leftarrow$ limit w_n using equation (2.5)

$$\mu_n \leftarrow q\hat{w}_n$$

if $\hat{w}_n = 1.0$ **then**

$$B \leftarrow 1.0 - \frac{1}{\mu_n}$$

Algorithm 3: Model-Based Controller for DA-LBE Vegas, loosely adapted from Hayes *et al.* [24, Fig. 7]

where the resulting μ is the factor by which the queuing delay is adjusted. B is the chance to ignore a loss signal, only enabled when w is 1.0 and thus at its most aggressive. α is the Vegas parameter.

In the DA-LBE kernel component, the queuing delay adjustments are applied as follows:

On every received ACK:

$$d \leftarrow \mathrm{RTT} - \mathrm{RTT}_{\mathrm{base}}$$

$$\mathrm{RTT} \leftarrow \mathrm{RTT}_{\mathrm{base}} + \frac{d}{\mu}$$

Algorithm 4: Algorithm for adjusting perceived RTT in the DA-LBE kernel component.

where d is the measured queuing delay. The measured RTT is changed in place and is then passed on to the underlying congestion controller as usual.

2.6.5 Soft Deadline

The word *deadline* should be read as *soft deadline* when used in relation to DA-LBE data transfers, the reason being that deadlines for DA-LBE flows are not, and can never be, absolute. The two ways to make data transfer deadlines absolute are either ensuring that all data is transferred in time for the deadline or, should the transfer not be complete in time for the deadline, stop the data transfer. Neither of these approaches are appropriate for DA-LBE: The former might require allowing the sending rate to surpass the fair share allocated for each flow; a practice which, as described in section 2.3, is unsafe for the stability

of the Internet. The latter is not appropriate for TCP, as it departs from what an application expects from a TCP service. It follows that the responsibility of enforcing an absolute deadline should lie with the application, which also puts absolute deadlines out of scope of this work.

How the DA-LBE metacontroller reacts to the deadline is dependent on the model being used: In the models used in the implementation provided in Libdalbe the behavior is tied to a measured *error* term derived from the proportional difference in the measured actual sending rate versus the target sending rate[9]. The value for the target sending rate is capped at a reasonable maximum value, for which the model will adjust the flow to be as aggressive as is allowed. After the deadline has passed this maximum target sending rate is kept, thus the model runs at maximum aggressiveness and will compete fairly with other TCP traffic until the end of the flow.

2.6.6 Previous work by Lars Erik Storbukås

Our work is built on top of a Linux kernel DA-LBE component engineered by Lars Erik Storbukås as part of his Master's thesis [70]. His component provides mechanisms in the Linux kernel to gather statistics and adjust congestion signals for an arbitrary CC, which we employ in Libdalbe to implement DA-LBE control mechanisms. The API for interfacing with the component, provided through the standard Linux TCP header, as well as a detailed breakdown of the provided functionality and use cases for Libdalbe can be found in appendix B.

Storbukås's goal, as stated in his sections *Problem statement*, *Motivation*, and *Research questions* [70, section 1.1–1.3], was to "Impose LBE behaviour with a notion of time in order to support soft deadlines by dynamically adjusting the aggressiveness when competing with BE network traffic [...]" and "Support a wide range of CCs as opposed to attempt to develop a one-size-fits-all CC [...]". In his thesis he describes how he has implemented functionality in the Linux kernel to achieve these goals. He concludes that his implementation of controlling arbitrary CCs is successful in limiting the sending rate of his chosen CC algorithms, Cubic and Vegas. We found that we needed to make changes to the kernel for the DA-LBE component to operate as presented by Storbukås. In section 3.6 we describe the changes we made.

2.7 Summary

In this chapter we outlined how applications wanting to transmit data with a lower bandwidth allocation, but still with certain timeliness demands, can achieve this using a DA-LBE transport service. We described how we can create a DA-LBE service by tuning an existing BE TCP to exhibit LBE-like behavior and changing its parameters during the connection to maintain a target rate with the goal of finishing transmission within a soft deadline. The DA-LBE transport services we create, one for TCP Cubic and one for TCP Vegas, are *metacontrollers* that estimate the congestion in the network and their own congestion impact by mapping congestion signals to a *price* and applying adjustments to the underlying TCP to achieve DA-LBE service. DA-LBE Cubic is adjusted through generating *phantom* ECNs, as if a router in the network had signalled congestion, while DA-LBE Vegas has its perceived queuing delay adjusted.

[9] The sending rate required to reach the deadline on time.

Chapter 3

Libdalbe

Libdalbe is an interface to the DA-LBE statistics and control mechanisms in the kernel. The library allows application developers to create and experiment with their own *meta*-congestion controllers (*metacontrollers*), potentially providing LBE or DA-LBE service, without having to massively adapt their applications. The metacontrollers, by virtue of adapting actual TCP machinery conceptually residing in the Transport layer in the TCP/IP-stack, shown in figure 2.1, do not suffer from the protocol overhead caused by implementing a transport over UDP and can directly make low-level changes to the transport being controlled. Implementations of model-based controllers enabling DA-LBE functionality for both TCP Cubic and TCP Vegas are provided as proof of concept, along with testing and validation in chapter 5.

3.1 Structure

Libdalbe is a library which allows utilization of DA-LBE enabled sockets. Figure 3.1 shows how an application interacts with the library interface and the kernel.

DA-LBE sockets are created using a function similar to that of the standard `socket` system call, where the developer must additionally provide a deadline and the expected size of the data transfer. They must also provide the underlying TCP congestion control algorithm to adapt, selected from a list of supported protocols, and optionally their own custom metacontroller. The function returns a standard Unix file descriptor, through which the developer can send data as they would any other file descriptor as shown by the call to `send` in figure 3.1.

New sockets are handed to a daemon thread which performs the necessary meta-congestion control operations asynchronously on behalf of the developer's application. Each time interval the daemon will update the DA-LBE information on a socket and call the corresponding control function. The daemon will never block access to a socket; the developer may continue to write to the socket regardless of the deadline, until they desire to close it.

Sockets must be closed through a library-provided function identical to the `close` system call. This allows the library to clean up the resources used for the given socket and close it gracefully with the actual `close` system call.

When the developer desires to cease using DA-LBE sockets, they must call the `dispose` function, which will stop the daemon thread and dispose of allocated resources.

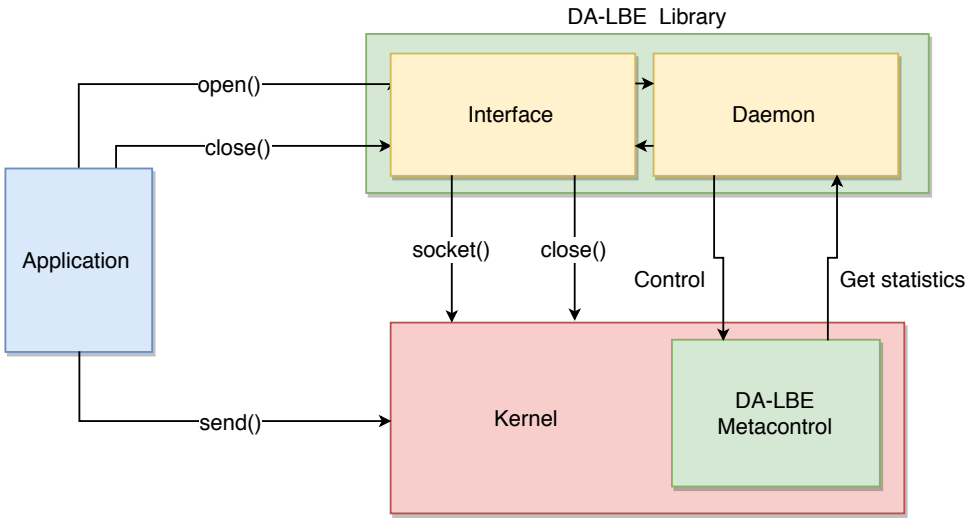


Figure 3.1: An overview of Libdalbe in relation to the user's application, the kernel, and the DA-LBE kernel component.

3.2 Basic Operation and Design Decisions

The intent of Libdalbe is to enable manipulation of low-level congestion control options in the kernel at runtime. Our aim is two-fold: (1) Provide an easy way for developers to use DA-LBE enabled data transfers, and (2) Allow experimenting with congestion control metacontrollers using the runtime control provided by our daemon. Secondarily, Libdalbe provides an easier interface for applications to access DA-LBE socket parameters than by manually executing cryptic system calls.

3.2.1 Library

Implementing the interface as a C library allows Libdalbe to expose the existing resources, the DA-LBE kernel component[1], in the kernel without wrapping them for use in another language. Libraries are trivial to import into C or C++ applications and, with a little extra work, can be exported for use in a multitude of popular programming languages [11], [15], [32], [79]. The same functionality provided in the library could arguably have been achieved through a Linux kernel module, a separate process, or a service instead of a library. These solutions would have allowed the same metacontrol functionality, but they would all have required an interface which would again warrant a library for ease of use instead of requiring Inter-Process Communication (IPC) or system calls. Implementing all of the required functionality in one library without having to rely on external setup was deemed the easiest solution both in terms of ease of use and ease of development.

Keeping the interface consistent with the standard socket interface (see section 3.4) was an important consideration. As a result, the changes required in the source code to replace a TCP connection setup with the functionality from this library are trivial to implement for anyone familiar with Unix sockets.

[1]The minutiae of the control algorithm being adapted are not accessible due to limitations in the user-facing Linux socket interface.

3.2.2 Daemon Worker Thread

The daemon operates as a separate thread inside the process of the user's application. The library is structured such that the main thread — the one belonging to the application using the library — is not tied up in socket control operations. Interface functions simply start or stop the daemon and manipulate the list of sockets on which the daemon operates; this way we can guarantee that each socket is updated at precisely the right time, barring heavy calculations in the control function.

A process or service based implementation would have provided the same type of safety. In Linux, file descriptors can be passed between processes using UNIX domain sockets [36, sec. 61.13.3], which opens the possibility for using shared sockets to facilitate this kind of metacontrol. The daemon being in a different process would have made it possible to use signals to wake the worker thread from sleep without the potential of needlessly interrupting the main application and without hijacking signals from the developer, as described in section 3.7.2. Sharing sockets between processes, however, is neither trivial nor portable and it would still have made a library desirable for comfortable use by application developers.

The daemon in the finished implementation is very simple; it iterates over its list of open sockets, sleeps until the next socket is ready, and performs the chosen metacontrol operation. Locks are in place to make sure the daemon always works on a valid open socket, assuming the connection has not broken down, and that the main thread cannot remove sockets while they are being worked on.

3.2.3 Allowing for Custom Metacontrollers

Libdalbe facilitates the implementation of custom metacontrollers. The daemon exposes underlying statistics for the active congestion control along with DA-LBE metrics, allowing the metacontroller to reason about the state of congestion on the network and adjust the congestion control accordingly. On opening a socket, the user may provide their own functions, metacontrollers, to adapt the underlying congestion control. They may optionally provide a function to explicitly initialize their own data structures which will be called from the daemon, alleviating the need for writing thread-safe code for this purpose.

Though the implementations provided with Libdalbe are DA-LBE controllers, the custom controllers need not be. The kernel control interface may be used to provide a general LBE service, or a service not related to LBE at all. A long lived flow could for example be controlled to disregard the deadline and provide LBE service only during working hours. Libdalbe allows for creative use of the custom metacontroller functionality beyond that of the intended DA-LBE purpose.

3.3 Interface Overview

The basic user facing interface contains four functions with which to open and close communication sockets and initializing and disposing of the library resources in a graceful manner. Following is a brief description of the interface along with the basic operation of its functions. A detailed explanation of parameters, return values, and inner workings can be found in appendix A.

- **`dalbe_init`** Initialize Libdalbe. Must be called before using any of the other functions of the interface, otherwise their behavior is undefined.

- **dalbe_dispose** Dispose of the resources generated by Libdalbe. Should be called after the application has finished using library resources, so unneeded memory and the inactive daemon can be reclaimed.
- **dalbe_open** Open a new DA-LBE enabled socket. A new socket is opened with which the user can transfer data, and optionally operate a TCP metacontroller.
- **dalbe_close** Close a DA-LBE socket previously created by a corresponding call to `dalbe_open`. Closes the given socket and disposes of its resources.

3.4 Sample Usage

```
1 int sock;
2 sock = socket(AF_INET,
3               SOCK_STREAM,
4               IPPROTO_TCP);
5 connect(sock, addr, addrlen);
6 while (not_finished)
7         send(sock, data_ptr,
              data_remaining, flags);
8 close(sock);
```

Listing 3.1: Typical minimal connection setup and usage of a socket on the sender side of a TCP connection (error handling omitted).

```
1  int sock;
2  dalbe_init(NULL);
3  sock = dalbe_open(AF_INET,
4      "cubic", flow_size, deadline,
5      NULL, NULL, 0,
6      NULL);
7  connect(sock, addr, addrlen);
8  while (not_finished)
9          send(sock, data_ptr,
                data_remaining, flags);
10 dalbe_close(sock, NULL);
11 dalbe_dispose(NULL);
```

Listing 3.2: Typical minimal connection setup and usage of a DA-LBE socket (error handling and custom metacontrol omitted).

Listing 3.2 shows a simple usage example of Libdalbe. Note that this example does not show the use of custom metacontrollers, for which one would make use of the arguments on line 5. The argument given `NULL` on line 6 is the optional buffer for error messages. In comparison to listing 3.1, a simplified example of standard socket usage, DA-LBE sockets require only initialization and disposal of the library in addition to a slightly altered function call to create a socket. A complete example application transferring a message arbitrarily many times is included in appendix F.

3.5 Custom Metacontrollers

Figure 3.2 shows an overview of an application that has set a custom metacontroller. The application interfaces directly with the kernel for typical socket operations, i.e. the sending and receiving of data, while the daemon handles DA-LBE operation. The figure also shows how the custom metacontroller acts directly on the DA-LBE kernel component, while the daemon handles the gathering of DA-LBE metrics.

The library is set up to facilitate controller algorithms which make their decisions largely based on the two sets of measurements, one for the current interval and one for the preceding interval. This structure rose out of a specific need in the controllers being planned for implementation and was kept as a simple baseline; the measurements used by both of the

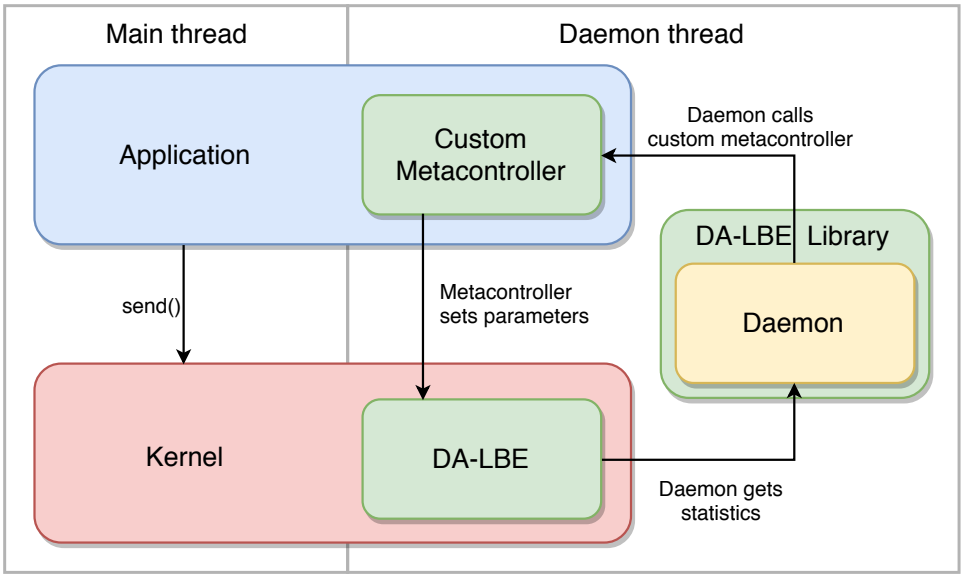


Figure 3.2: Overview of an application using a custom metacontroller. The application in this figure has defined their own custom metacontroller and passed it to the socket opening call in Libdalbe.

controllers, described in detail in section 3.5.2, are always taken from either the current or the previous time interval. Developers needing a longer history of measurements are free to set up the needed structures in the initialization function without needing to concern themselves with potential synchronization issues arising from sharing memory with the daemon.

Interface functions are available to aid in setting parameters on the DA-LBE kernel component. Libdalbe translates from the floating point values supported in the metacontroller to the required values for fixed point arithmetic in the DA-LBE kernel compenent [70, section 3.4.1]. *Libexplain* [42] provides helpful error messages for the kernel interface.

Mechanisms such as setting the chance for a delay-based flow to ignore `cwnd` backoff on loss events, meant to let DA-LBE Vegas compete on par with Cubic if needed, can easily be tuned to out-compete any TCP. Libdalbe does not have the ability to check whether an alteration to the TCP would result in unfair behavior, so the user must take extreme care not to run unsafe experiments on the Internet.

3.5.1 Basic Usage by Example

Upon opening a DA-LBE socket, the user may provide a function for the daemon to call in order to perform metacontrol. The function takes the DA-LBE library representation of a socket as its sole argument. No further restrictions are imposed; no values are required to be set, no callbacks are required to signal completion to the library, and no value need be returned.

Listing 3.3 shows an example metacontroller where the chance of triggering a phantom ECN is mapped to the proportion of time passed in relation to the deadline without any further modeling. No auxiliary storage is required in this case and thus there is no need for an initialization function.

In listing 3.4 more functionality is showcased in a contrived example of a metacontroller which increases the phantom ECN probability exponentially with different rates after a set limit, resetting after hitting a probability of 0.5. The controller needs state to be kept between

```
1  uint32_t deadline = 600;
2  void linear_phantom_ecn_scale(struct socket *sock) {
3                  uint32_t secs = sock->time_remaining.tv_sec;
4                  double ph_ecn_p = secs / deadline;
5
6                  dalbe_set_phantom_ecn_probability(sock->fd, ph_ecn_p, 0);
7  }
```

Listing 3.3: An example custom metacontroller.

```
1  struct cc_info {
2          double ph_ecn_p;
3  };
4
5  void init_cc(struct socket *sock) {
6          struct cc_info *data = sock->info_now->data;
7          data->ph_ecn_p = 0.001;
8          dalbe_set_phantom_ecn_probability(sock->fd, data->last_p, 0);
9  }
10
11 void cc_func(struct socket *sock) {
12         struct cc_info *info_now  = sock->info_now->data;
13         struct cc_info *info_prev = sock->info_prev->data;
14
15         if (info_now->ph_ecn_p > 0.5)
16                 info_now->ph_ecn_p = 0.01;
17
18         else if (info_prev->ph_ecn_p > 0.3)
19                 info_now->ph_ecn_p = info_prev->ph_ecn_p / 2.0;
20
21         else
22                 info_now->ph_ecn_p += info_now->ph_ecn_p;
23
24         dalbe_set_phantom_ecn_probability(sock->fd, info_now->ph_ecn_p, 0);
25 }
```

Listing 3.4: A example of a metacontroller showcasing saved state and explicit initialization.

calls to the metacontrol function, consequently the first state must be initialized, as is shown in the initialization function defined on line 5. Listing 3.4 would have required the developer to give `sizeof(struct cc_info)` as the argument for `user_data_size`, allowing the library to allocate memory for the metacontroller state.

3.5.2 Sample Implementations

Provided as part of Libdalbe are two sample implementations of DA-LBE congestion metacontrollers for controlling TCP Cubic and TCP Vegas as described in section 2.6.3 and section 2.6.4. Both metacontrollers work by applying adjustments in intervals of 10 s with longer intervals of 60 s to adjust for longer-term network behavior. They are both MBC, meaning that they model the underlying TCP based on metrics read from the DA-LBE kernel component and apply adjustments to achieve the desired behavior. If the model accurately represents the TCP under control, and if the current network conditions hold, the MBC should be able to achieve the exact desired outcome of its adjustments.

The source code for both of the metacontrollers can be found in appendix F.

3.5.2.1 MBC TCP Cubic

The metacontroller for controlling TCP Cubic is very close to that of algorithm 1. The main change made is scaling the longer-term network price by the Cubic β parameter, so instead of using the chance of a loss event as the network price, we are effectively using the average change in `cwnd` size — similar equation (2.7) from the DA-LBE Vegas MBC.

In section 2.6.2.1 we mentioned how the network price was clamped to a value range, scaling from least aggressive, for lower values, to most aggressive for higher values. For our implementation of a DA-LBE Cubic metacontroller, we clamped w to $[0.0001, 1.0]$, and the growth in w is limited to 5 % of the new value. The w value used in intermediate operations, such as the limiting of the rate of growth, is always limited so intermediate calculations don't yield ridiculous results.

3.5.2.2 MBC TCP Vegas

The DA-LBE Vegas metacontroller is almost identical to the one described in algorithm 3, with the notable addition of a growth limiter for φ. The growth in w for DA-LBE Vegas is the same as the weight growth limiter in DA-LBE Cubic, but the φ growth limit is set to 1 % of the new value. We want φ to be a fairly stable value, as it is a factor in the final value used for adjusting the Vegas queuing delay and can therefore greatly impact the final DA-LBE Vegas behavior.

3.5.3 Debugging Libdalbe

Correct behavior of the sample DA-LBE metacontroller implementations required extensive debugging to get right. For this purpose we made the metacontrollers output all of their registered statistics and calculated socket parameters for each interval. This information is used in chapter 5 to correlate DA-LBE statistics with flow features seen in the simple throughput graphs, helping explain the behavior of Libdalbe.

Figure 3.3 shows an example of the debug graphs for a DA-LBE flow[2]. Each graph represents separate statistics, with related values shown in the same graph where the scales allow.

- Figure 3.3a shows the metacontroller weight parameter on the left axis and the probability of triggering phantom ECNs on the right. The phantom ECN probability is inversely related to the weight; a higher weight indicates that the TCP should send more aggressively (less LBE-like), which is achieved by lowering the phantom ECN probability.

- Figure 3.3b shows the two network prices that the DA-LBE Cubic metacontroller utilizes to achieve DA-LBE behavior. The line labeled "Measured" is the measured price of congestion on the network; a higher measured price indicates there is more congestion. The "Model" network price is an estimation of the network price that will allow

[2] This set of graphs is generated from the run described in section 5.1.1, though used here only as an example. Analysis follows in chapter 5.

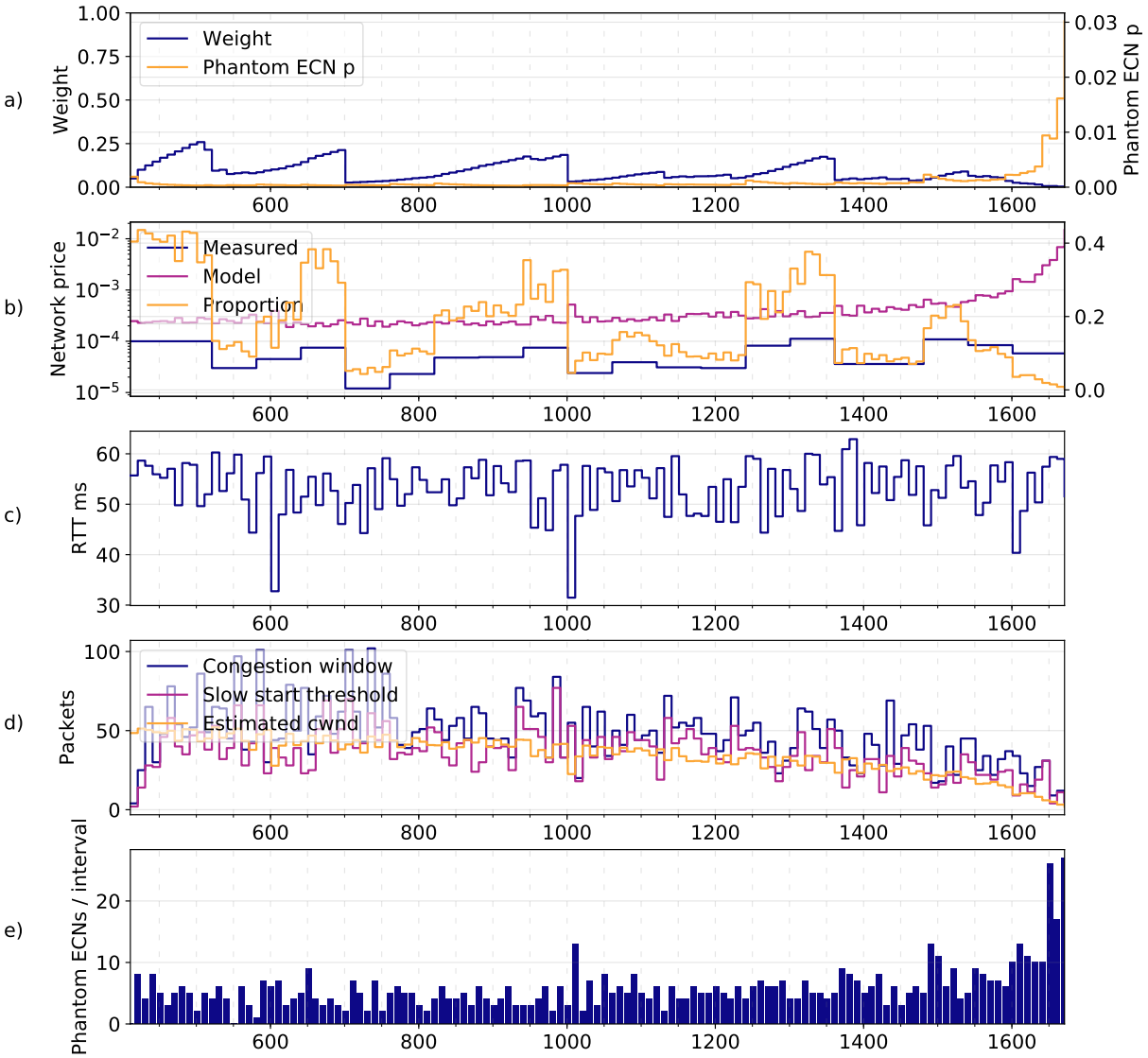


Figure 3.3: Debug graphs for Libdalbe DA-LBE Cubic, containing metacontroller statistics for each interval the flow was active.

Libdalbe to achieve its target rate. The proportion between these two, plotted on the right axis, is used to derive the weight[3].

- Figure 3.3c shows the registered RTT, read from the underlying TCP socket. The value is a slightly smoothed running average. This RTT reading is available through default socket operations.
- Figure 3.3d shows the congestion window and slow start threshold of the TCP congestion controller in packets. The line labeled "Estimated cwnd" represents the estimated congestion window used to derive the model network price in figure 3.3b.
- Figure 3.3e shows the amount of phantom ECNs triggered from each interval. This value can be approximated as the product of the current sending rate and the set phantom ECN probability; plotting it allows us to ensure that the phantom ECN probability is set and acting correctly.

3.5.4 Error Handling

Libdalbe handles errors that occur while performing socket related activities as well as errors resulting from the user supplying invalid parameters. Out-of-memory errors are currently *not* handled given the simplicity of the library and the magnitude of the heap memory used.

All library functions return a negative value on error, and an error message is copied into an optional pointer provided by the user. Where applicable, the negative return value is the one returned from the invocation of a system call or standard library function as part of the standard library operation. This should help experienced users in debugging unexpected behavior.

3.6 Changes in the Kernel

Although not originally part of this thesis, some elements of the DA-LBE kernel modifications did not work as they should have and needed to be fixed before proper DA-LBE operation could be achieved. The fixes are mostly workarounds to have the DA-LBE kernel component function just well enough for our purposes[4].

3.6.1 Generation of Phantom ECN Signals

The models that employ phantom ECNs as a price adjustment scheme are predicated on the probability of triggering an ECN on *every* ACK. In the DA-LBE kernel component this was not implemented as specified in its documentation; the phantom ECN calculation was being done on every ACK only if the slow path of the acknowledgement handling code was entered. This caused ECNs to only trigger on certain non-typical ACKs, resulting in a much higher sending rate than the metacontroller was aiming for. Fixing this issue required ensuring phantom ECNs could also be generated on the TCP fast path.

Additionally, there was a bug in how the phantom ECN probability scaled for ACKs that acknowledged multiple segments. In such cases it is mathematically sound to multiply the probability by the amount of segments being ACKed; however, in the slow path case,

[3] Together with the "error", the proportion between the target and actual sending rates. Neither the error, the target rate, nor the measured rate are shown in this set of debug graphs. The debug graphs used in chapter 5 are presented together with the throughput graph, a context in which rates are not needed as extra debug information.

[4] Supervisor David Hayes helped in the identification and fixing of the bugs mentioned here.

ACKs may occasionally acknowledge no packets at all. Acknowledgements for zero packets, probably DupACKs, are still valid signals of congestion and must generate phantom ECNs; the fix was a simple check to set a minimum of one packet ACKed for the probability calculations.

3.6.2 Calculation of EWMA

The phantom ECN adjustment mechanism requires tracking the average time between congestion events, used to aid in leveraging free bandwidth in the absence of congestion. For this purpose the DA-LBE kernel component averages the delta between each received ack using an EWMA with a weight specified by the user. The implementation had three errors, two of which impacted our DA-LBE mechanisms:

1. The calculation requires calculating the difference between two unsigned integers. This *diff*-variable was defined as also being of an unsigned type which, for the case where the new value to be averaged is smaller than the existing value, means that the difference variable underflowed. The error manifested itself as an oscillation in throughput, as the phantom ECN machinery would routinely be turned off given massive EWMA values. We made all relevant variables signed.

2. Milliseconds were being employed for the calculation of the moving average; for our use case this is not granular enough. In testing we found that the rounding error introduced by fixed point arithmetic proved too significant for the time scales required, where calculating the difference between the current average and the new value results in single digits of milliseconds or less. We made the calculations use nanoseconds, which will be valid for as long as the timestamp can be represented in 2^{63} bits.

3. Finally, there was a faulty check for whether the user had set a custom weight for the EWMA calculation. The check was for an unsigned variable being either zero or not-zero, which will always be true. We removed the check, but added a check to skip calculations if the weight is ever set to zero; in this case the average value never changes.

3.6.3 Inflation of Queuing Delay

Another unsigned underflow error was found in the mechanism for adjusting perceived queuing delay in delay-based controls. In addition, the calculation was inverted in terms of the set congestion price: A high congestion price adjustment, μ values greater than 1.0, should lower perceived delays in the underlying TCP, but this was initially not the case and we had to flip the calculation.

3.7 Shortcomings

In this section we describe some weaknesses in libdalbe and features that could not be implemented.

3.7.1 Locks and Blocking

Multithreaded code brings with it a multitude of complications. Concurrent access to shared state, potential race conditions, and signal handling are but a few possible problems one must tackle. Libdalbe is conservative in its use of multithreaded functionality. It does not utilize

signals, application programmers have to implement their custom metacontrollers as callback functions, and all shared state is tightly locked. Still, some threading problems persist in the final implementation.

Information about each opened socket is stored in a linked list which is shared with the daemon. The list is sorted in chronological order based on the time until the next metacontrol update; the daemon only ever has to sleep for the next socket in the list. The time interval is hard-coded, resulting in a minimalistic update procedure: Given that the time until the new socket must be worked on is *always* 10 seconds in the future, the new socket can always be put after the socket with the current longest wait time. This solution requires no magic tricks involving signals or polling in the daemon to check for new sockets as the new socket will never need updating before the current sleep cycle is finished.

Given the simplicity of the socket list, the daemon sleep mechanism, and not making use of signals, Libdalbe can not wake the daemon once it is sleeping in wait of a socket update. Consequently, disposing of Libdalbe takes up to ten seconds as the main thread has to wait for the daemon to wake in order to terminate it gracefully[5]. For our use case this did not pose a major problem and time constraints towards the end of the project have led to it never receiving further attention. The ideal fix for having to wait for the daemon is waking it using signals — which brings with it significant problems — or sleeping in a trivial loop of e.g. 1 second at a time until the socket is ready for operation, checking for the shutdown condition on each wake.

Another potentially major problem is inconsistent state reported from the DA-LBE kernel component. Sockets should be safe for multithreaded use[6], but exporting data to user space from different threads might yield incorrect data. In an attempt to mitigate such a problem, should it exist, Libdalbe only ever performs socket operations in the daemon thread.

3.7.2 Signals and Custom Intervals

Signals are a way for applications to send each other or themselves notifications; by way of the `kill` system call for signalling a process or process group or, more relevant in the case of Libdalbe, `pthread_kill` to signal a thread in the same process. An application may register a *disposition*, a function to call or a default action to perform upon receiving a signal; this disposition is active for all threads in the process. Dispositions can be ignored on a per-thread basis, but one cannot define multiple dispositions for one signal.

Signals can interrupt threads inside blocking system calls, meaning a thread blocking on a sleep call will be woken up to handle the signal. They are a possible solution for the problem of waking the daemon either for graceful shutdown or for operating on a newly added socket. For Libdalbe to be able to support custom update intervals the daemon must be able to dynamically wake from sleep to handle new sockets.

Linux defines up to 32 signals, not all of which support custom handling [35, `signal(7)`]. Two signals are open for developers to define for their own use, as well as up to 32 *realtime signals* (depending on standard library implementation) as defined by POSIX [27, sec. 2.4.2]. Given that there is a fixed limit on active signals and that each signal can have only one disposition, library developers should not claim signals as their own; they cannot be certain the signal is not in use by the user of the library.

A solution fit for production use should use an existing, extensively tested, library for time-keeping purposes. A possible candidate would have been Libuv [39]; a "support library

[5]The daemon is shut down gracefully by *joining* the thread.

[6] As specified by POSIX [27, sec. 2.9.1], in which neither `getsockopt` nor `setsockopt` are explicitly listed as *thread*-un*safe*.

with a focus on asynchronous I/O" which supports firing events on a timer, socket updates in the case of Libdalbe.

3.8 Summary

This chapter introduced Libdalbe. Libdalbe is a C library that gives access to DA-LBE control mechanisms in the kernel, and facilities interacting with these on a per-flow basis through a metacontroller. Application developers can utilize sockets opened through Libdalbe the same way they would standard Berkeley sockets in Linux, and the API is structured such that converting from using standard sockets to using Libdalbe sockets requires minimal effort. A daemon thread updates flow statistics on each socket and performs metacontrol operations as defined by the given metacontroller. Default metacontrollers are provided which implement a DA-LBE transport service using either TCP Cubic or TCP Vegas.

Chapter 4

Testing Setup

In this chapter we describe the environment in which our tests are run, and how we orchestrated the testing of Libdalbe. The goal in testing Libdalbe is verifying that the library interface works as described in chapter 3, with focus on the interaction with the DA-LBE kernel component. Verifying that the library performs correctly is done through testing that the sample metacontroller implementations behave similarly to the metacontrollers described by Hayes *et al.* [24].

We test on two setups: A local test bed with an emulated dumbbell setup, and over a connection to Cork, Ireland over the Internet. The former is described in detail in section 4.1 and the latter is described in section 4.3. The local test bed is a controlled environment in which we can perform rapid testing during development to identify and fix bugs and finally confirming the correct operation of Libdalbe. The setup consists of five computers, with separate network interfaces for management and testing, networked together with four of them acting as edge nodes and one emulating the delays and bottlenecks of a real larger network. Testing over the Internet will prove whether Libdalbe performs as expected in a real environment.

4.1 Test Bed Setup

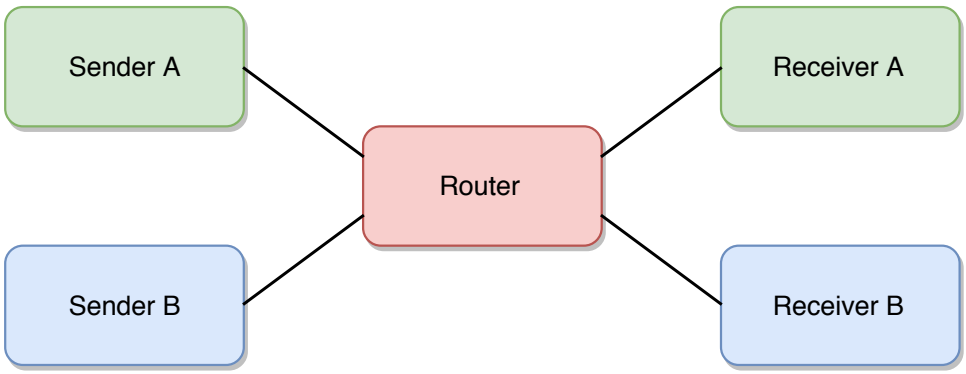


Figure 4.1: Simplified overview of the connections in the physical test bed setup.

The network situation we are targeting is an approximation of a real-world network, in which hosts are communicating with each other using a standard loss-based TCP while we are experimenting with new congestion controllers on a pair of nodes. A good configuration

for this kind of test scenario is that of the dumbbell topology, in which two networks are connected by routers and the nodes in each network share the same router. When communicating across the networks, all of the nodes in each network must share the resources of the one connecting link. In testing, this one connecting link is used to shape traffic flowing between the networks, so as to emulate the behavior of a real network.

As shown in figure 4.1, our test bed consists of two pairs of nodes, pair A and pair B. These nodes will be communicating through different means depending on the configuration for each test. Nodes in pair A will be communicating solely through communication channels opened through Libdalbe, while the nodes of pair B primarily communicate using BE Cubic. Pair B is responsible for all background traffic, including up to several TCP flows, and additionally random background noise meant to alleviate the potential effects of global synchronization.

All nodes are directly connected to the router using 1 Gbit network links with negligible propagation delay. For the final test set up, rate limiters are set up and a propagation delay is added to emulate a real network. This is further described in section 4.2.

4.1.1 Setup specifics

An exact description of the physical equipment and operating system setup for the test bed is found in tables 4.1 and 4.2.

4.1.1.1 Test Bed Hardware

	Processor specification				
Node	Model	Frequency	Memory	Storage media	Network interface
Sender A	AMD Athlon 64 X2 4800+	2.50 GHz	2 GB	Seagate ST3320620AS	Broadcom BCM5722
Sender B	AMD Athlon 64 X2 4200+	2.20 GHz	4 GB	WDC WD1600JS-60M	Broadcom BCM5722
Receiver A	Intel Core 2 Duo E4600	2.40 GHz	4 GB	Seagate ST3250310AS	Intel 82571EB
Receiver B	Intel Core 2 Duo E8400	3.00 GHz	2 GB	Seagate ST3250310AS	Broadcom BCM5722
Router	Intel Core 2 Duo E6750	2.66 GHz	4 GB	Seagate ST3320620AS	Intel 82571EB

Table 4.1: Test bed hardware specifications.

There was a real concern that both the processors and the hard drives present in the sender nodes, as shown in table 4.1, were not powerful enough for our experiments. In testing, however, both of the nodes proved powerful enough to generate and send test data as well as capturing packet data on the fly. Compression is employed to ensure data gathering is not limited by slow hard drives.

The network interfaces named in table 4.1 are the ones connecting the test network, as shown in figure 4.1. A separate set of network interfaces connects the test bed machines to a management server, through which the test bed can access the Internet.

4.1.1.2 Test Bed Operating Systems

Table 4.2 shows the operating system set up used for the test bed. All machines were first set up with Debian 9, after which the DA-LBE sender, sender A, was reconfigured with a custom kernel containing the DA-LBE kernel component and the router was reconfigured for better rate limiter performance, the latter of which is further elaborated on in sections 4.1.2

	Operating system			
Node	Distribution	Kernel release	Kernel version	Modifications
Sender A	Debian 9	4.13.0-rc1	net-next [41, commit 736b9b9c506e]	DA-LBE kernel compenent
Sender B	Debian 9	4.9.0-5	Debian 4.9.65-3+deb9u2 (2018-01-04)	
Receiver A	Debian 9	4.9.0-6	Debian 4.9.82-1+deb9u3 (2018-03-02)	
Receiver B	Debian 9	4.9.0-5	Debian 4.9.65-3+deb9u2 (2018-01-04)	
Router	Debian 9	4.9.65		1 kHz, no dynticks

Table 4.2: Test bed operating systems and kernel specifics.

and 4.2.3. The custom DA-LBE kernel, elaborated on in section 4.1.2, present on Sender A is based on a branch of the Linux kernel containing future changes to the network stack.

The exact kernel version for the router node was not registered before recompiling with custom options. Given the reported kernel release, it should be safe to assume that the kernel, excepting custom options, matches that of the B nodes. Receiver A was updated once in conjunction with debugging rate limiter performance and is on a slightly newer version of Debian.

4.1.2 Kernel Modifications

Both the sender of pair A and the router require modifications to the kernel for proper operation. The sender requires a kernel containing the DA-LBE kernel component; a set of modifications based off of the *net-next*[1][41] Linux kernel branch. The router, however, requires more subtle changes.

A higher *tick rate* is required for rate limiters to function properly on Linux. The tick rate is the rate at which the operating system will wake to perform administrative tasks; for some rate limiters, such as Hierarchical Token Bucket (HTB) [13], the tick rate governs how often packets can be dequeued for sending. Linux distributions meant for use in desktop computers, such as the Debian Linux we are using, are typically using a 100 Hz or 300 Hz tick rate by default. Additionally they may be configured to use *dynticks*, a timer mode in which the kernel is not woken for administrative tasks during idle periods to limit power usage.

The Linux manual page for HTB [35, `tc-htb(8)`] notes that in order to limit the sending rate to an average of 10 Mbit on a kernel with a tick rate of 100 Hz, a *burst* allowance of 12 kB is required; meaning packets are allowed to be dequeued for sending 12 kB at a time. Behavior of this kind does not occur on a physical link of the same maximum rate. Given that the network links connecting the test bed support speeds of 1 Gbit and the rate is capped at 100 Mbit, detailed in section 4.2, packets may in bursts be delivered at up to ten times the speed which would have been physically possible on a real setup with 100 Mbit network links. This burstiness is thus a potential problem for both test accuracy and the stability of the DA-LBE kernel component and must be diminished where possible. The kernel on the router machine has been configured to tick at a rate of 1 kHz, the maximum configurable tick rate in Linux [35, `time(7)`], and dynticks have been turned off.

[1]Net-next contains changes to the network stack, often experimental, that have yet to make it into the main Linux kernel.

4.1.3 Testing with a Known Environment

Linux employs several techniques to save power or increase the perceived responsiveness of the system, some of which may have an impact on the integrity of our tests. It is important that the tests are reproducible, to the extent that a network test allows; for tests on the Internet this is impossible, but the test bed can be tuned so that the environment is constant between tests. Following is a list of the parameters that need to be set for every test to operate in the same environment.

- **Pause frames** [25] are a type of Ethernet frame (see section 2.1.1) that a node can send to signal that it is overwhelmed. Included in the message is a count stating for how long to pause. Should the networking equipment used in testing send pause frames, we might get spikes in packets that would not be equal for every test run. To the best of our knowledge, the network interfaces employed in our test bed do not support pause frames, but they are explicitly turned off in case our reporting tools return wrongful information.

- **Interrupt coalescing** [43] is the act of holding off on an interrupt, such as the notification of data arriving, until more arrive in an effort to reduce system load. Coalesced interrupts might, as pause frames, introduce burstiness that could be hard to account for or reproduce. This mechanism can be toggled on a per-interface basis and is turned off for all interfaces used in testing.

- **Segment offloading** [75, `networking/segmentation-offloads.txt`] is a mechanism for the operating system to hand off unnecessary work to the network hardware. If a user tries to send a large chunk of data the operating system would need to split the data into segments of a size suitable for the network; segment offloading hands this work off to the network card to save on CPU resources. Conversely, the network interfaces can also pack segments back together for the kernel on the receiver.

  Segment offloading can alter the timing of packets logged using our packet capture tools and may cause unpredictable or unreproducible behavior. The mechanism can be toggled on a per-interface basis and is turned off for all interfaces used in testing.

- **`tcp_no_metrics_save`** [35, `ip-tcp_metrics(8)`] is a parameter for the Linux TCP machinery to cache information about previous TCP flows to each destination host. The cache keeps metrics that allow for a faster handshake and information about the route such as average RTT, RTT variance, reordering on the route, and the previous `ssthresh`. The `tcp_no_metrics_save` parameter is system-wide and is disabled using a file in the `/proc` file system on all test bed nodes.

- **Network Time Protocol (NTP)** [55]–[58] keeps the system clocks of the test bed nodes synchronized. NTP gradually skews the system clock into synchronization with an Internet NTP server. System clocks must stay in sync so that packet data captured on separate machines can be merged; running an NTP service while testing may render the captured data worthless. Consequently, we turn NTP on between test runs to keep the clocks synchronized and turn it off during testing.

4.2 Emulation

Ideally we want our tests to run on a real dumbbell test bed setup. Only five machines are at our disposal; we must make the best of what is available. Emulation must be employed to

achieve the desired properties of a dumbbell topology that are missing in the physical setup.

For testing Libdalbe we want an RTT approximating that of an arbitrarily defined *typical* data transfer. A round trip time of 30 ms is chosen, representing a transfer between Oslo and somewhere in mainland Europe[2]. The capacity of the link should be representative of what could be considered a *good Internet connection*[3] but must also be low enough for the test bed machines to record all data without loss; 100 Mbit is arbitrarily chosen as a good bandwidth limit. The router's buffer size should be equal to the Bandwidth Delay Product (BDP), which should be large enough to allow TCP to achieve full bandwidth without being excessively large: $100\,\text{Mbit/s} \times 30\,\text{ms} = 250$ packets.

4.2.1 The Virtual Setup

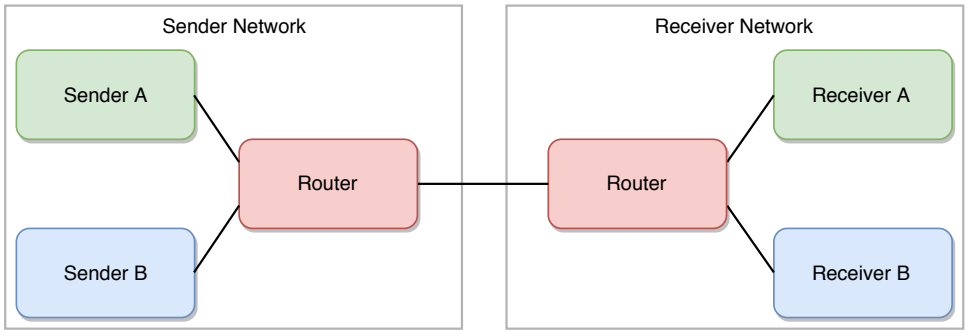


Figure 4.2: Simplified overview of the virtual test bed setup. Nodes in each separate network are connected to their own router which is in turn connected to the router of the other network.

Figure 4.2 shows the emulated network setup of the test bed. Edge nodes, the senders and receivers, are connected directly to their local router by 1 Gbit network links. The routers themselves are connected by a 100 Mbit network link with a propagation delay of 15 ms.

Due to a quirk in the network emulation, described in section 4.2.2, traffic between nodes in the same emulated network is delayed in the same fashion as the inter-network link. Consequently nodes cannot, for testing purposes, be considered to be in the same network. None of our tests depend on traffic between nodes in the same network; this traffic being delayed is not detrimental.

4.2.2 Router Node Emulation Specifics

All emulation is done in the test bed node termed the *router node*, the node to which the other four machines are connected. An overview over the emulation setup on the router node is shown in figure 4.3.

Linux offers software routing facilities through the Iproute2 [74, `networking/iproute2`][35, `ip(8)`] package, but additional measures were needed for our special use case of selectively sending traffic to a shared queue. The shared queues are set up as virtual network interfaces using *IFBs* [74, `networking/ifb`]. Network traffic from the nodes is filtered to one of two IFBs based on the source and destination network; the two virtual interfaces each handle traffic going in one direction between networks. Each of the virtual interfaces is equipped with

[2]The RTT to `google.de` as measured from Fornebu, Akershus, Norway is roughly 28.8 ms.

[3]Telenor, a Norwegian ISP, advertises speeds of 100 Mbit/s (bidirectional) as suitable for multiple users. The 250 Mbit/s package is targeted at "families that need extra network speed" [17].

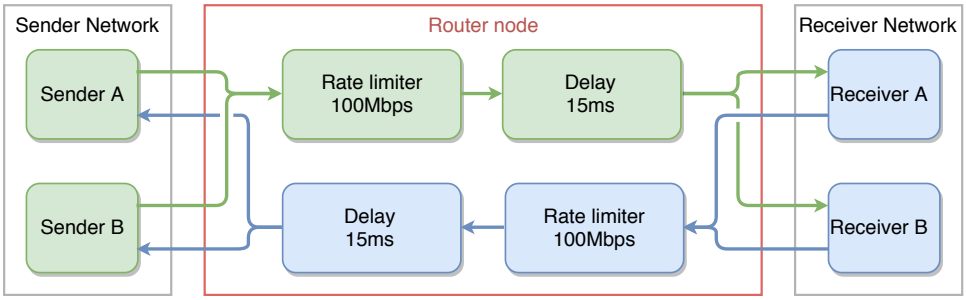


Figure 4.3: How the network is emulated in the router. Messages that cross the network boundary share a bottleneck with other traffic from the source network, as denoted by the different colors for each network and their paths through the router. Note that the colors in this figure denotes a different grouping than that of figures 4.1 and 4.2.

a rate limiter and a fixed size buffer housing a FIFO (tail drop) queue. The addition of propagation delay is done on the physical interfaces connecting the edge nodes using Netem [74, `networking/netem`][35, `tc-netem(8)`]. Delay is added for outgoing traffic to all nodes regardless of filtering as Netem showed inconsistent performance when applied only to the virtual interfaces, described in section 4.2.5.

4.2.3 Rate Limiters

We are aiming to perform testing on a network with a virtualized 100 Mbit network link. The underlying hardware could not be set to operate in native 100 Mbit mode[4]which might have been the best solution for rate limiting. The rate limit must be done through rate limiting software.

Rate limiting in Linux is often done through the use of *token bucket* algorithms such as Token Bucket Filter (TBF) [35, `tc-tbf(8)`] or HTB [13], [35, `tc-htb(8)`]. Token bucket algorithms work by having each packet to be sent *pay* for its size in tokens, spending one token for each byte. A packet FIFO queue is kept, along with a queue for tokens called the *token bucket*. The token bucket is replenished at a set rate, resulting in packets being dequeued for sending at the same rate as tokens are refilled.

The size of the token bucket, typically called *burst* in the documentation, determines how many tokens can be spent instantaneously should an incoming packet encounter an empty packet queue. TBF and HTB both replenish tokens based on the kernel tick rate; to achieve the desired sending rate the burst parameter has to be tuned so that enough bytes can be dequeued for sending every tick. These algorithms are inherently bursty, depending on the interval at which packets can be dequeued.

Another more advanced alternative for rate limiting is that of Hierarchical Fair Service Curve (HFSC) [69], [35, `tc-hfsc(7)`]. HFSC allows giving different service to different classes of data: Data classified as *realtime* are given bandwidth and delay guarantees, where excess bandwidth is distributed evenly among the remaining classes. HFSC is more complex than warranted by the simple purpose of applying a classless rate limit to all traffic. The exact method by which HFSC dequeues packets is different from the token buckets; though we cannot claim to know exactly how this is handled, it can be shown that these differences

[4] The interfaces list `100baseT/Full` as a valid mode of operation, "Full" specifying full-duplex. When attempting to activate this mode the interfaces would fall back to half-duplex operation.

exhibit more favorable characteristics for correct emulation. The characteristics of each rate limiter are described in section 4.2.4.

4.2.4 Determining the Best Rate Limiter

Working with a 100 Mbit rate limit on network links supporting 1 Gbit is not a perfect representation of a real slower link. The documentation for both of the token bucket algorithms is clear on the fact that they need to dequeue packets in bursts. It would seem that the Linux Traffic Control (TC) network in general cannot operate faster than the kernel tick rate.

HTB is chosen as the baseline rate limiter. We test HFSC against the baseline, along with different options where applicable. For our use case, all classless traffic, the options for HFSC are limited.

HTB, like HFSC, has options allowing for applying different rates to different classes of traffic. Most of its options are meant for the cases where excess bandwidth can be split amongst subclasses of traffic, but it is not clear how these same options operate when only working with one class of traffic. We set *rate* to 100 Mbit, the maximum allowed transfer rate; in practice the algorithm will target an average of this rate.

The two options of real interest are `burst` and `cburst`; both of which govern the size of their respective token bucket. The `burst` parameter allows child nodes to temporarily send as fast as their parent, primarily meant for use with multiple classes where subclasses must share bandwidth. `cburst` however sets the size of a token bucket whose tokens allow sending at infinite speeds, essentially the maximum speed of the network interface. Exactly how these parameters work when set together is not thoroughly documented.

Another important consideration is the source of our packet timestamps. The network interface hardware listed in table 4.1 does not provide hardware timestamped packets (or the kernel driver does not support them), so the timestamps we use in our tests are gathered from the Linux kernel which, as mentioned in section 4.1.2, are affected by the kernel tick rate. Gathered timestamps being affected by the kernel tick rate probably affects the test results in table 4.3 significantly, but for our test bed there is no way to remedy this issue. Better hardware, or driver support, is needed for more accurate results in future work.

4.2.4.1 Burstiness

We set up a test to compare different combinations of parameters for HTB and compare them against HFSC. The test consists of one sender transferring data to one receiver in the other network, crossing the router node and its rate limiters. The test is run ten times, with each run lasting five minutes.

We define a *burstiness*-metric by which we will judge the rate limiters: For a bit rate of 100 Mbit/s and a Maximum Transmission Unit (MTU) of 1500, the average time Δ between packets arriving should be 120 µs. A packet is considered *bursty* if it arrives after less than Δ time has passed since the last one was received, meaning it arrived faster than physically possible on a real 100 Mbit connection. A bursty packet and its immediate predecessor are part of a *run* of bursty packets. Finally, we define runs of bursty packets of length 3 or more to be a *burst* of packets.

Figure 4.4 shows how packets arrive on a rate limited 1 Gbit/s link. The green packets and arrows represent perfect spacing between packets, theoretically leading to an even 100 Mbit/s rate as exemplified by the wholly green flow of packets on the second line of the figure. The purple packets and arrows are arriving slower than the perfect 120 µs limit, but this is to be

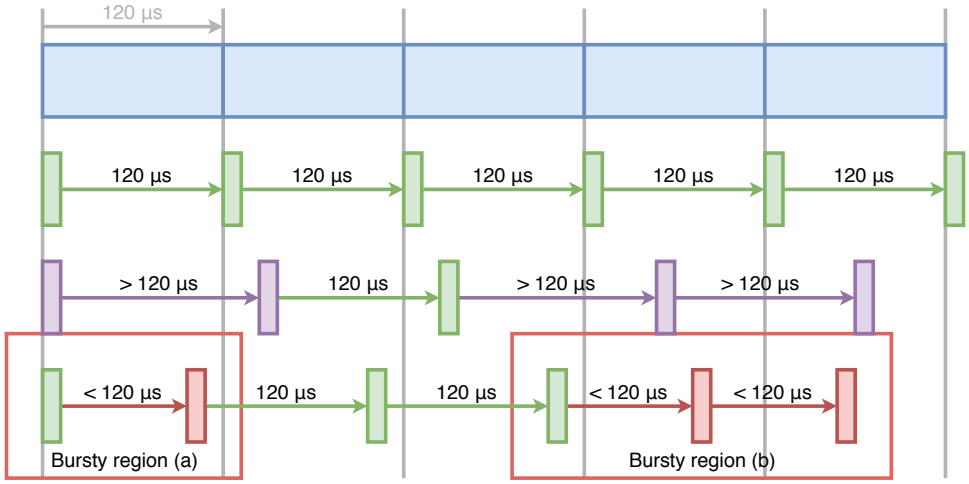


Figure 4.4: Example of burstiness. The blue packets show how packets would arrive over a real 100 Mbit/s link, where one packet takes 120 µs to receive and packets immediately follow each other. The other packets are examples of a rate limiter being applied to a 1 Gbit/s link, where packets take 12 µs to arrive and artificial pacing of packets enforces the 100 Mbit/s limit. The grey vertical lines are 120 µs-markers. Note that the times listed on the arrows represent the time between each packet being fully received, thus the 12 µs needed to actually receive the packet are not counted.

expected when the link is not utilized at full capacity. The red packets are *bursty*, with the red arrows denoting a too-short interval between the packets arriving. The two bursty regions are runs of packets that arrive too quickly, including the immediately preceding packet. The first bursty region, figure 4.4a, is not a burst because it only consists of two packets. Figure 4.4b, the second bursty region, is an actual burst because it spans three packets. All of the red packets in figure 4.4 would be marked as bursty, but only the second bursty group is counted as a burst.

4.2.4.2 Rate Limiter Burstiness Test Analysis

For HTB, the `burst` parameter is not expected to show differing behavior from the norm, but we expect the `cburst` parameter to affect the ability of HTB to achieve and enforce the set sending rate. With `cburst` allowing bursts of infinite speed however, it is also expected that this parameter leads to more recorded bursts as per our definition. How HFSC fares we cannot guess at in advance, not knowing its internals.

Table 4.3 shows results from the burstiness tests with the parameters in use for each test. First and foremost, HFSC along with HTB with the `cburst` parameter set are the configurations that achieve actual sending rates of close to 100 Mbit/s. This points to the remaining configurations adhering more strictly to the target rate in that packets are rarely allowed to be sent too quickly, which is also reflected in the percentage of bursty packets recorded. The sending rate for the HTB configuration with only `burst` set to 50000 and the one with no bursting allowed at all are both sending at an average of 98 Mbit/s [5], almost a whole megabit slower than the other configurations. TCP is not expected to average 100 % bandwidth usage, but the configurations with the lower sending rate seem to be sending

		Packets		Bursts			
Limiter	Options	Count	Bursty	Count	Mean	Median	Stddev.
HFSC		24.7 Mpkts	29.6 %	15 197	6.82 pkts	3 pkts	8.68 pkts
HTB	burst: 50000, cburst: 1	24.5 Mpkts	6.7 %	2779	15.11 pkts	25 pkts	11.65 pkts
HTB	burst: 50000, cburst: 50000	24.7 Mpkts	30.2 %	16 435	5.05 pkts	3 pkts	6.52 pkts
HTB	burst: 1, cburst: 50000	24.7 Mpkts	30.5 %	17 118	4.94 pkts	3 pkts	6.33 pkts
HTB	burst: 1, cburst: 1	24.5 Mpkts	6.7 %	2823	15.44 pkts	25 pkts	11.62 pkts

Table 4.3: Results of the ten burstiness tests, with metrics totaled. Note that an HTB parameter setting of 1 means the relevant parameter is disabled, as setting a value of 0 results in HTB choosing the default setting for that parameter. HFSC has no relevant available options.

artificially lowly.

Of the remaining choices, the ones that reach 24.7 Mpkts sent for ten tests, HFSC shows the fewest bursty packets, 29.6 % as opposed to 30.2 % and 30.5 % for HTB, and the fewest bursts total. With ten tests being run for each configuration, we deem these differences in recorded bursty packets to be significant. Though the median burst consists of only 3 packets for all of these configurations, HFSC does cause some longer bursts as revealed by its higher mean burst length and standard deviation. We choose the total number of bursts and bursted packets as the most important metrics for our use case and choose HFSC as our rate limiter.

4.2.5 Netem as a Rate Limiter

Netem [74, `networking/netem`][35, `tc-netem(8)`] is a tool meant to provide network emulation facilities for testing purposes. It can emulate delay, loss, duplication, corruption, re-ordering, and rate control. Netem would ideally go on the two virtual interfaces of the test bed, applying a rate limit and delay for the shared queues. In practice, however, Netem could not manage to consistently uphold the set rate limit, restricting its use to only apply packet delay.

The most natural location to introduce delay would be on the shared network queues, but due to how Linux applies these mechanisms to network interfaces, assigning Netem to the IFBs directly would have made the underlying packet queue, the one for the rate limiter, act as both a regular network buffer and additionally as a buffer for delayed packets. This last requirement of also buffering delayed packets means we cannot set a FIFO queue with a known limit as the number of packets needing to be delayed varies with the sending rate. For the final setup Netem is thus used to delay *all* outgoing traffic regardless of source and destination because it has to be applied on the outgoing physical interfaces of the router. Packet delay being applied for all traffic means we can not run tests for which communication between nodes in the same network, that is sender-to-sender or receiver-to-receiver, is a factor[6].

[5] 24.5 Mpkts/no. tests/no. seconds $\times$ MTU $\times$ 8/1000000 = 98

[6] We are not doing experiments where this is an issue. We can however imagine experiments containing traffic between hosts in the same network, for example they might include testing for the impact of Libdalbe on Internet traffic, with the WAN connection being the bottleneck, when the DA-LBE flow, e.g. from a backup service, is transmitting to another host in the same network at up to gigabit speeds. If the WAN connection in the router is 100 Mbit/s, it could be interesting to see if Libdalbe managed to claim the remaining 900 Mbit/s without negatively affecting the Internet traffic.

4.3 Cork Setup

The other main setup for our tests is one allowing for performing tests over the Internet. Tests done in this environment will show how Libdalbe performs in a more real scenario. With the amount of unpredictable background traffic we cannot draw conclusions about short-term changes in the behavior of our metacontrollers; this will be reflected in the experiments we perform in this environment, see the analysis in section 5.2.

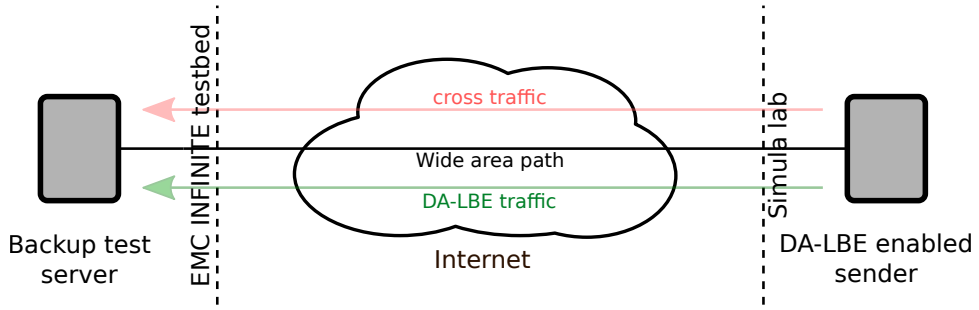


Figure 4.5: WAN topology for the Cork setup [7, figure 12]. Used with permission.

Owing to Dell EMC being a contributor to the NEAT project[7], we have access to a server on their premises in Cork, Ireland for running Internet tests. In contrast to the local test bed, we know nothing about the network connecting the two machines, excepting the gateway machines through which the local sender reaches the Internet. An overview of the Cork setup is shown in figure 4.5.

	Processor specification				
Node	Model	Frequency	Memory	Storage media	Network interface
Sender A	AMD Athlon 64 X2 4800+	2.50 GHz	2 GB	Seagate ST3320620AS	Broadcom BCM5755
Sender B	AMD Athlon 64 X2 4200+	2.20 GHz	4 GB	WDC WD1600JS-60M	Broadcom BCM5755
Cork (receiver)[8]	Intel Xeon E5-2609	2.40 GHz	Sufficient[9]	Sufficient[9]	Intel 82599ES

Table 4.4: Hardware specifications for the Cork setup.

Table 4.4 shows the hardware specifications of the Cork test setup. Note that the network interfaces of the senders are different from the ones noted in table 4.1; the listed network interfaces are the ones connecting to the test bed management machine, through which the sender access the Internet. Simula's Internet link has a capacity of 500 Mbit/s. The exact capacity of the link connecting our Cork machine to the Internet is not known, but we can conclude from testing that it supports rates of up to 500 Mbit/s.

The expected RTT for the Cork setup is measured to be 41 ms or more. At the time of testing there were seventeen network hops between the sender and the receiver, see appendix C.1. It is likely that one or more of these gateways employs an AQM that might affect Libdalbe performance, see section 5.2.1 for some discussion on this.

[7] Thanks to Zdravko Bozakov at Dell EMC.

[8] We did not have access to full hardware specs on the Cork receiver.

[9] The memory and storage hardware was more than sufficient for our purposes, though the exact figures are unknown.

4.4 Test Execution

For testing Libdalbe we were looking for a tool with a specific set of capabilities. Evolved from simple shell scripts used in the beginning of the project, we over time developed a suite of tools to perform the needed tasks. As the set of requirements grew, so too did the tooling.

The tools used for testing the final implementation of Libdalbe have the following capabilities:

- Invoke any procedure for initialization and teardown.
- Call on any program for testing or analysis.
  - The program may be a binary on the target system, one we upload prior to testing, or a script file with a custom way to invoke it.
  - Each program can set a starting delay
  - A duration can be set. The corresponding duration parameter for the application is used if available, otherwise the program is killed.
  - Programs may overlap in duration.
- Automatic gathering of test results.
- Require no user input to function.
- Robust enough to run for hours without intervention

The capabilities of the developed tooling all rose out of requirements of the desired tests. Initialization procedures may involve setting kernel module parameters, operations on network interfaces, and Linux TC [35, `tc(8)`]. Proper testing of any TCP requires analysis of its behavior when competing with up to multiple other TCPs, therefore any test orchestrator must be able to launch overlapping testing applications. Some of the tests are required to be run multiple times to ascertain typical performance; depending on user input in any part of the testing process would inhibit the running of excessively long tests. The test orchestration suite was successfully used to run overnight tests for twelve hours at a time.

4.4.1 Test Orchestrator

The test orchestrator is a Python [45, versions 3.6.5, 3.6.4+, 3.5.2] tool based on the python module Fabric [20], a framework for running and automating system administration tasks. A set of tasks is defined along with remote nodes on which the tasks should be run. Fabric allows for tasks to be run in parallel, where one task is run on multiple nodes concurrently; in our tools each phase of testing — e.g. initialization, testing, teardown — is defined as a parallel task.

The tool is run in a Jupyter Lab [37] environment, allowing for easy visualization and separation of functionality within Python scripts. Jupyter facilitates rapid iteration during development which is valuable when writing software for remote testing; an endeavour with many potential points of failure.

A script designed to run on the nodes, some of which can not run the newest Python version, is copied to the remote node for every test. This node-script uses Plumbum [18] to control the local test processes.

4.4.2 Software on the Test Nodes

The only custom applications on the test nodes are Libdalbe and the scripts put there by the test orchestrator for local control. Gathering of data and generation of background traffic is done by freely available tools.

The BE Cubic flows that will be competing with Libdalbe are generated by Iperf [14], a tool for "active measurements of the maximum achievable bandwidth on IP networks". The only requirement is for the chosen tool to employ the system default TCP for transport, while preferably also allowing for a duration to be set on startup. Iperf provides the required functionality and more; it can output detailed statistics on its own throughput, which can aid in debugging.

With the test bed router employing the tail drop queue management scheme, global synchronization [46] could become an issue. We add random background noise to combat global synchronization, using Distributed Internet Traffic Generator (D-ITG) [4] to send UDP packets of constant size with exponentially distributed inter-packet delay.

We capture all packets generated as part of the running test with Tcpdump [73]. The same software package additionally provides Libpcap, which we employ for packet analysis, see section 4.4.3. The hard drives present in the test bed machines were having trouble writing packet data at the rates required, so Tcpdump data is compressed on the fly using Xz [10] to remove the hard drive bottleneck.

4.4.3 Analysis Software

Analysis of the captured Pcap data, the output format of Tcpdump, is done with a custom C program utilizing Libpcap [73]. Along with the logs generated by our custom tools, this data is collated using Python [45, version 3.6.5] with Jupyter Lab [37] for easy alteration and quick iteration. The data can easily be plotted in several different forms and formats, such as a representation of the internal metrics used in the metacontrollers showcased in figures 5.5 and 5.8, or simply throughput graphs, e.g. figure 5.2.

4.5 Summary

In this chapter we described our efforts in the setup, tooling, and analysis of testing Libdalbe. We set up two environments for testing; one local test bed on which we can control the exact parameters of the test on a single bottleneck, and a connection with Cork, Ireland on which we can test for the general behavior of Libdalbe over the longer term. Our local test bed emulates a dumbbell network topology with two senders and two receivers, using a central router node that adds delay to all traffic and applies a rate limit on all traffic passing between the emulated networks. We did tests of rate limiters to attempt to limit the effects of any artifacting that may occur as a result of rate limiting traffic to 100 Mbit/s on a 1 Gbit/s network link, and we made configuration changes on the router node to alleviate similar issues resulting from kernel functionality. The connection to Cork runs over the Internet on a 500 Mbit/s link with a delay measured to 41 ms, crossing around 17 network hops on the way with unknown configurations. We developed tools for the automatic initialization and running of our tests, as well as automatic capturing of test data and debug information generated by Libdalbe and the metacontrollers. We analyze Libdalbe performance and debug output using a set of tools using a similar environment to that of our automation software.

Chapter 5

Results and Analysis

In this chapter we describe the experiments that we ran and evaluate their outcome and behavior.

We ran experiments on the local test bed described in section 4.1, a controlled environment, to compare Libdalbe performance to that of the simulations carried out by Hayes *et al.* [24]. Our model-based DA-LBE Cubic controller, see section 2.6.3, is implemented exactly as described by Hayes *et al.* in their paper and should show similar behavior to that seen in their simulations. Analysis on the performance of DA-LBE Cubic on the test bed can be found in section 5.1.1. The DA-LBE Vegas controller as described in section 2.6.4 does not work like the implementation by Hayes *et al.* and thus cannot be directly compared. This analysis and comparison can be found in section 5.1.2.

We run simple experiments on the Internet setup described in section 4.3 to check whether Libdalbe performs as expected when facing the large amounts of cross traffic, noise, and possibly active queue managers, found on the Internet. The Internet test runs are analyzed by way of Jain's fairness index as described in section 2.3.1.4 and equation (2.1), for which Libdalbe, being a Less-than Best Effort transport, should consistently achieve a less than fair sending rate given that the deadline permits it. These experiments and their analysis are detailed in section 5.2.

5.1 Test Bed Performance

In [24], Hayes *et al.* describe a simple experiment to test their DA-LBE model. We will see the performance of Libdalbe in competition with both one and multiple BE TCP flows, and additionally how the DA-LBE flow behaves when there is, momentarily, no other traffic.

Table 5.1 defines the test for the local test bed as a set of flows with a given start and end time, visualized in figure 5.1. This is the exact same experiment run by Hayes *et al.* to verify their metacontroller behavior in simulations; their graphs can be found in appendix E for direct comparison. The BE flows, being greedy, will try to send at maximum capacity for their entire duration. The DA-LBE sender will send 1625 MB with a deadline of 1300 s,

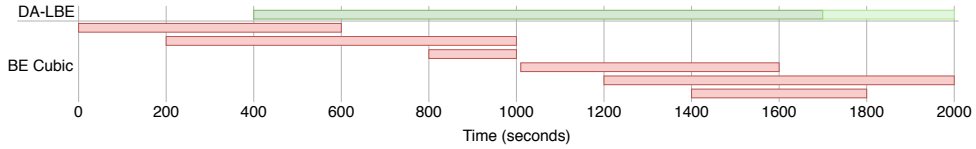


Figure 5.1: GANTT-chart for the experiment defined in table 5.1.

Flow type		Start	End	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 5.1: Test for the local test bed setup, defined as a set of flows with a start time and end time. The flows with a defined end time will halt all sending at that time.

requiring an average sending rate of 10 Mbit/s; it should finish by $t = 1700$ s, though the soft deadline criterion allows it to finish later. In figure 5.1, the possibility of the DA-LBE flow arriving after the deadline is shown in a brighter shade of green. We add 10 Mbit/s of background traffic to combat global synchronization by sending UDP packets of size equal to the MTU at an average rate of 833 pkts/s with the time between each packet sent decided by an exponential distribution. Areas of special interest in this test are the boundary points where a competing flow ends or a new one begins, the heavy traffic and impending deadline at $t \geq 1400$ s, and the 10 s interval starting at $t = 1000$ s in which there is no competing traffic.

The short period with no traffic should show a clear distinction between the two algorithms under testing. While there are mechanisms in place in the DA-LBE Cubic metacontroller to make it greedy in the absence of competition, it will not be able to take full advantage of short periods like the one at $t = 1000$ s. Cubic is limited by its loss signal, as lost packets and ECNs cannot inform about the absence of congestion; the signal itself is only triggered upon congestion. Adding to this, the DA-LBE Cubic metacontroller is not scaling existing signals, but rather inserting new ones proportionally to the sending rate, or more specifically proportionally to the rate of ACKs being received. If we were controlling Cubic through scaling signals, having the real congestion signals dropping to zero would also cause no additional adjustment from the metacontroller, but as we are inserting new congestion signals proportionally to the sending rate, DA-LBE Cubic will not by itself manage to increase its sending rate since the faster it is sending, the faster it is triggering phantom ECNs. DA-LBE Vegas, reacting to queuing delay, should be able to increase its sending rate in these cases because we are scaling an existing signal, not adding new congestion events, and because delay as a congestion signal can notify of there being no congestion.

Less obvious than the 10 s congestion free period at $t = 1000$ s is that a similar effect occurs to some extent every time a competing flow leaves the network. When the first competing BE flow leaves the network at $t = 600$ s, there is an opportunity until the second competing flow catches up where there is extra available bandwidth. We expect that DA-LBE Vegas should show a slightly increased sending rate here; it will immediately react to the lower queuing delays. DA-LBE Cubic is self-limiting in that any increase in sending rate results in an increase in the congestion signal thereby quelling itself through ECNs; it might see a tiny increase in sending rate during this window, but it will in any case be too small to reliably measure.

The period at $t = 1400$ s might cause Libdalbe to miss the deadline, depending on the

amount of data left to send at that point. If the target rate — the average sending rate needed to reach the deadline on time, as explained in section 2.6.2 — is exceedingly low Libdalbe might not allow its congestion controller to become aggressive enough to compete; conversely if the target sending rate is sufficiently high Libdalbe might not be able to reach it while remaining fair, let alone less than fair.

In these tests we are mostly interested in the short-term characteristics exhibited by the different congestion metacontrollers; the controlled environment in which the tests are run allows us to observe and inspect the aforementioned points of interest at will. The Internet tests will let us review the long-term behavior of Libdalbe.

5.1.1 Cubic

TCP Cubic is a purely loss-based TCP which we adapt to DA-LBE behavior by inserting phantom ECNs. The congestion metacontroller described in section 2.6.3 alters the rate at which phantom congestion signals are sent in intervals of 10 s; should the sending rate suddenly increase there will be a matching increase in ECNs, due to metacontroller adjustment, that cannot be changed until the next interval. By the time the DA-LBE Cubic metacontroller is able to react to extra available bandwidth, through detecting low counts of congestion signals in the previous interval, there may already be new competitors, as is the case at $t = 1000$ s in the experiment from table 5.1. Likewise, at $t = 600$ s there is a momentary lull in congestion events that DA-LBE Cubic will not be able to utilize.

We expect DA-LBE Cubic to complete its sending within the deadline by virtue of being equal to its competitors: This assumption hinges on the Cubic model used in the metacontroller to sufficiently model the behavior of the TCP being controlled as well as the competing congestion controls — congestion controls that might vary slightly between different Linux kernel versions.

Figure 5.2 shows our test run from table 5.1 featuring a DA-LBE Cubic metacontroller. The bursty nature of Cubic impedes our ability to distinguish short periods of lower congestion from the typical behavior of the congestion controller.

The DA-LBE Cubic flow seems to be steadily sending slightly in excess of its target rate, even when competing with three other BE flows at $t >= 1400$ s. The slightly exaggerated aggressiveness is consistent with the behavior seen in the MBC throughput graph in the simulations by Hayes *et al.*, see figure E.1c. Their graphs do not explicitly show the DA-LBE target rate, but the continual decrease in sending rate when approaching the deadline reveals that the DA-LBE flow is *ahead of schedule* and does not require an aggressive sending rate to reach its goal. We can conclude from this simple throughput graph that the Libdalbe implementation of the DA-LBE Cubic congestion metacontroller behaves in a similar manner to the simulation results presented by Hayes *et al.* [24].

5.1.1.1 Cubic Metacontroller Analysis

Figure 5.3 contains Libdalbe DA-LBE Cubic debug output from the test run seen in figure 5.2. Interestingly, though we postulated that DA-LBE Cubic would not be able to claim the free bandwidth at $t = 1000$ s or the lower levels of congestion when flows leave the network, it still does show events that significantly lower the aggression. Figure 5.3a shows significant drops in weight, and therefore aggressiveness, at $t = 520$ s, $t = 700$ s, $t = 1000$ s, and $t = 1360$ s. All of these events seem to be correlated with drops in measured network congestion, as given by the line labeled *Measured* in figure 5.3b, which indicates that there was a low number of loss events for the preceding interval.

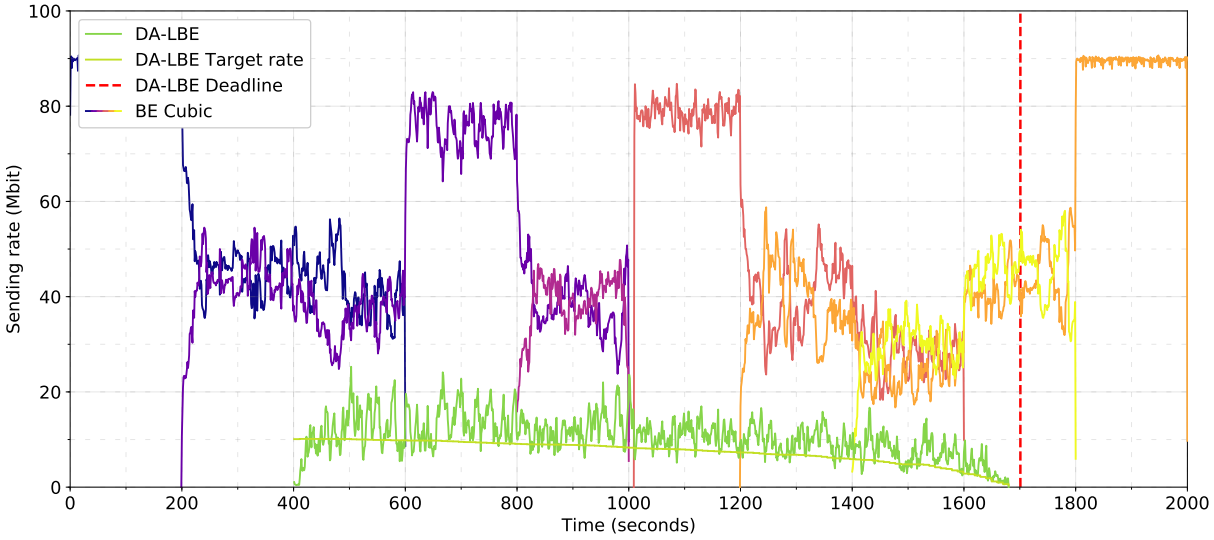


Figure 5.2: One-second averages of bits sent for the test defined in table 5.1, with the DA-LBE Cubic metacontroller, run on the local test bed setup.

However, the drops in measured network congestion seem to be more tied to TCP locking behavior than the points of interest mentioned in section 5.1; the only significant events, visible to the human eye in figure 5.2, immediately preceding any of these drops is the two spikes in DA-LBE Cubic sending rate before the drop at $t = 700\,s$. The drop in network congestion at $t = 1000\,s$ could not have been caused by the short period of no congestion, as by that point the BE Cubic competitors wouldn't have yet ceased sending.

The period of no congestion at $t = 1000\,s$ does show a slight impact on DA-LBE Cubic. Figure 5.3c reveals an RTT measurement almost equal to the BaseRTT, which would have resulted in a low estimated `cwnd` in the Cubic model and thus a higher model price, meaning more throttling. Figure 5.2 shows no significant increase in sending rate for the interval at $t = 1000\,s$, which we understand to be a result of this lowered aggression, in addition to the previously mentioned weakness of DA-LBE Cubic not being able to exploit a momentary lack of congestion.

5.1.2 Vegas

TCP Vegas reacts to both delay and loss. Our metacontroller mainly alters the delay signal, and adjusts the loss signal by ignoring a percentage of loss signals if the DA-LBE w parameter reaches 1.0. If there is no competing traffic on the network, the delay adjustments we make will be negligible and we can expect the TCP to behave as standard TCP Vegas. DA-LBE Vegas should be able to react to changing network conditions much quicker than our DA-LBE Cubic metacontroller. We expect DA-LBE Vegas to react to all flows leaving the network as well as the period of no traffic. Increases in the sending rate like this will likely be followed by a corresponding dip in sending rate as the DA-LBE Vegas metacontroller detects that the congestion on the network has dropped and decreases its w parameter to reflect the network state. Because there is a limit on the growth in w, to promote LBE behavior, DA-LBE Vegas will not be able to compete with the traffic following the spike for a while.

Figure 5.4 shows our test run of table 5.1 featuring a DA-LBE Vegas metacontroller. The α and β parameters are both set to 16 for this test, meaning that Vegas will itself aim to maintain a standing queue of 16 packets. Having $\alpha = \beta$ will give the metacontroller a higher degree

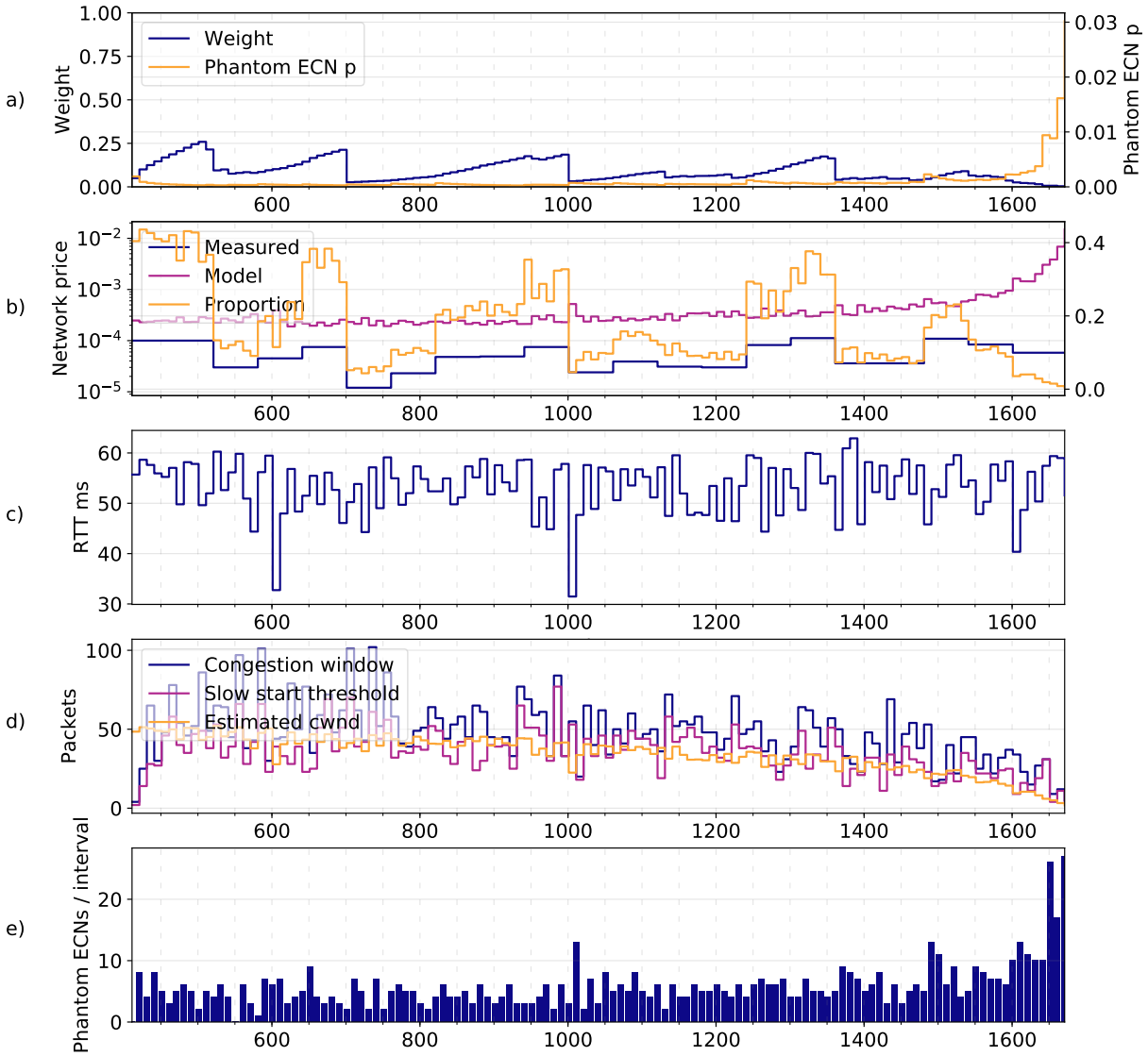


Figure 5.3: Debug graphs for Libdalbe DA-LBE Cubic on the test bed setup, containing metacontroller statistics for each interval the flow was active.

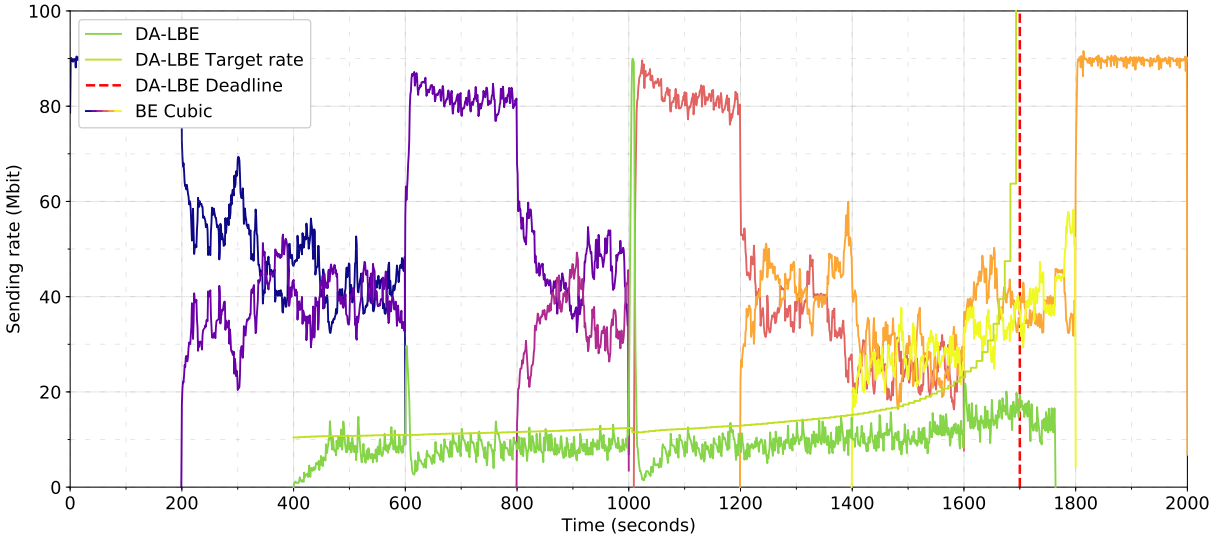


Figure 5.4: One-second averages of bits sent for the test defined in table 5.1, with the DA-LBE Vegas metacontroller, run on the local test bed setup. We set α, $\beta = 16$.

of control as there is no middle zone, as described in section 2.4.2, in which Vegas does not alter its `cwnd` and thus even small changes in the queuing delay adjustment will have an immediate effect. Setting the control parameters this high also scales the model used in the DA-LBE Vegas metacontroller as it reads the Vegas parameters on initialization. Having the congestion control itself be more aggressive means that the metacontroller does not need to alter congestion signals as much.

Contrary to the DA-LBE Cubic metacontroller, DA-LBE Vegas manages to achieve higher sending rates in the short periods of lowered congestion as seen at $t = 600\,s$, $t = 1000\,s$, and to a lesser extent at $t = 1600\,s$. However, we notice that for every significant spike in throughput, there is a corresponding dip immediately following. This behavior of throttling after momentarily having enjoyed lower congestion seems to be detrimental to the benefit of being able to achieve the higher throughput in the first place, though it succeeds in providing a thoroughly LBE-like service. Observe how, after the spike at $t = 1000\,s$, the DA-LBE flow does not reach the target rate before around $t = 1090\,s$, at which point the required target rate has risen to the same level it was before the spike; indicating that the spike provided no real advantage in completion time. Though there was no advantage in completion time, these periods of lower sending rate upon the sudden advent of competing traffic show how DA-LBE Vegas responds in a LBE-like manner when it backs off from a bandwidth monopoly.

5.1.2.1 Vegas Metacontroller Analysis

Figure 5.5 shows the debug output generated by the DA-LBE Vegas metacontroller as values reported in each ten-second interval. In 5.5c we can see how the RTT drops drastically for a short time, apparently corresponding with flows leaving the network at $t = 600\,s$ and $t = 1000\,s$. DA-LBE Vegas, estimating congestion as a function of the queuing delay, is able to achieve a higher throughput in these intervals, clearly visible in figure 5.4.

Figure 5.5a shows the weight dropping drastically following the events at $t = 600\,s$ and $t = 1000\,s$. This drop in weight indicates that the perceived delay is being heavily inflated as a result of the sending rate measuring higher than anticipated. Had the network

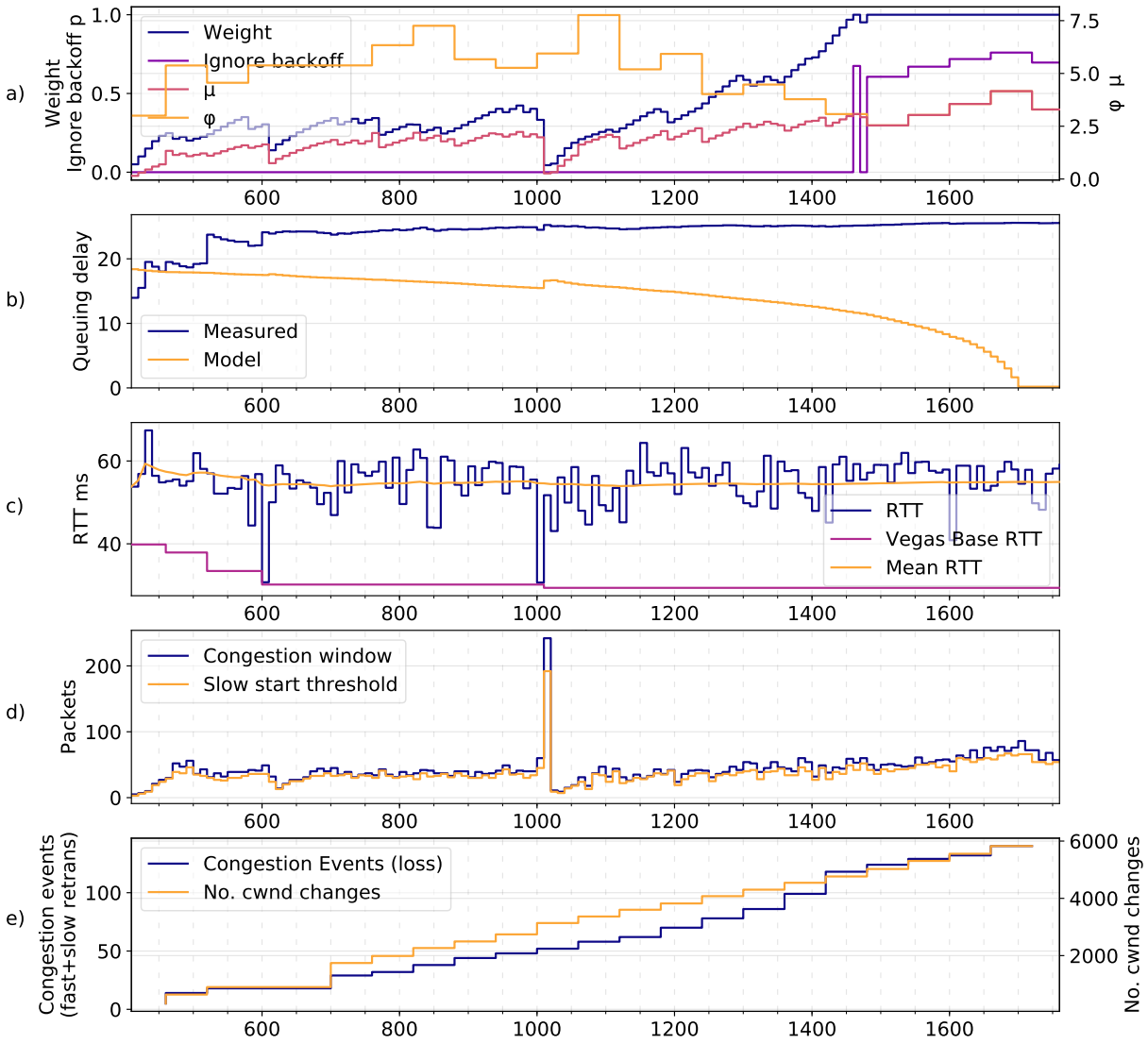


Figure 5.5: Debug graphs for Libdalbe DA-LBE Vegas on the test bed setup, containing metacontroller statistics for each interval the flow was active.

congestion remained low DA-LBE Vegas would still have managed to send with some speed, especially once the network price estimation indicated by φ in figure 5.5a caught up, but the reintroduction of congestion at $t = 1010\,s$ causes the sending rate to fall until the weight is allowed to rise to its previous levels.

5.1.2.2 Accuracy of Measured Queuing Delay

Vegas relies on an accurate estimation of the real propagation delay of the link, the BaseRTT. Should the estimated BaseRTT be too high, a real possibility if Vegas is started on an already congested network, Vegas will perceive the queuing delay to be lower than the actual value, causing the congestion controller to send faster than it should. An estimation that is too low, which could result from a route change, will cause Vegas to send slower than the congestion warrants.

Figure 5.5b shows the measured queuing delay, calculated by $\overline{RTT} - \text{BaseRTT}$. The line labeled *Model* is the target queuing delay for the model to impose on the underlying congestion controller. Note that, as seen in figure 5.5c, DA-LBE Vegas does not have an accurate BaseRTT estimate until $t = 600\,s$ and is not able to infer the actual propagation delay until $t = 1010\,s$. These jumps in the BaseRTT are reflected in the measured queuing delay in figure 5.5b, which also affects the weight seen in figure 5.5a. In a perpetually congested network DA-LBE Vegas might not ever be able to find the real propagation delay and thus can never tune its sending rate properly.

5.1.2.3 Vegas and the Target Rate

In the DA-LBE Vegas run shown in figure 5.4 it is clear that the Libdalbe flow never consistently achieves its target sending rate. Even as the weight in figure 5.5a reaches 1.0, causing 60 % of loss events to be ignored as shown by the line labeled *Ignore backoff*, and having its perceived RTT decreased by a factor of 2.5, DA-LBE Vegas still does not manage to send quickly enough. We suspect that DA-LBE Vegas can never truly compete with Cubic in a tail-drop managed bottleneck like the one in our local test bed environment.

Each Cubic flow traversing the one bottleneck will fill the queue to $> 70\%$ of its capacity, as given by the Cubic's β parameter[1]. Additional Cubic flows on the same bottleneck further increase pressure on the queue. In the case of the local test bed with its 1 BDP buffer size, a 70 % full queue equals a queuing delay of 21 ms. From approximately $t = 1370\,s$ onwards in figure 5.4, the queue is being decreased by a factor of 4.5 to 2.5, meaning DA-LBE Vegas sees queuing delays of at least 4.67 ms to 8.4 ms. Even when seeing only 4.67 ms of extra delay, translating to a queue of 38.89 pkts, DA-LBE Vegas is seeing queuing delays more than *twice* as high as its β allows.

The observations on the effects of queuing delay are corroborated by figure 5.5e, in which the congestion events caused by delay, the line labeled *No. cwnd changes* shown on the right Y axis, outnumber the congestion events from loss by a factor of 40. With the amount of throttling solely from delay-based congestion events, the sending rate is never sufficient to cause significant loss and thus the number of congestion events from loss does not grow.

For DA-LBE Vegas to efficiently compete with any loss-based BE TCP on such a constrained bottleneck, it needs provisions to ignore part of its β-decided delay target. The chance of ignoring backoff on loss, the `cwnd_no_backoff` DA-LBE socket parameter, cannot alleviate Vegas's own delay-based throttling. Setting β to higher values could

[1] Congestion is only signaled when the queue fills completely and a packet is lost, at which point Cubic will scale its `cwnd` by 0.7, meaning the queue will still be filled to 70 % capacity.

potentially alleviate the queuing delay problems in this specific test scenario, but for Internet performance it could make DA-LBE Vegas act unreasonably aggressive, contrary to its, and our, design goal. Another possibility could be altering the φ-calculation in section 2.6.4 to detect and alleviate these situations where there are large standing queues, or changing the queuing delay adjustment to scale down more aggressively during great congestion.

5.1.3 Test Bed Performance Summary

The test bed experiments helped showcase Libdalbe behavior in some very particular situations that we could not have easily achieved on the Internet. We saw DA-LBE Cubic performing well, if a little aggressive, strictly adhering to its target rate in varying degrees of competing traffic. DA-LBE Cubic was, as expected, not able to make use of the $t = 1000\,s$ period of no congestion. Where DA-LBE Cubic was constantly a little more aggressive than needed, DA-LBE Vegas continually sends slightly slower than needed. We believe this behavior is to blame on the very core of Vegas's design, in that targeting a fixed number of standing segments in the queues is simply untenable on a network as congested as our test bed. DA-LBE Vegas was able to send faster in the short periods of lower or no congestion, but its strict adherence to LBE behavior means the extra claimed bandwidth weren't enough to have the transmission complete in time.

5.2 Performance on the Internet

For the tests between Oslo and Cork we cannot easily reason about short-term changes. The tests are run on the Internet; there is no knowing what kind of background traffic there is, or how traffic is being shaped along the route. The neatly generated random background noise of the test bed experiments, though not realistic, was random enough to limit the effects of global synchronization, but for the Internet tests the noise eliminates our ability to explain what is happening in short time frames. We can of course guess at the events behind certain flow features, especially when we can compare the DA-LBE flow and the BE Cubic flow running alongside it.

Due to all competitors adhering to the principle of AIMD — a reasonable assumption given that this behavior is required by RFC 5681 [54] — wherein their increased sending rate implies a similarly increased back-off on loss events, we expect our DA-LBE flows to be able to compete decently and possibly reach their deadlines with fair reliability. Whether they can also remain LBE is difficult to say in advance.

The first Internet tests are identical to the ones run on the local test bed, shown in section 5.2.1 These are done primarily to inspect how DA-LBE performs in real network conditions. The extra background flows and the points of interest outlined in section 5.1 will probably not affect Libdalbe as much as they did in figures 5.2 and 5.4, though it could be interesting to see what kind of effect we *do* see.

5.2.1 Test Bed Comparison

To compare the general behavior between the test setups we ran two tests that are similar to those done on the test bed setup, with the two main differences, apart from running in wholly different domains, being the absence of an emulated rate limiter and the neat background noise of our test bed. While the setup differences prohibit a real apples-to-apples comparison, we can help draw attention to the features and quirks one might see in the Cork tests that are not problems on the local test bed — and vice versa. We do not expect that the

points of interest mentioned in section 5.1 will show significant departure from the typical DA-LBE behavior exhibited in the rest of the flow.

The behavior of competing BE flows in these comparison experiments can not be used to infer Libdalbe behavior as in section 5.1 where significant traffic events in the competing flows could be directly correlated with the behavior of Libdalbe. Under the controlled conditions of the test bed, the competing flows will be the only real influence on Libdalbe, but on the Internet our own competing flows are near indistinguishable from others. Our competing flows can however help us guess at the external network conditions in order to better explain Libdalbe behavior. Figures 5.6 and 5.7 show the experiment from table 5.1 performed on the Cork setup. Note that, owing to a much higher available bandwidth, the Y axis for the Cork experiments is on a logarithmic scale to easier differentiate features in the DA-LBE flow as otherwise the DA-LBE flow would have spanned only a tiny slice of the vertical axis.

5.2.1.1 Cork Cubic

Figure 5.6 shows DA-LBE Cubic on the Cork setup performing similarly to the test bed behavior shown in figure 5.2. The Cork test seems to be exhibiting more extreme behavior; it features sending rate spikes of more than 2.5 times that of the target, e.g. as seen at $t = 490\,s$ and $t = 790\,s$, and frequent dips by a factor of up to 5. The dips in Libdalbe throughput show a slight correlation with dips in the competing BE traffic, as seen at $t = 950\,s$ and $t = 1540\,s$. Dips in both Libdalbe and BE throughput are probably indicators of sudden bursts of congestion in the network. As in figure 5.2, there is no obvious reaction to flows leaving the network.

DA-LBE Cubic does on both setups seem to consistently send in slight excess of its target rate and reach its deadline by a significant margin. More extreme behavior on the Internet connection with Cork is expected when we don't control the background noise and competing traffic. The experiment described in section 5.3 will attempt to ascertain whether this behavior holds for every run of Libdalbe on the Internet.

5.2.1.2 Cork Vegas

DA-LBE Vegas again offers behavior that invites analysis more so than the DA-LBE Cubic experiment. The performance of DA-LBE Vegas in figure 5.7 shows a significant departure from the behavior seen in figure 5.4. Most notably, the DA-LBE Vegas flow in this experiment

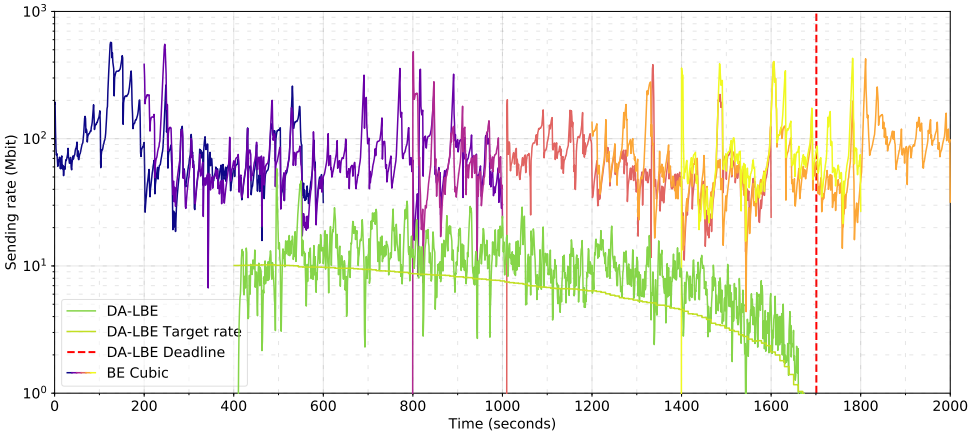


Figure 5.6: One-second averages of bits sent for the test defined in table 5.1, with the DA-LBE Cubic metacontroller, run on the Cork setup.

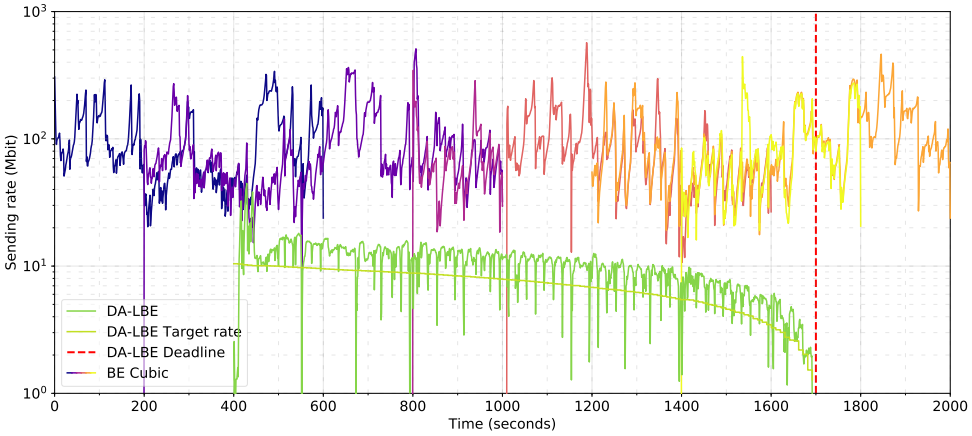


Figure 5.7: One-second averages of bits sent for the test defined in table 5.1, with the DA-LBE Vegas metacontroller, run on the Cork setup. We set α, $\beta = 16$.

seems to have no trouble reaching its deadline, though not by as big a margin as DA-LBE Cubic did. Contrary to the test bed test, where the Vegas β parameter inhibited Libdalbe from sending aggressively enough, DA-LBE Vegas on the Cork setup seems to behave more like an ideal DA-LBE flow: It is adhering to its target rate, reaches the deadline seemingly without major trouble, and does not compete on the level of its BE competitors after establishing a good BaseRTT.

Figure 5.8 gives more insight into why the DA-LBE Vegas metacontroller running on the Cork setup behaves differently from the test bed. The main factor in the Internet performance is probably the measured queuing delay; figures 5.8b and 5.8c show that the BaseRTT is quickly established and that the RTT, both the average and the momentary rolling average measurements, never rises far beyond 2.5 ms over the BaseRTT. These low averages of queuing delay are expected on the Cork setup as having standing queues in a network with a capacity of 500 Mbit/s would require constant traffic of the same rate, a traffic behavior that is different to the typically bursty nature of Internet traffic[2]. Figure 5.8e shows that there are much fewer congestion events from delay than in the test bed test, though the increase in sending rate and the noise on the Internet lead to a corresponding increase in loss events. There seems to be a correlation between dips in Libdalbe throughput and that of the BE throughput; these bursts of congestion are likely the main contributor to the loss seen by Libdalbe.

Though 2.5 ms is the average RTT value used in the Vegas model of our DA-LBE metacontroller, seen in figure 5.8b, the momentary RTT value is probably much closer to the BaseRTT given the bursty nature of Internet traffic; if there was any significant queuing delay present, it would be heavily inflated as evidenced by the μ value of almost zero. We suspect that the massive bandwidth of the link allows for near-BaseRTT delay for most traffic, with sudden large bursts of loss or higher RTT as can be seen in the dips in the throughput of both Libdalbe and its BE competitors. These conditions allow for DA-LBE Vegas to operate in a BE-like manner which means our metacontroller adjustments to the perceived queuing delay are the main driver of `cwnd` reduction.

The fairly consistent major dips in throughput spaced about 20 s apart are curious; they don't seem to line up with our metacontroller intervals, which points to a recurring external event being the source of these features. They seem to be correlated, at least in part, to the

[2]Internet traffic is typically bursty with periods of silence, as explained in section 2.3.2.

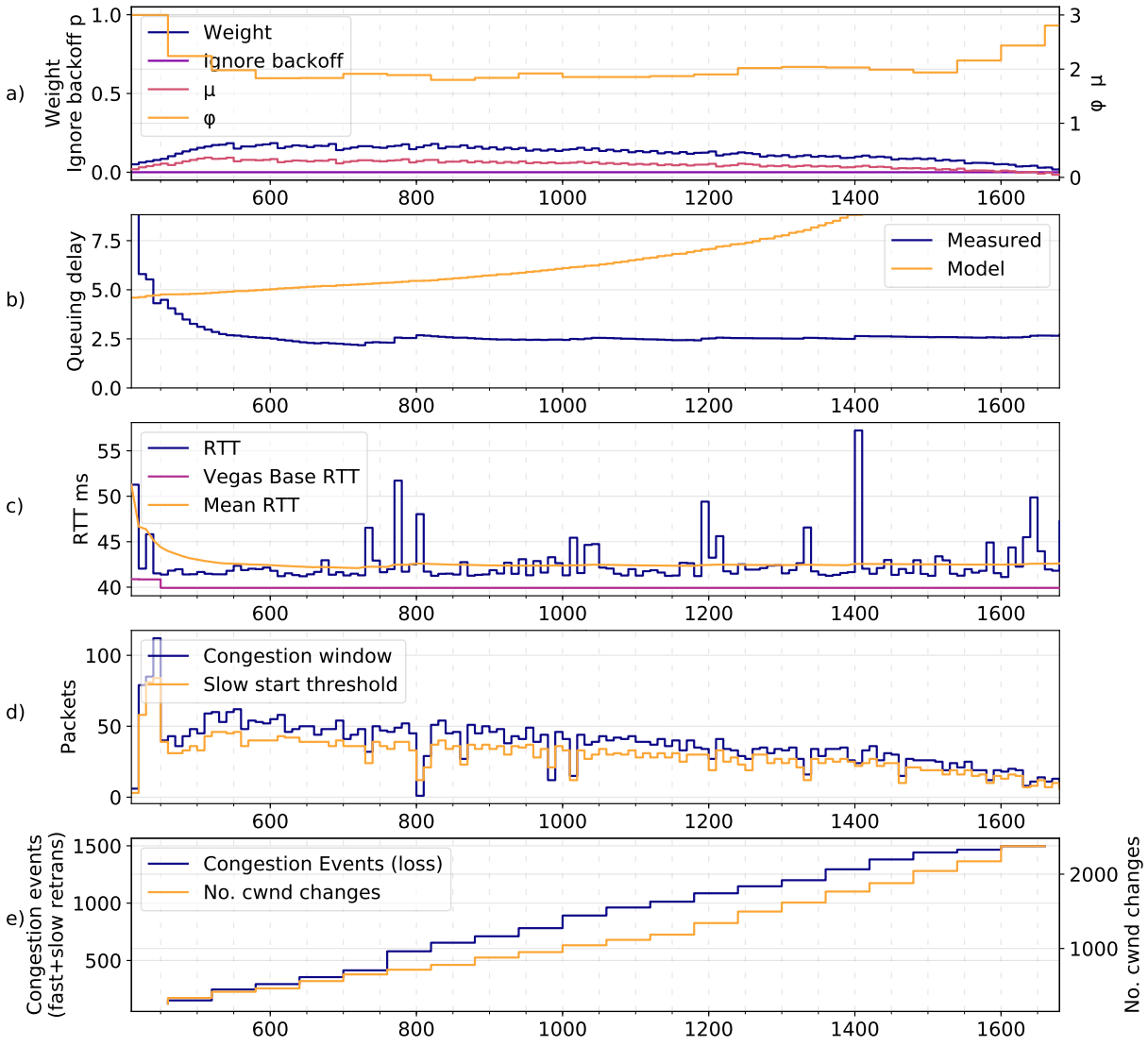


Figure 5.8: Debug graphs for Libdalbe DA-LBE Vegas on the Cork setup, containing metacontroller statistics for each interval the flow was active.

same congestion events that cause the competing BE flows to shrink their `cwnd`s. Our best guess is that a limiting buffer along the route is overflowing due to some regular burst in traffic, causing spikes of packet loss. The source of the regular traffic that might cause these bursts is hard to guess, but it is probable that it is some regular process causing traffic close to either the sender or close to the receiver. This kind of heavy regular congestion occurring in core networking equipment is unlikely.

5.2.2 Performance on the Internet Summary

Both of our metacontrollers seem to be operating as we would expect a DA-LBE service would for the Cork setup. DA-LBE Cubic is still sending faster than necessary, sometimes excessively so, but is by and large transmitting in an LBE-like manner while reaching its deadline in good time. The behavior of DA-LBE Vegas is interesting in that, different from the test bed experiment, it is like DA-LBE Cubic sending in excess of its target rate. The comparatively aggressive DA-LBE Vegas performance is probably explained by there being much less queuing delay in the network than on our congested single network link in the test bed.

5.3 Long-term Behavior

In this section we run an experiment from which we hope to derive meaningful data on the typical behavior of Libdalbe over multiple test runs. The experiment consists of fifty runs of a simple ten-minute test in which a DA-LBE flow competes against a single Cubic flow. The deadline for these tests is set much more aggressively than in previous tests so that Libdalbe will have fight harder to maintain its target rate; in setting the required aggression higher we hope to see Libdalbe at points having to compete on the same level as the BE-services and thereby showing a wider range of behavior from our metacontrollers.

We use a fairness metric to gauge whether Libdalbe upholds LBE-like behavior . The fairness results can be used to guess at how well Libdalbe can compete in order to reach its deadline — especially on the test bed where we can calculate the exact expected fairness outcome — but to provide deeper analysis on the ability of Libdalbe to keep a deadline we will explicitly look at the completion times of our experiments.

We run the longer-term experiment for both DA-LBE Cubic and DA-LBE Vegas, on both the Cork and the test bed setup. The test bed tests in this experiment will, due to the controlled nature of the setup, probably perform similarly throughout all of the fifty tests. Tests run on the Internet, on the other hand, are subject to a different type of noise, uncontrolled cross-traffic, which may significantly increase the variability of some tests.

Table 5.2 shows the test definition for the tests used in our long-term experiment. Our

Flow type	Start	End	Duration
DA-LBE	10 s		
BE Cubic	0 s	720 s	720 s

Table 5.2: Test for the Cork setup, defined as a set of flows with a start time and end time. The BE flow will halt all sending at its defined end time. The DA-LBE flow will keep sending until its payload has been transmitted in full (though not exceeding the 720 s limit given to the BE flow).

DA-LBE sender is still sending 1625 MB but now with a deadline of 600 s, which gives us a required average sending rate of 21.67 Mbit/s. The exact chosen amount of data is an arbitrary value reused from the scripted definitions of table 5.1 in our test orchestrator. The experiment consists of 50 test runs which should be enough to give us a solid indication of the behavior of Libdalbe. The BE Cubic flow is given ten seconds to warm up so that we are measuring Libdalbe impact on a steady-state BE competitor, meaning that the Libdalbe deadline is $t = 610$ s and that Libdalbe is given 710 s to finish each test.

5.3.1 Fairness

In this section we look at the fairness values from fifty runs of the test defined in table 5.2. The fairness is given by Jain's fairness index, explained in section 5.3.1, where we only consider data from within the timespan that the Libdalbe flow is active.

5.3.1.1 Expectations

We saw in figure 5.2 how DA-LBE Cubic sent slightly faster than it seemed to be targeting. It is expected that DA-LBE Cubic scores slightly higher in fairness than the other configurations for this experiment. Similarly, we have during development seen DA-LBE Vegas sending slightly above its target for the Cork setup; whether it compares to DA-LBE Cubic for this setup we cannot tell in advance.

It should be noted that, although presented in the same figure, the fairness results from the Cork and the local test bed experiments are not directly comparable. In the controlled test bed environment Libdalbe must compete against a greedy flow on a single bottleneck. The background noise is reasonably uniform and we know the exact available bandwidth. This environment is not one in which we envision Libdalbe being deployed.

The controlled environment does however allow us to reason about the expected fairness: Having a Libdalbe flow of 1625 MB completing in 600 s leaves capacity for transmitting 5125 MB of in the same time for the greedy flow, totalling 90 % of the entire bandwidth, with background noise occupying the last 10 %. The fairness calculation for the local test bed is thus $\frac{(1625+5125)^2}{2\times(1625^2+5125^2)} \approx 0.788$. Values much lower than 0.788 suggest that the Libdalbe flow did not finish in time. Values in excess of 0.788 could mean that either Libdalbe was too aggressive and finished early, or that the competing BE flow could not achieve its allocated bandwidth. Given the slightly eager sending rate of DA-LBE Cubic in figure 5.2 and the slight tardiness of DA-LBE Vegas in figure 5.4, it is reasonable to assume that similar results will show for these experiments, even with the increased required sending rate.

For the Cork experiments we have no information about the network apart from the equipment connecting the senders to the Internet and can therefore not reason in advance about the expected fairness. What we aim to show for the Cork experiments is that a Libdalbe flow competing with a BE flow results in strictly less than fair behavior. Exactly how much less than fair the results should be we cannot reasonably predict, but we expect the resulting fairness to be significantly lower than that achieved in the test bed experiments, due to there being considerably more bandwidth available for the competing greedy flow[3], and that the distribution of results will be much wider owing to unpredictable cross traffic and background noise.

[3]For example, if Libdalbe with its LBE service achieves 20 Mbit/s on average and its competitor achieves 100 Mbit/s on average, a very conservative estimate for the greedy flow, then the resulting fairness should be $\frac{(20+100)^2}{2\times(20^2+100^2)} \approx 0.692$.

5.3.1.2 Results and Analysis

Figure 5.9 and table 5.3 show the resulting fairness values from the 50 tests defined in table 5.2. At first glance the fairness measurements seem to correlate well with our expectations from section 5.3.1.1. Both DA-LBE-adapted TCPs appear to be reasonably less than fair, favoring their competition, on the Cork setup, with some outliers that can probably be shown to be a result of sudden changes in network conditions. The test bed results show DA-LBE Cubic measuring fairness values at close to the theorized maximum for the test bed, while DA-LBE Vegas appears to be transmitting much too slowly.

DA-LBE Cubic Cork Libdalbe on the Cork setup appears to be behaving less than fairly, with the median measuring at 0.683 and 0.674 for DA-LBE Cubic and DA-LBE Vegas respectively. DA-LBE Cubic shows only a few outliers, the most aggressive of which coincidentally almost matches the aforementioned minimum fairness of 0.788 required for the test bed setup. Figure 5.10 reveals that this most BE-like outlier test run is likely a result of the competing TCP hovering between 50 Mbit/s and 75 Mbit/s for roughly half of the test run while Libdalbe in those same periods managed to stay fairly close to its target rate. The BE competitor not managing to achieve a very high sending rate points to significant congestion from external traffic, which also means that our DA-LBE flow is having to compete in an almost BE-like manner to reach its target sending rate.

DA-LBE Vegas Cork The results for DA-LBE Vegas on the Cork setup show a spread of about the same size as DA-LBE Cubic, indicated by the whiskers of the box plot and the roughly similar standard deviation in measurements as seen in table 5.3, but a slightly less concentrated core of measurements as shown by the larger box. Though reasonably close to the DA-LBE Cubic results, DA-LBE Vegas shows slightly more variable behavior both indicated by a larger spread of the centremost half of the data and the two aggressive outliers. This behavior could show that DA-LBE Vegas is more reactive to the environment than DA-LBE Cubic is. Figure 5.11 shows the most BE-like outlier test run in which there is a lot of congestion in the network, as indicated by the BE flow not managing to achieve much more than 100 Mbit/s for most of the flow. Tests run in a continually congested network like that in figure 5.11 will naturally show Libdalbe achieving higher fairness scores, as it has to compete more to maintain its target rate while the greedy flow as result of congestion cannot claim the remainder of the bandwidth for itself. The same outlier test run also shows DA-LBE Vegas competing on par with its BE competitor until the real BaseRTT is discovered at around

	Fairness				
Experiment	Mean	Median	Stddev.	Max	Min
Cubic Cork	0.683	0.681	0.034	0.772	0.611
Vegas Cork	0.674	0.670	0.040	0.823	0.617
Cubic test bed	0.773	0.773	0.002	0.776	0.766
Vegas test bed	0.686	0.684	0.020	0.733	0.647

Table 5.3: Fairness results from the Libdalbe experiments on the Cork setup. The line in the middle separates results that are not directly comparable. Raw results can be found in table D.1.

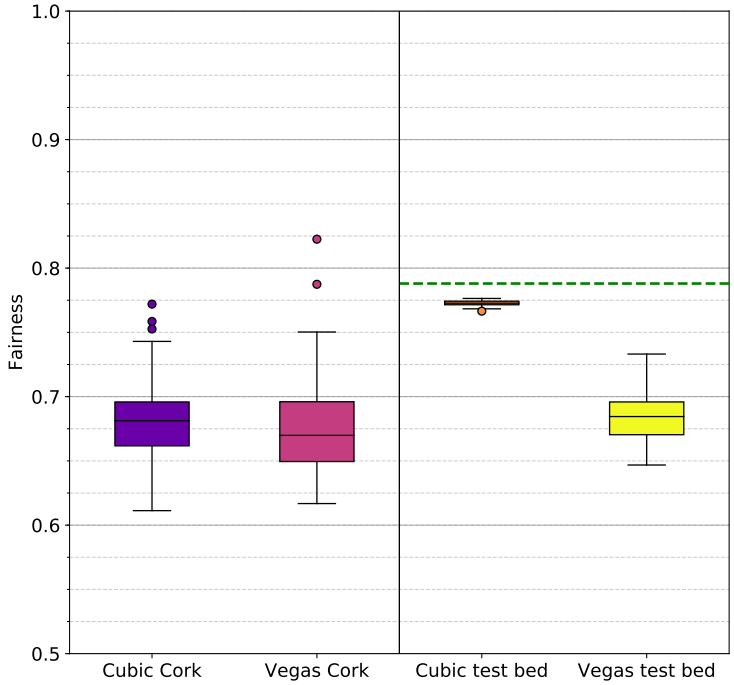


Figure 5.9: Box plot showing fairness measurements. The boxes cover the middle 50 % of the data, the middle line inside the boxes is the median, and the whiskers extend to the furthest measurement from the median still within 1.5 IQR on either side. The entire range of possible values for Jain's fairness index with two flows is shown. The dashed green line is the ideal resulting fairness if the Libdalbe flow finished exactly on the deadline and the competing TCP manages to utilize all of the remaining bandwidth.

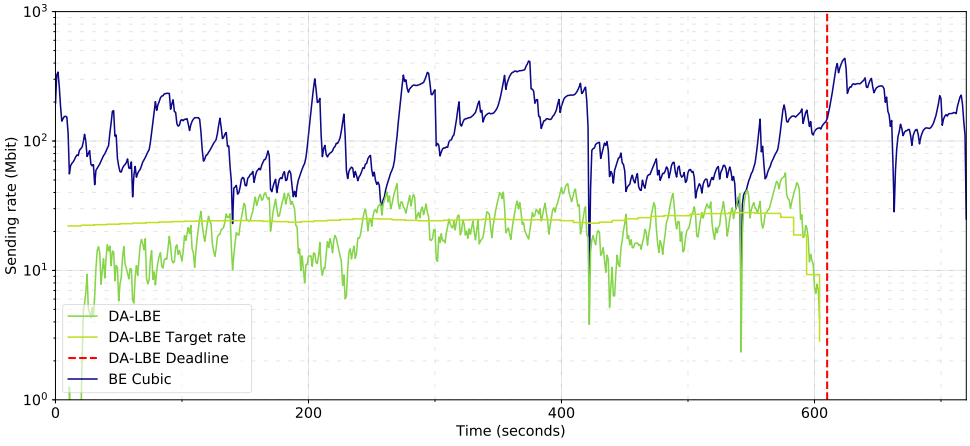


Figure 5.10: Throughput graph of the most BE-like DA-LBE Cubic outlier test run on the Cork setup. Note that the dashed red line, representing the deadline, is drawn at $t = 610$ s because the graph is plotted from the start of the whole test, including the 10 s starting delay of Libdalbe.

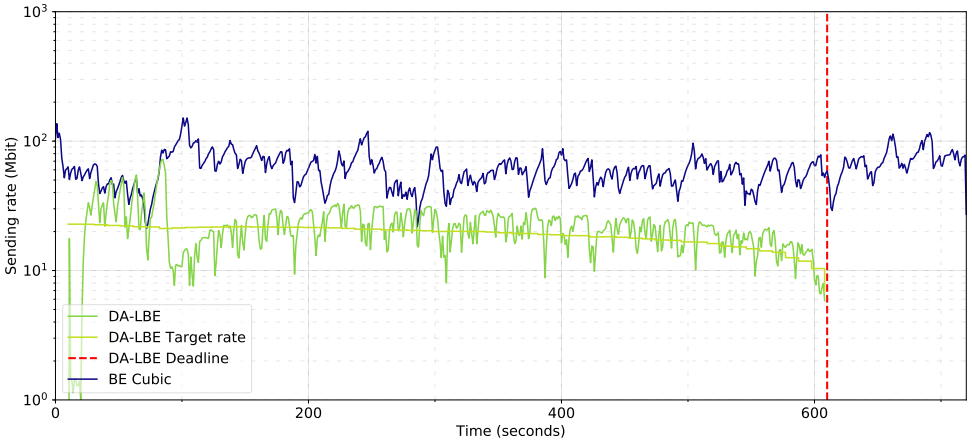


Figure 5.11: Throughput graph of the most BE-like DA-LBE Vegas outlier test run on the Cork setup.

$t = 40\,s$; with the metacontroller, using Vegas's measured BaseRTT, also wrongly estimating the queuing delay, there is a real possibility for these DA-LBE metacontrollers to out-compete BE TCP if configured incorrectly.

DA-LBE Cubic Test Bed DA-LBE Cubic on the test bed, shown in section 5.1.1 to perform proficiently in that specific scenario, behaves almost exactly as predicted for fairness. Taking into account that the competing BE flow will probably not be able to claim all of its allocated bandwidth, a median fairness result of 0.773, as seen in table 5.3, is indicative of exemplary DA-LBE behavior. The results from the DA-LBE Cubic test bed experiment show a remarkable regularity, with a standard deviation of just 0.002, an order of magnitude smaller than for the other experiments, and a total spread of only 0.01 where the other experiments show a minimum spread of 0.086. This can point to the accuracy with which the metacontroller described in section 2.6.3 models TCP Cubic, at least in a controlled environment; whether the higher spread of the DA-LBE Cubic Cork experiment signals inaccuracies in the model or if it is simply a result of Internet volatility is hard to ascertain.

DA-LBE Vegas Test Bed Where DA-LBE Cubic managed to achieve close to optimal fairness measurements, DA-LBE Vegas does not. Table 5.3 shows the median fairness score for DA-LBE Vegas on the test bed to be 0.684, almost as low as DA-LBE Cubic on the Cork setup, where there was much more bandwidth available. This low score is indicative of the BE TCP massively out-competing Libdalbe in this specific scenario. We will see in section 5.3.2.2 how the small spread of fairness measurements, the second lowest standard deviation in measurements, is due to DA-LBE Vegas rarely finishing within the test bounds.

5.3.2 Completion Times

To keep testing time within reasonable bounds, the experiment described in table 5.2 is configured with an absolute maximum limit of 720 s for each test run, giving the DA-LBE flow 710 s to reach its deadline. The test orchestrator sends a shutdown signal to the test application at $t = 710\,s$, after which the application is given another couple of seconds to shut down before it is forcibly killed at approximately $t = 720\,s$.

5.3.2.1 Expectations

For the Internet tests, where the momentary available bandwidth is unknown, we cannot reasonably predict the resulting completion times based on the fairness data in figure 5.9. From the fairness measurements we can however guess that DA-LBE Cubic and DA-LBE Vegas will perform similarly in relation to each other as they did for fairness, regardless of whether they finish their transfers by the deadline. As DA-LBE Vegas shows slightly fluctuating fairness results, at least more so than DA-LBE Cubic, we expect it to also exhibit similarly variable completion times. Comparison with the single test run from figure 5.7 can show us whether the behavior from that run was a fluke.

The test bed runs are again not indicative of real world performance, though they can show whether Libdalbe can uphold its deadline target while fighting lock-out when sharing one bottleneck with BE Cubic. DA-LBE Cubic is expected to finish close to the deadline even under pressure, as we saw in figure 5.9 that its fairness was close to what we predicted in a model-perfect scenario, though whether it can consistently reach its deadline for all 50 tests is uncertain. With its median fairness of 0.684 and its most aggressive measurement of 0.733 — far below all DA-LBE Cubic measurements and further below our calculated expected fairness for a DA-LBE flow completing on time — we can state with reasonable confidence that DA-LBE Vegas never reaches the deadline.

5.3.2.2 Results and Analysis

Figure 5.12 and table 5.4 show the resulting completion times from the 50 tests defined in table 5.2. A cursory overview of the completion time measurements indicate that our expectations from section 5.3.2.1 regarding the Cork setup experiments were accurate. On the Cork setup, DA-LBE Cubic reaches its deadline for every test run while showing a reasonably tight spread of measurements, and DA-LBE Vegas shows a compact middle two quartiles with a few outliers. For the test bed setup on the other hand, DA-LBE Cubic manages to reach the deadline with fair consistency while even the most aggressive outlier of DA-LBE Vegas struggles to reach the deadline.

DA-LBE Cubic Cork DA-LBE Cubic consistently finishes its transmission within the given deadline, a not so surprising result given the persistent aggressiveness that DA-LBE Cubic has shown for the Cork setup in section 5.2.1. That more than 75 % of the test runs finish in the penultimate interval, 580 s to 590 s, and not the final one, is curious: This many flows not finishing in the final interval could indicate that the overly eager sending rate of DA-LBE Cubic is not simply sending faster by a factor of the sending rate. Even with the large spread of fairness measurements most test runs finish more than a whole interval early which may suggest that the early completion time is not closely tied to sending rate or network congestion, as long as it is not constricted as is the case for the local test bed experiments.

DA-LBE Vegas Cork The completion times for DA-LBE Vegas on the Cork setup show a tight core of results which suggests stable performance from Vegas in DA-LBE mode when run on the Internet. The extreme outliers may be DA-LBE Vegas, for the aggressive test runs, wrongly estimating the BaseRTT or, for the late test runs, encountering a heavily congested network or a sudden increase in congestion when nearing the deadline.

Figures 5.13 and 5.14c show how the DA-LBE Vegas test that finished the earliest, at 37.2 s before the deadline, was impacted by 125 ms to 150 ms of delay until 150 s into the test. After

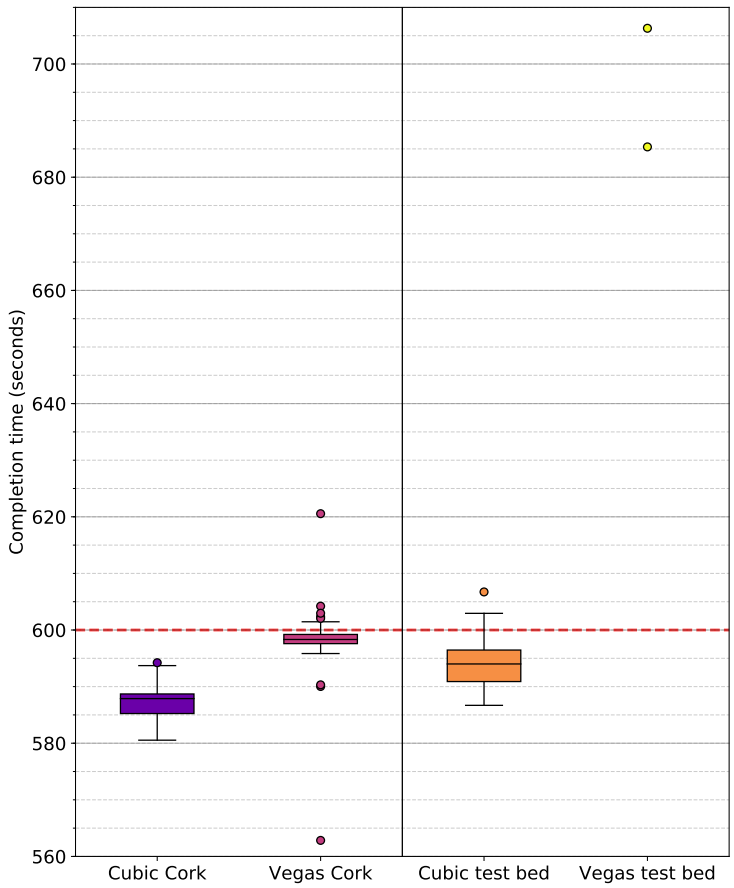


Figure 5.12: Box plot showing completion time for the same test as in figure 5.9, with the same options used for the box plot features. Completion time is listed as time relative to the start of the DA-LBE flow. The dashed red line is the deadline given to Libdalbe. The range spans from early enough to show all completion times until the absolute end of the test, 710 s after Libdalbe was started. Only a few outlier runs from the DA-LBE Vegas test bed experiment finished within the time limit, all of the other test runs lie outside the upper bounds of the plot.

	Completion times (delta from target)				
Experiment	Mean	Median	Stddev.	Max	Min
Cubic Cork	−12.9	−12.1	2.965	−5.8	−19.5
Vegas Cork	−1.9	−1.7	6.369	20.6	−37.2
Cubic test bed	−5.7	−6.0	4.148	6.7	−13.3
Vegas test bed	117.9	119.0	5.019	119.0	85.4

Table 5.4: Completion time results from the Libdalbe experiments on the Cork setup, shown as seconds relative to the deadline of 600 s. The tests should be forced to finish at 110 s after the deadline; results higher than this are the test applications failing to shut down in a timely manner. The line in the middle separates results that are not directly comparable. Raw results can be found in table D.2.

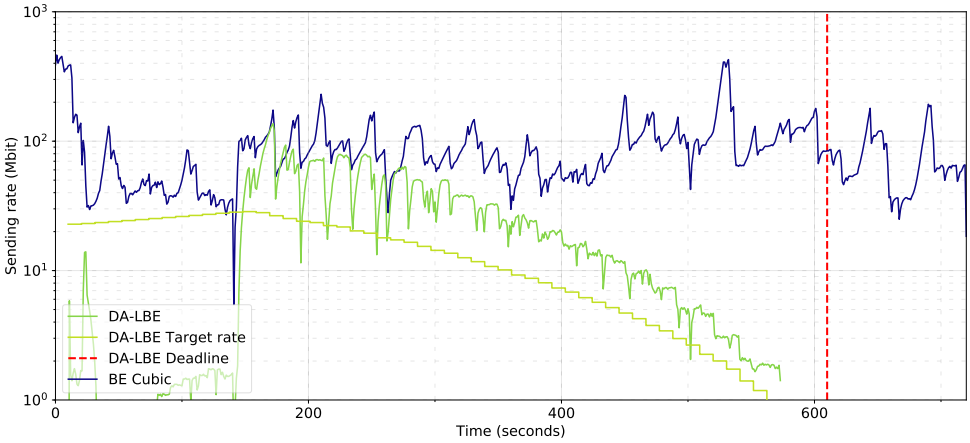


Figure 5.13: Throughput graph of the DA-LBE Vegas Cork test run that finished the earliest.

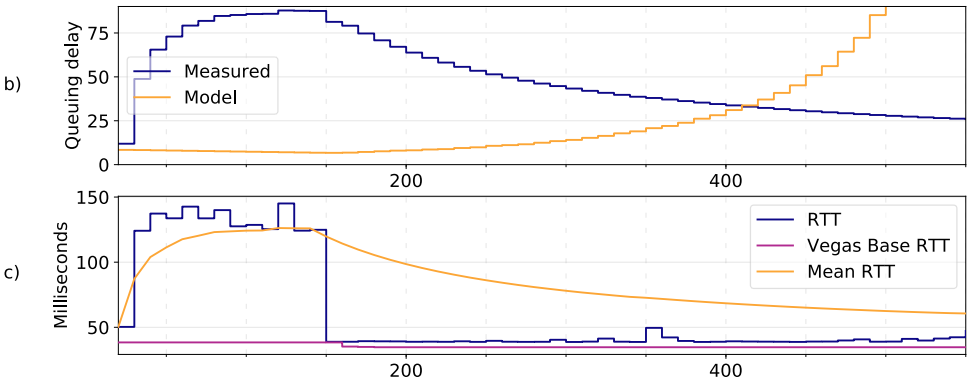


Figure 5.14: Libdalbe debug graphs showing queuing delay and RTT from the DA-LBE flow seen in figure 5.15. The range is limited to the time that our DA-LBE Vegas was active. This figure is a cutout from figure D.1.

this plateau of elevated delay the mean RTT, also seen in figure 5.14c, has been influenced enough so as to be a perpetual overestimation of the real RTT for the remainder of the flow. The use of a total mean RTT as opposed to a moving average to calculate queuing delay, as seen in figure 5.14b, is meant to stabilize the behavior of DA-LBE Vegas so as to make it less volatile. In this case however, the inflated mean RTT made the Vegas model perceive the queuing delay in the network to be significantly higher than the real values hinted at by the RTT measurements in figure 5.14c, resulting in DA-LBE Vegas being tuned to compete much more aggressively than warranted.

The last DA-LBE Vegas test run to finish also seems to be affected by rapidly changing network behavior. As seen in figure 5.15, after an initial period of an obvious BaseRTT misestimation, DA-LBE Vegas seems to be on track for reaching its deadline. At roughly $t = 540\,s$ there is an abrupt change in network conditions causing Libdalbe to scale back on its aggressiveness and ultimately missing the deadline by 20.6 s. Figure 5.16c shows a slight plateau in RTT at $t = 540\,s$ onwards which likely is the cause of Libdalbe ceasing its steady sending rate. The metacontroller is limited in how much it can scale its aggressiveness per interval, so there is likely no chance for Libdalbe to catch up to the ever-increasing target rate in the last 60 s before the deadline. The large spike in RTT at $t = 570\,s$ is likely a quick spike that was read by the metacontroller at just the wrong moment, as indicated by the corresponding much shorter spike at the same location in the throughput graph.

DA-LBE Cubic Test Bed DA-LBE Cubic manages to complete transmission by the deadline for most of its test runs. The late finishers seem to include *locked* TCP behavior, exemplified in figure 5.17, in which one flow continually increases its sending rate as its competitor continually similarly decreases their sending rate — in spite of the random noise added to combat exactly such behaviors. The most prominent case of TCP locking in figure 5.17 at roughly $t = 300\,s$ sees Libdalbe out-compete the BE flow for a short while, though the events that cause the missed deadline happen at $t = 575\,s$ and $t = 590\,s$ in which, for a short while, the BE flow shows a peak in sending rate as the LBE flow dips into a valley right before the deadline. Shown in appendix D.4, this behavior towards the very end of the test is exhibited in all of the test runs that finished after the deadline, though for some it happens a couple of intervals before the deadline and in some cases the effect is small enough not to really warrant attention if it weren't for the fact that the deadline was not met. As explained in section 2.6.5,

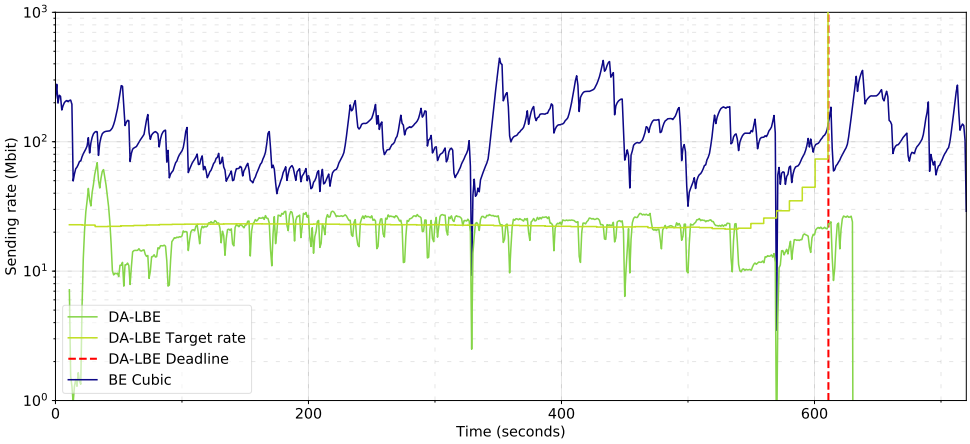


Figure 5.15: Throughput graph of the DA-LBE Vegas Cork test run that finished last.

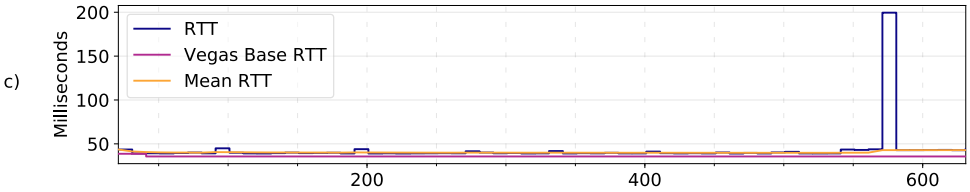


Figure 5.16: Libdalbe debug graph showing RTT from the DA-LBE flow seen in figure 5.15. The range is limited to the time that our DA-LBE Vegas was active. This figure is a cutout from figure D.2.

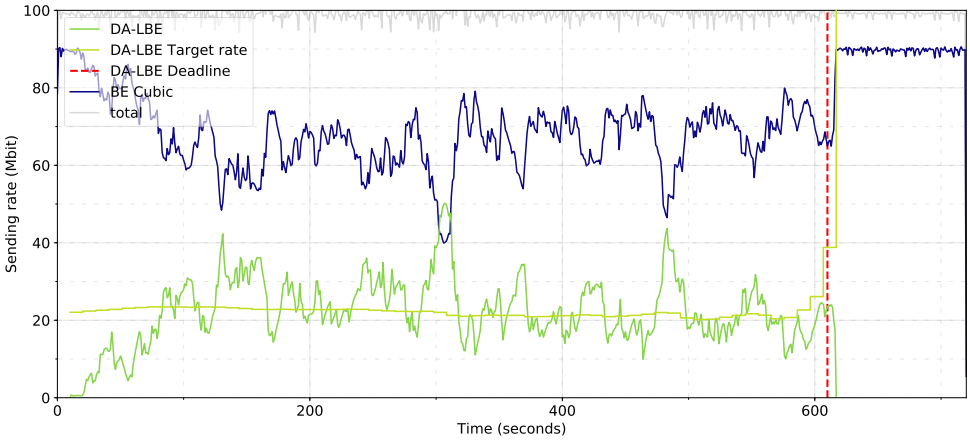


Figure 5.17: Throughput graph of the DA-LBE Cubic test bed run that finished last.

having our protocols miss the deadline by at most 6.7 s in the case of DA-LBE Cubic is still within acceptable DA-LBE performance and thus we can state that even the slower outliers here are successful DA-LBE runs.

DA-LBE Vegas Test Bed The DA-LBE Vegas experiments on the test bed do not seem to be performing well: Only in two outlier cases did DA-LBE Vegas manage to finish within this absolute deadline, with no test runs finishing within the Libdalbe deadline. Most of the recorded runs finish just outside the upper boundary of the plot in figure 5.12.

The test run that finished the earliest is shown in figure 5.18, where it can be seen that DA-LBE Vegas very rarely manages to reach, and can never maintain, the target rate. Figure 5.21, showing the Libdalbe debug output from the same test run as in figure 5.18, gives a few clues as to why the Libdalbe sending rate remains insufficient.

As in previous times we have examined Libdalbe debug output from DA-LBE Vegas in this chapter, the real BaseRTT takes time to establish. For this test run the real BaseRTT is never found, with the best estimate measuring 35 ms, 5 ms more than the real propagation delay. This should work in the favor of DA-LBE Vegas, as a higher BaseRTT estimate leads to smaller perceived queues. Where this BaseRTT misestimation probably hurts the sending rate however is every time a lower BaseRTT measurement is made the aggressiveness is tuned down and has to climb back up. The new BaseRTT estimates each show clear jumps in the measured queuing delay, seen in figures 5.19b and 5.19c, which will in turn impact the weight parameter and thus the sending rate.

Figure 5.20 shows throughput for the *second* earliest finishing test run, with accompanying Libdalbe debug output in figure 5.21. This is the slowest of the two outlier test runs that

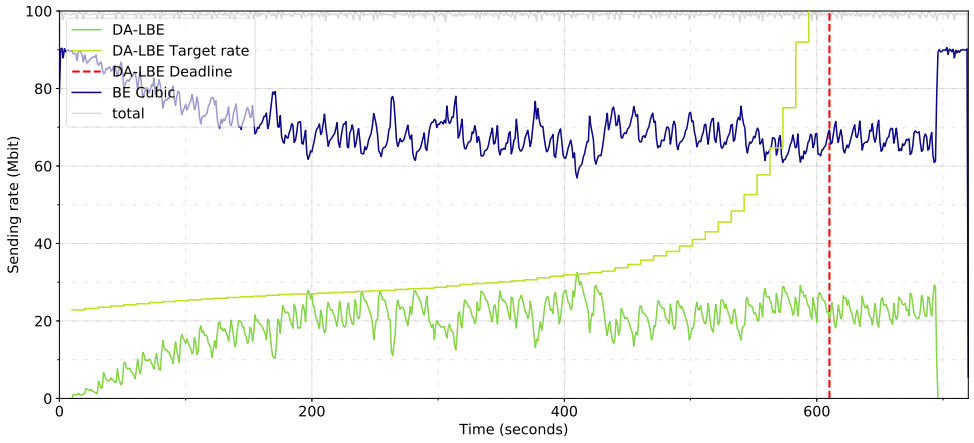


Figure 5.18: Throughput graph of the DA-LBE Vegas test bed run that finished the earliest.

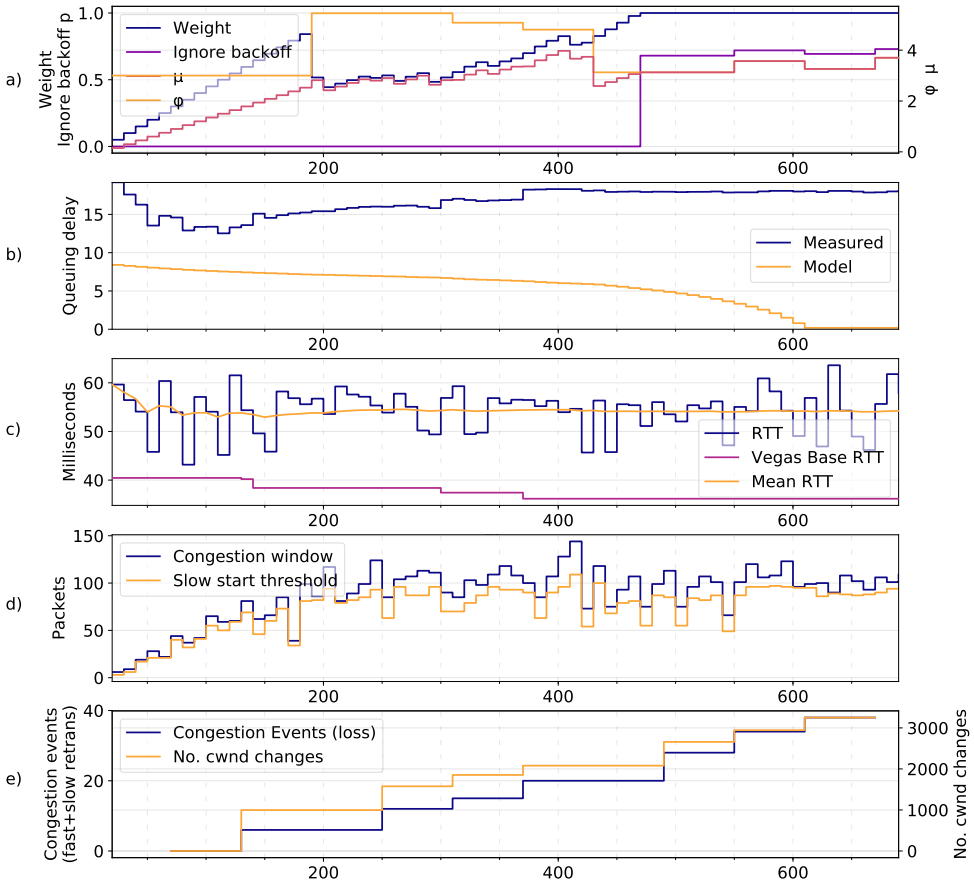


Figure 5.19: Libdalbe debug graphs from the DA-LBE Vegas test bed test run that finished the earliest. The range is limited to the time that our DA-LBE Vegas was active.

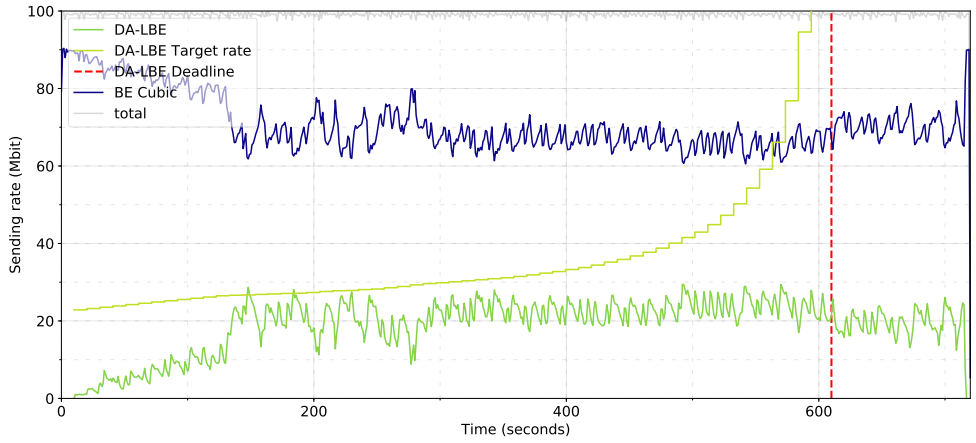


Figure 5.20: Throughput graph of the DA-LBE Vegas test bed run that finished the second earliest.

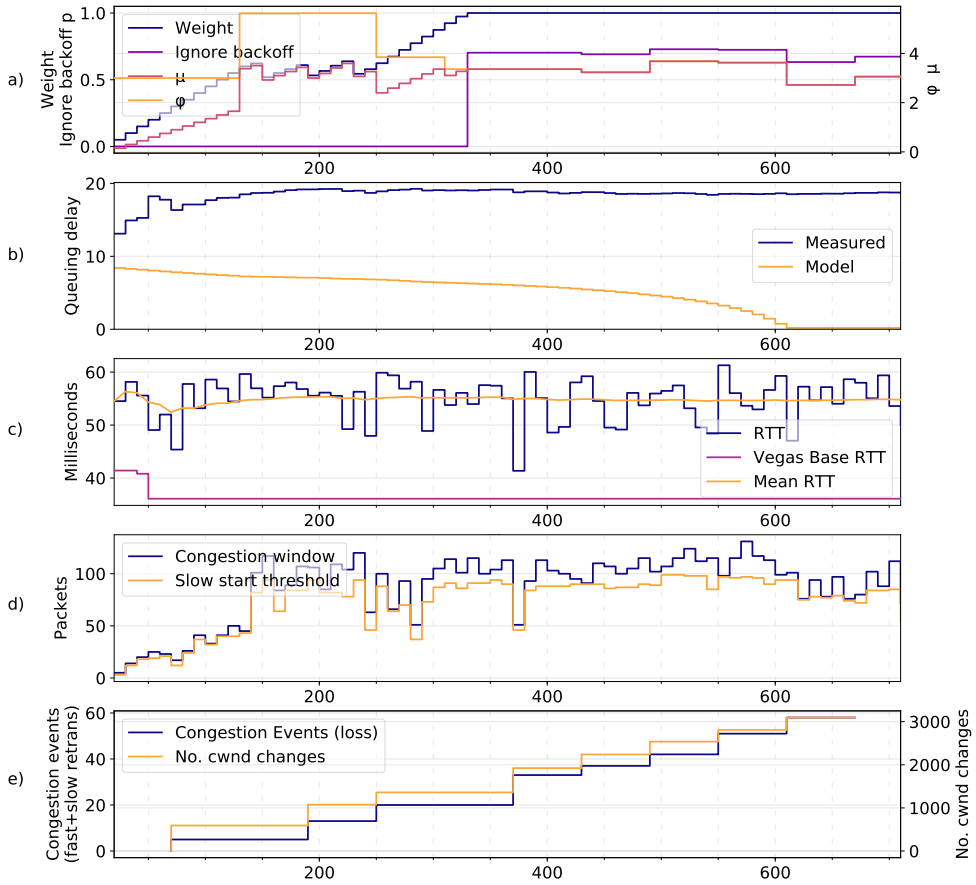


Figure 5.21: Libdalbe debug graphs from the DA-LBE Vegas test bed test run that finished the second earliest. The range is limited to the time that our DA-LBE Vegas was active.

finished within the absolute time limit. The second earliest test run to finish does exhibit an initial wrongful BaseRTT estimation, seen early on in figure 5.21c, though there are no more new estimations to keep bringing the aggression down. Figure 5.21a shows that maximum aggressiveness is reached and the backoff parameter kicks in at approximately $t = 330\,s$, yet DA-LBE Vegas still cannot compete with BE Cubic. Even in these two most aggressive cases DA-LBE Vegas struggles.

We had hoped that allowing DA-LBE Vegas to ignore a percentage of loss events when going for maximum aggression, indicated by the line labeled *Ignore backoff* (chance to ignore a loss event) in figure 5.21a, would make it able to compete with a BE TCP, but it seems to have no noticeable effect. Figure 5.20 shows no significant difference in sending rate after $t = 330\,s$ when the loss event ignore parameter goes into effect in this test run. The lack of impact can probably be explained by the almost total lack of congestion events for the DA-LBE Vegas flow, as seen in figure 5.21e where it is revealed that our Libdalbe flow saw no more than 60 loss events during the whole 710 seconds the flow was active. With `cwnd` changes caused by delay outnumbering loss signals by the thousands, it is no surprise that ignoring up to 75 % of loss signals causes no significant impact on sending rate, similarly to what we noted in section 5.1.2.3. As remarked for the table 5.1 experiments in section 5.1.2.3, with BE Cubic occupying 70 % of the queue on the single bottleneck our DA-LBE Vegas metacontroller will never be able to compete.

	Test run no.	
	31	49
Delta completion time	+85.35 s	+106.31 s
Fairness	0.733	0.725

Table 5.5: Measurements from the only two test bed DA-LBE Vegas test runs that finished within the absolute time limit.

A consequence of having most of the test runs finishing between $t = 710\,s$ and $t = 720\,s$ is that those same test runs will all register similar fairness measurements and thus skew the total results. Table 5.5 shows the completion time and fairness measurements for the two outliers. These values are at the very top of the upper whisker of measurements for DA-LBE Vegas in figure 5.9 and reveal that, as we touched on in section 5.3.1.2, its seemingly tight core of fairness measurements is due to the test runs being forced to shut down. This is further validated by table 5.4, which shows that the median and maximum completion times for DA-LBE Vegas on the test bed are both 119 s after the deadline, meaning over half of the measurements are exactly 119 s. Table D.2 reveals that only 6 out of 50 test runs completed at earlier than 199 s after the deadline. The fairness measurements still carry some sense of validity: Though the test runs were cut short, the fairness results still show that DA-LBE Vegas acted in a LBE manner in competing with BE Cubic. We can however not guarantee that DA-LBE Vegas would have continued to act in this manner if the test runs had not been cut short, and so the only truly representative measurements are those in table 5.5.

5.3.3 Long-term Behavior Summary

In section 5.3 we wanted to see how Libdalbe performed over multiple test runs. We ran 50 tests for each of our DA-LBE congestion metacontrollers on each of our setups and analyzed their fairness and completion times.

For fairness we saw both DA-LBE Cubic and DA-LBE Vegas on the Cork setup measuring mostly under a fairness of 0.7, which is leaning towards the least fair, or LBE-like, side of the scale. Some outliers tend to the fairer side, meaning more BE-like, which for the DA-LBE Cubic test runs seems to be resulting from the BE competitor not managing to attain a very high sending rate. The most fair DA-LBE Vegas test run exhibits a similar situation in which our DA-LBE flow keeps to its sending rate while the competing BE-flow struggles to claim the rest of the bandwidth due to external congestion. The measurements for the test bed experiments show larger differences between DA-LBE Cubic and DA-LBE Vegas, with DA-LBE Cubic scoring almost exactly the fairness value we predicted for a DA-LBE flow reaching its deadline with perfect bandwidth utilization. DA-LBE Vegas seems to score heavily on the less-than fair side of the scale; we predicted that the low fairness scores meant that DA-LBE Vegas never reached the deadline which we later showed to be correct.

Completion times for the Cork setup seem to better highlight how changing network conditions alter Libdalbe performance. Both DA-LBE Cubic and DA-LBE Vegas finish their transmission in a tight window, with DA-LBE Cubic consistently finishing more than ten seconds early. We highlighted some runs of DA-LBE Vegas on the Cork setup that showed quirks of DA-LBE Vegas, namely BaseRTT misestimation, the pitfalls of using total mean RTT for estimating queuing delay in the metacontroller, and how the metacontroller backs off significantly on increased congestion even when close to the deadline. DA-LBE Cubic on the test bed seems to be performing well, for the most part sticking to its target rate while fairly consistently reaching its deadline. We showed how DA-LBE Cubic, being pitted against another loss-based TCP, exhibits locking behavior, in which one of the two TCPs sees a temporary major increase in sending rate while its competitor sees a corresponding dip — in spite of the random noise added to the test bed setup to combat exactly this situation. DA-LBE Vegas struggles on the test bed setup; as we noted earlier, in section 5.1.2.3, the Vegas congestion controller itself is affected too much by the bottleneck congestion for our metacontroller to make it aggressive enough.

Chapter 6

Conclusion

This thesis presented Libdalbe, an interface to DA-LBE control mechanisms in the kernel, along with working implementations of DA-LBE transport services. In chapter 2 we explained how applications wanting to transmit data with a lower bandwidth allocation could use Less-than Best Effort services and that we could make these LBE services adhere to a soft deadline through adjusting congestion signals for the underlying congestion control, causing it to perceive more or less congestion on the network as needed. This adjustment of congestion signals is calculated by modeling congestion on the network based on the congestion signals received by the TCP being controlled.

We developed a library, Libdalbe, to adjust perceived congestion signals in order to achieve a DA-LBE service. Application programmers can use Libdalbe sockets in place of their standard Berkeley sockets to utilize DA-LBE-enabled transports. Libdalbe allows for custom congestion metacontrollers, of the sort described in chapter 2, which Libdalbe will utilize for socket control asynchronously from the programmer's main application. As part of Libdalbe we implemented two model-based DA-LBE metacontrollers, one for TCP Cubic and one for TCP Vegas, both of which manage to achieve LBE-like behavior while adhering to a soft deadline.

We created a local test bed on which we emulated a dumbbell network topology with added propagation delay and rate limiting to test and verify Libdalbe performance. The solutions chosen for emulation were thoroughly tested to ensure a setup of reasonable likeness to a real network. To test Libdalbe and the congestion metacontrollers in a real Internet environment, with its random noise and cross traffic, we set up an Internet connection to Cork, Ireland. We developed a robust framework for testing, allowing us to run predefined tests sequentially, each with custom host-specific procedures for initialization, testing, analysis, and teardown. This included extensive debugging capabilities for our metacontrollers, the output of which enables deeper analysis of Libdalbe behavior when set in context with the performance metrics.

In chapter 5 we ran experiments covering each of our metacontrollers running on each of our two test setups, with the goal of ascertaining that Libdalbe succeeds in providing DA-LBE service in both environments.

6.1 Conclusions from our Evaluations

We quantitatively evaluated Libdalbe using two keys metrics: Jain's fairness index, as described in section 2.3.1.4, and transmission completion times. The fairness measurements will show whether our metacontrollers can uphold LBE-like qualities in various degrees

of congestion. The *deadline-aware* part of DA-LBE will be evaluated through transmission completion times, for which we aim for Libdalbe to finish within the given deadline as far as network conditions allow.

On the local test bed we saw how our metacontrollers work when set to compete for bandwidth on a single bottleneck. We calculated a value for the expected fairness measurement given that the DA-LBE flow finishes exactly on time and that the competitor utilizes all of the remaining bandwidth. If the DA-LBE flow scores a fairness higher than this value we can say for certain that it was too aggressive, as it would have to have finished much too early and possibly out-competing a BE flow to do so. Lower scores than this expected fairness signifies either that the DA-LBE flow finished late or that, more likely, the BE competitor did not manage to utilize the remaining available bandwidth.

DA-LBE Cubic in this environment measured a fairness close to the optimal calculated value, consistently scoring on the LBE-side of fair while seemingly reliably reaching its deadline. This signifies that the DA-LBE Cubic model is accurate in its modeling of Cubic both for the sender and for the competitor, and that the DA-LBE control adjustments behave as expected. DA-LBE Vegas appears to behave much less than fair in this scenario, which in the very least proves that the metacontroller acts LBE-like.

Completion times for our experiments on the test bed show vast differences between the two metacontrollers. Only DA-LBE Cubic manages to complete its sending close to the deadline, with only six of its test runs finishing after the deadline. All of the late DA-LBE Cubic runs seem to have reacted to network congestion in a way that made them send too slowly just before reaching the deadline. We show that TCP locking was not quite eliminated on our test bed. We hypothesized that the low fairness scores of DA-LBE Vegas on the test bed could mean that none of its test runs finished in time, and we were proven right. Only two test runs, both of them outliers, finished within the bounds of the experiment and we showed that even though a misestimation of the BaseRTT should make Vegas more aggressive, DA-LBE Vegas can not compete with the 70 % standing queue of BE Cubic.

On the Internet setup we tested for Libdalbe performance on a real network, where we do not control neither the noise nor (all of) the competing traffic. The Internet setup has a much higher bandwidth than the local test bed setup, and a slightly longer RTT. With the higher sending rates causing correspondingly bigger reactions to loss events, due to the AIMD behavior of the competing TCPs, we expected Libdalbe to more easily finish sending by the deadline. The more interesting gauge of Libdalbe success on the Internet setup was whether it managed to consistently maintain LBE-like behavior.

DA-LBE Cubic tends to compete a little more aggressively than needed, but never so much as to approach BE-like sending rates for longer periods, and never so much as to finish transmission unreasonably early. Its fairness measurements put it as leaning towards LBE behavior more so than BE, though some outliers show high fairness measurements. We showed that, for the most aggressive outlier, network conditions did not allow the competing BE Cubic to claim much bandwidth, while DA-LBE Cubic held to its target rate and finished almost exactly on time.

We showed DA-LBE Vegas to be mostly LBE-like in behavior, except in situations where the BaseRTT is wrongly estimated or the mean RTT is affected enough to significantly skew the perceived queuing delay for the rest of the flow. The most aggressive outlier for DA-LBE Vegas was also shown to, like the most aggressive DA-LBE Cubic test run, result from higher than normal external congestion for that run.

We found both of our congestion metacontrollers to finish sending within reasonable time frames when employed on the Internet. With the deadline set at 600 s, DA-LBE Cubic finished before the deadline on every test run, while DA-LBE Vegas had seven test runs

finishing too late; none of DA-LBE Vegas's runs finished later than about 20 s after the deadline, which when further examined showed DA-LBE Vegas favoring LBE behavior when encountering congestion just before the deadline. Some quirks in DA-LBE Vegas, namely the BaseRTT misestimation in TCP Vegas itself and the use of a total mean RTT for the Vegas in the metacontroller, cause DA-LBE Vegas to occasionally send at much higher rates than warranted by the target rate and the momentary network congestion, in some cases leading to the transmission completing much too early. While high sending rates in the absence of congestion is desirable in a DA-LBE transport and expected of DA-LBE Vegas, the misestimation of BaseRTT in TCP Vegas or a too high mean RTT measurement in the DA-LBE Vegas metacontroller may cause a more aggressive sending rate even in the face of congestion.

In general, we saw LBE-like behavior to be upheld excepting some cases where DA-LBE Vegas got a wrong measurement for its BaseRTT. DA-LBE Cubic scores seemingly very fair (BE-like) on the test bed setup, though we showed that those values were close to the expected optimal fairness for that environment. Libdalbe reaches its deadline with fair reliability on the Internet, with the slower test runs exhibiting a clear tendency of LBE behavior when faced with network congestion, which we deem to be an acceptable reason for tardiness. DA-LBE Vegas when run on the test bed, an environment with just a single bottleneck with a 70 % standing queue, absolutely fails to meet its deadline; though DA-LBE Vegas does exhibit the desirable traits of an LBE service, it fails as a DA-LBE service in such high-congestion environments.

6.2 Future Work

In this section we present some points in our work that could be improved, and some that could be expanded upon in future research.

6.2.1 Improvements to the Application

In section 3.7 we discuss some targets for improving the current implementation of Libdalbe. The specific problems mentioned therein are the locking problems that arise in multithreaded applications, such as concurrent access to shared resources, and the support for sending signals to other threads or processes for allowing more customization of DA-LBE options. We also can not verify the correct performance of Libdalbe when running multiple connections in parallel, as we in this thesis focused on the ability of Libdalbe to simply achieve DA-LBE. These kinds of problems could probably be fixed by making use of existing robust libraries for handling asynchronous tasks and message passing. Further in the category of small improvements to the application is error handling, discussed in section 3.5.4, specifically handling out-of-memory errors; Libdalbe attempts to handle errors gracefully and report failures back to the user, but more consideration could go into properly handling severe error states such as running out of memory when creating a user-facing application or service like Libdalbe.

For our testing we created a minimal application that had sufficient functionality for testing one long-lived flow using just one connection. In preparation of deploying this kind of service in a production environment, testing should be done using the kinds of applications that will actually be employed in that environment. In the environment of the Internet, it would be helpful to test Libdalbe performance in e.g. a fully featured file synchronization and backup application like Rsync [12], whose source code is openly available and thus open for alteration for testing purposes.

Lastly, in chapter 1 we touched on NEAT and how Libdalbe was supposed to have fit into their framework as a DA-LBE service available for applications to use. Early inclusion into NEAT would have helped in testing as we would have been able to test with any NEAT-enabled application.

6.2.2 Additional Testing

Our experiments for checking DA-LBE behavior, on both the local setup and the Internet, could be improved with several additions and extra experiments that would help in designing a robust DA-LBE metacontroller:

- Choosing a higher required sending rate for the long-term tests was probably not ideal; setting the required target rate equal to the one in the shorter tests would have allowed for better results comparisons. With more time it would have been interesting to see how Libdalbe performs in many different aggression scenarios, especially when requiring very low sending rates, or rates that would require sending with BE-like greediness.
- More and longer periods of low congestion would have allowed for analysis of metacontrollers that can not immediately exploit small periods, such as our implementation of DA-LBE Cubic. How long a metacontroller takes to claim the available bandwidth and subsequently how long it takes to give it up would both be of great interest.
- Having a steady-state DA-LBE flow interrupted by bursts of BE traffic, would have helped check for the ability of the metacontroller to effectively back off from having claimed all available bandwidth, similarly to the previous point, with the difference being that the metacontroller might behave differently when being allowed full bandwidth before being challenged.
- Testing Libdalbe in different applications against real traffic traces, checking whether it significantly affects user-perceived quality of service for the BE flows, could further highlight use cases for DA-LBE services. Real traces could also show whether end users benefit at all from background services employing DA-LBE transport.
- Playing back real usage scenarios, traces of a user interacting with an application rather than the traffic itself, with applications for which Libdalbe support has been added, could help highlight use cases for DA-LBE in user-facing applications such as backups, streaming media players, or a utility like Rsync [12].

Our experiments were sufficient to prove that Libdalbe does indeed achieve LBE-ness while adhering to a soft deadline, but more diverse testing ought to be done if a metaprotocol like this is to be deployed on the Internet.

6.2.3 Areas of Interest for Future Research

In our work we encountered some topics that could warrant further exploration in future research.

- The DA-LBE Cubic performance in our experiments seemed to be slightly more aggressive than needed, behaving consistently enough to warrant us suggesting that the time by which DA-LBE Cubic beats the deadline may be caused by something else than simply sending faster than the required rate by a simple factor. Making DA-LBE

Cubic send more closely to the target rate might be achieved by a more accurate Cubic model, both for the sender and for its competitors. An inaccurate model of the network BE traffic could turn DA-LBE Cubic away from the true target rate, and so too can a wrongful model for the Cubic on the sender itself.

- The DA-LBE Vegas metacontroller needs tuning to be able to work in highly congested scenarios such as our local test bed. Our analysis indicates that this is a difficult issue which in itself warrants more exploration.
- The DA-LBE Vegas metacontroller in general could benefit from a deeper analysis. We showed one test run in which the metacontroller, employing a mean RTT for the whole run, was sufficiently affected by an initial plateau of high RTTs to subsequently overestimate the congestion on the network for the rest of the flow. A good target for future exploration is the stability of DA-LBE Vegas performance, and perhaps research into ways to ensure DA-LBE Vegas still sends in a LBE manner even if the BaseRTT is incorrectly estimated.
- Our metacontrollers could be affected by AQMs on the network. While we don't expect simple preemptive packet drop or ECNs to severely affect the performance of Libdalbe, AQMs which categorize traffic and treat each category differently, such as the ones described by Abbas *et al.* [1], might affect the perceived congestion in our metacontrollers.
- The DA-LBE kernel component includes provisions to ignore ECNs and even loss signals. While reacting to ECNs is not required of a TCP implementation, allowing a TCP to ignore loss could lead to sending faster than would be fair to other BE TCPs. Extreme care must therefore be taken in implementing metacontrollers using these parameters. Whether such parameters should be made easily accessible to application programmers without requiring administrative rights on the machine needs to be examined if Libdalbe is to be made available beyond just computer science researchers.

6.3 Achieving our Thesis Goals

In section 1.2 we outlined three goals for this thesis. This thesis presented the library, Libdalbe, we developed to provide accessible and robust DA-LBE service to application programmers. We analyzed Libdalbe and our metacontrollers and found them to be mostly successful in the goal of achieving LBE-like behavior while adhering to a soft deadline, or DA-LBE. When deployed on the Internet, both of our implemented solutions, DA-LBE Cubic and DA-LBE Vegas, transmit all of their data in a timely manner, though the experiments for the test bed show worse performance for DA-LBE Vegas when facing heavy congestion. Fairness measurements show that both of our metacontrollers maintain, for the vast majority of test runs, a consistently less than fair sending rate, which proves that they act LBE-like.

In implementing Libdalbe with two DA-LBE metacontrollers, and testing these in both emulated and real environments, we have fulfilled our goals for this thesis set out in section 1.2. To the best of our knowledge we have implemented the first working solution for a DA-LBE-enabled transport service.

Bibliography

[1] G. Abbas, Z. Halim, and Z. H. Abbas, "Fairness-driven queue management: A survey and taxonomy," *IEEE Communications Surveys Tutorials*, vol. 18, no. 1, pp. 324–367, 2016, ISSN: 1553-877X. DOI: 10.1109/COMST.2015.2463121.

[2] Apple, *Tcp_ledbat.c*, 2010. [Online]. Available: https://opensource.apple.com/source/xnu/xnu-1699.32.7/bsd/netinet/tcp%7B%5C_%7Dledbat.c.auto.html (visited on 06/26/2018).

[3] I. van Beijnum, *Understanding bufferbloat and the network buffer arms race*, 2011. [Online]. Available: https://arstechnica.com/tech-policy/2011/01/understanding-bufferbloat-and-the-network-buffer-arms-race/ (visited on 04/16/2018).

[4] A. Botta, A. Dainotti, and A. Pescapé, "A tool for the generation of realistic network workload for emerging networking scenarios," *Computer Networks*, vol. 56, pp. 3531–3547, 2012. DOI: 10.1016/j.comnet.2012.02.019. [Online]. Available: http://wpage.unina.it/a.botta/pub/COMNET%7B%5C_%7DWORKLOAD.pdf.

[5] Z. Bozakov, A. Brunstrom, D. Damjanovic, T. Eckert, K. R. Evensen, K.-j. Grinnemo, A. F. Hansen, N. Khademi, S. Mangiante, P. Mcmanus, G. Papastergiou, D. Ros, M. Tüxen, E. Vyncke, and M. Welzl, "NEAT Deliverable D1.1 NEAT Architecture," Tech. Rep., 2016, pp. 1–72. [Online]. Available: https://www.neat-project.org/wp-content/uploads/2018/04/D1.1-v1.1.pdf.

[6] L. S. Brakmo, L. L. Peterson, and S. W. O. Malley, "TCP Vegas : New Techniques Detection for Congestion and Avoidance," *Techniques*, vol. 24, Issue, no. TR 94 04, pp. 24–35, 1994, ISSN: 0146-4833. DOI: 10.1145/190314.190317. [Online]. Available: http://reference.kfupm.edu.sa/content/t/c/tcp%7B%5C_%7Dvegas%7B%5C_%7D%7B%5C_%7Dnew%7B%5C_%7Dtechniques%7B%5C_%7Dfor%7B%5C_%7Dcongestion%7B%5C_%7D53554.pdf.

[7] A. Brunstrom, D. Damjanovic, K. Evensen, G. Fairhurst, A. F. Hansen, F. Haugseth, D. Hayes, T. Hirsch, T. Jones, N. Khademi, P. Mcmanus, A. Petlund, D. Ros, T. Rozensztrauch, R. Santos, D. Stenberg, M. Tüxen, E. Vyncke, H. Wallenburg, F. Weinrank, and M. Welzl, "NEAT Deliverable D4.3 Validation and evaluation results," Tech. Rep., 2018, pp. 1–77. [Online]. Available: https://www.neat-project.org/wp-content/uploads/2018/05/D4.3.pdf.

[8] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "Bbr: Congestion-based congestion control," *ACM Queue*, vol. 14, September-October, pp. 20–53, 2016. [Online]. Available: http://queue.acm.org/detail.cfm?id%20=%203022184.

[9] G. Carofiglio, L. Muscariello, D. Rossi, and S. Valenti, "The quest for ledbat fairness," in *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*, 2010, pp. 1–6. DOI: 10.1109/GLOCOM.2010.5683559.

[10] L. Collin, *XZ Utils*, 2018. [Online]. Available: https://tukaani.org/xz/ (visited on 05/09/2018).

[11] *Creating Extension Libraries for Ruby*, 2017. [Online]. Available: https://github.com/ruby/ruby/blob/trunk/doc/extension.rdoc (visited on 05/01/2018).

[12] W. Davison, *Rsync*, 2018. [Online]. Available: https://rsync.samba.org/ (visited on 06/25/2018).

[13] M. Devera, *HTB Home*, 2003. [Online]. Available: http://luxik.cdi.cz/%7B~%7Ddevik/qos/htb/ (visited on 05/08/2018).

[14] Esnet and Lawrence Berkeley National Laboratory, *iPerf - The TCP, UDP and SCTP network bandwidth measurement tool*, 2016. [Online]. Available: https://iperf.fr/ (visited on 04/25/2018).

[15] *Extending Python with C or C++ — Python 3.6.5 documentation*, 2018. [Online]. Available: https://docs.python.org/3/extending/extending.html (visited on 05/01/2018).

[16] W.-c. Feng, D. D. Kandlur, D. Saha, and K. G. Shin, "BLUE: A new class of active queue management algorithms," *Ann Arbor*, pp. 1–27, 1999. [Online]. Available: http://www.eecs.umich.edu/techreports/cse/99/CSE-TR-387-99.pdf%20http://www.cs.ust.hk/faculty/bli/660h/feng99blue.pdf.

[17] *Fiber med høy hastighet og mye kapasitet - Telenor*. [Online]. Available: https://www.telenor.no/privat/bredband/fiber/ (visited on 05/09/2018).

[18] T. Filiba, *Plumbum: Shell Combinators and More*. [Online]. Available: https://plumbum.readthedocs.io/en/latest/#about (visited on 04/25/2018).

[19] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, 1993, ISSN: 1063-6692. DOI: 10.1109/90.251892.

[20] J. Forcier, *Fabric*. [Online]. Available: http://www.fabfile.org/ (visited on 04/25/2018).

[21] J. Gettys and K. Nichols, "Bufferbloat: Dark Buffers in the Internet," *Communications of the ACM*, vol. 55, no. 1, p. 57, 2012, ISSN: 00010782. DOI: 10.1145/2063176.2063196. [Online]. Available: http://dl.acm.org/citation.cfm?doid%20=%202063176.2063196.

[22] S. Ha and I. Rhee, "CUBIC : A New TCP-Friendly High-Speed TCP Variant," *ACM SIGOPS Operating Systems Review - Research and developments in the Linux kernel*, vol. 42, no. 5, pp. 64–74, 2008, ISSN: 0163-5980. DOI: 10.1145/1400097.1400105. [Online]. Available: http://www4.ncsu.edu/%7B~%7Drhee/export/bitcp/cubic-paper.pdf.

[23] 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," in *Proceedings 2000 International Conference on Network Protocols*, 2000, pp. 177–186. DOI: 10.1109/ICNP.2000.896302.

[24] D. A. Hayes, D. Ros, A. Petlund, and I. Ahmed, "A Framework for Less than Best Effort Congestion Control with Soft Deadlines," 2017, ISBN: 9783901882944.

[25] "Ieee standard for ethernet," *IEEE Std 802.3-2015 (Revision of IEEE Std 802.3-2012)*, pp. 1–4017, 2016. DOI: 10.1109/IEEESTD.2016.7428776.

[26] "Ieee standard for information technology–telecommunications and information exchange between systems local and metropolitan area networks–specific requirements - part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications," *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)*, pp. 1–3534, 2016. DOI: 10.1109/IEEESTD.2016.7786995.

[27] *Ieee std 1003.1-2017 (revision of ieee std 1003.1-2008): Ieee standard for information technology–portable operating system interface (posix(r)) base specifications, issue 7*, eng, 2018.

[28] *Internett-målinga - SSB*, 2018. [Online]. Available: https://www.ssb.no/teknologi-og-innovasjon/statistikker/inet/kvartal (visited on 05/29/2018).

[29] V. Jacobson, "Congestion avoidance and control," *ACM SIGCOMM Computer Communication Review*, vol. 18, no. 4, pp. 314–329, 1988, ISSN: 01464833. DOI: 10.1145/52325.52356. arXiv: arXiv:1011.1669v3. [Online]. Available: http://portal.acm.org/citation.cfm?doid%20=%2052325.52356.

[30] R. Jain and K. Ramakrishnan, "Congestion avoidance in computer networks with a connectionless network layer: concepts, goals and methodology," *[1988] Proceedings. Computer Networking Symposium*, pp. 134–143, 1997, ISSN: 01464833. DOI: 10.1109/CNS.1988.4990. arXiv: 9809095 [cs]. [Online]. Available: http://ieeexplore.ieee.org/document/4990/.

[31] R. K. Jain, D.-M. W. Chiu, and W. R. Hawe, "A quantitative measure of fairness and discrimination for resource allocation in shared computer system," *DEC technical report TR301*, vol. cs.NI/9809, no. DEC-TR-301, pp. 1–38, 1984. [Online]. Available: http://www.cs.wustl.edu/%7B~%7Djain/papers/ftp/fairness.pdf.

[32] *Java Native Interface Specification: 1 - Introduction*, 2017. [Online]. Available: https://docs.oracle.com/javase/9/docs/specs/jni/intro.html (visited on 05/01/2018).

[33] F. P. Kelly, A. K. Maulloo, and D. K. H. Tan, "Rate Control for Communication Networks : Shadow Prices , Proportional Fairness and Stability Published by : Palgrave Macmillan Journals on behalf of the Operational Research Society Stable URL : http://www.jstor.org/stable/3010473 Rate control for commun," vol. 49, no. 3, pp. 237–252, 1998.

[34] F. Kelly, "Charging and rate control for elastic traffic (corrected version)," *European Transaction on Telecommunication*, vol. 8, no. 1, pp. 33–37, 1997.

[35] M. Kerrisk, *Linux Man Pages*, 2011. [Online]. Available: http://www.kernel.org/doc/man-pages/ (visited on 05/08/2018).

[36] ——, *The linux programming interface : A linux and unix system programming handbook*, eng, San Francisco, 2010.

[37] T. Kluyver, B. Ragan-kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, and C. Willing, "Jupyter Notebooks—a publishing format for reproducible computational workflows," *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pp. 87–90, 2016, ISSN: 0015-0193. DOI: 10.3233/978-1-61499-649-1-87.

[38] T. V. Lakshman, A. Neidhardt, and T. J. Ott, "The drop from front strategy in TCP and in TCP over ATM," in *Proceedings of IEEE INFOCOM 96 Conference on Computer Communications*, vol. 3, 1996, pp. 1242–1250, ISBN: 0818672935. DOI: 10.1109/INFCOM.1996.493070. [Online]. Available: http://www.cs.tut.fi/%7B~%7Dkucherya/book/lakshman96.pdf.

[39] *Libuv: Cross-platform asynchronous I/O*, 2015. [Online]. Available: http://libuv.org (visited on 04/25/2018).

[40] *Linux kernel source tree*. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree.

[41] D. S. Miller, *net-next.git - David Miller's -next networking tree*, 2018. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/davem/net-next.git/ (visited on 05/07/2018).

[42] P. Miller, *Libexplain*, 2008. [Online]. Available: http://libexplain.sourceforge.net/ (visited on 04/25/2018).

[43] J. C. Mogul and K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," *ACM Transactions on Computer Systems*, vol. 15, no. 3, 1997, ISSN: 0734-2071.

[44] S. Murdoch, "Hot or not: Revealing hidden services by their clock skew," *Proceedings of the 13th ACM conference on Computer ...*, pp. 27–36, 2006, ISSN: 1595935185. DOI: 10.1.1.65.9298. [Online]. Available: http://dl.acm.org/citation.cfm?id%20=%201180410.

[45] Python Software Foundation, *Python*, 2018. [Online]. Available: https://www.python.org/ (visited on 05/09/2018).

[46] L. Qiu, Y. Zhang, and S. Keshav, "Understanding the performance of many TCP flows," *Computer Networks*, vol. 37, no. 3-4, pp. 277–306, 2001, ISSN: 1389-1286. DOI: 10.1016/S1389-1286(01)00203-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128601002031.

[47] R. Braden (Ed.), *Requirements for Internet Hosts - Communication Layers*, RFC 1122 (Internet Standard), RFC, Updated by RFCs 1349, 4379, 5884, 6093, 6298, 6633, 6864, 8029, Fremont, CA, USA: RFC Editor, 1989. DOI: 10.17487/RFC1122. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1122.txt.

[48] ——, *Requirements for Internet Hosts - Application and Support*, RFC 1123 (Internet Standard), RFC, Updated by RFCs 1349, 2181, 5321, 5966, 7766, Fremont, CA, USA: RFC Editor, 1989. DOI: 10.17487/RFC1123. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1123.txt.

[49] W. Stevens, *TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms*, RFC 2001 (Proposed Standard), RFC, Obsoleted by RFC 2581, Fremont, CA, USA: RFC Editor, 1997. DOI: 10.17487/RFC2001. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2001.txt.

[50] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, RFC 2119 (Best Current Practice), RFC, Updated by RFC 8174, Fremont, CA, USA: RFC Editor, 1997. DOI: 10.17487/RFC2119. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2119.txt.

[51] B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, and L. Zhang, *Recommendations on Queue Management and Congestion Avoidance in the Internet*, RFC 2309 (Informational), RFC, Obsoleted by RFC 7567, updated by RFC 7141, Fremont, CA, USA: RFC Editor, 1998. DOI: 10.17487/RFC2309. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2309.txt.

[52] M. Allman, V. Paxson, and W. Stevens, *TCP Congestion Control*, RFC 2581 (Proposed Standard), RFC, Obsoleted by RFC 5681, updated by RFC 3390, Fremont, CA, USA: RFC Editor, 1999. DOI: 10.17487/RFC2581. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2581.txt.

[53] K. Ramakrishnan, S. Floyd, and D. Black, *The Addition of Explicit Congestion Notification (ECN) to IP*, RFC 3168 (Proposed Standard), RFC, Updated by RFCs 4301, 6040, 8311, Fremont, CA, USA: RFC Editor, 2001. DOI: 10.17487/RFC3168. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3168.txt.

[54] M. Allman, V. Paxson, and E. Blanton, *TCP Congestion Control*, RFC 5681 (Draft Standard), RFC, Fremont, CA, USA: RFC Editor, 2009. DOI: 10.17487/RFC5681. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5681.txt.

[55] D. Mills, J. Martin (Ed.), J. Burbank, and W. Kasch, *Network Time Protocol Version 4: Protocol and Algorithms Specification*, RFC 5905 (Proposed Standard), RFC, Updated by RFC 7822, Fremont, CA, USA: RFC Editor, 2010. DOI: 10.17487/RFC5905. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5905.txt.

[56] B. Haberman (Ed.) and D. Mills, *Network Time Protocol Version 4: Autokey Specification*, RFC 5906 (Informational), RFC, Fremont, CA, USA: RFC Editor, 2010. DOI: 10.17487/RFC5906. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5906.txt.

[57] H. Gerstung, C. Elliott, and B. Haberman (Ed.), *Definitions of Managed Objects for Network Time Protocol Version 4 (NTPv4)*, RFC 5907 (Proposed Standard), RFC, Fremont, CA, USA: RFC Editor, 2010. DOI: 10.17487/RFC5907. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5907.txt.

[58] R. Gayraud and B. Lourdelet, *Network Time Protocol (NTP) Server Option for DHCPv6*, RFC 5908 (Proposed Standard), RFC, Fremont, CA, USA: RFC Editor, 2010. DOI: 10.17487/RFC5908. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5908.txt.

[59] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind, *Low Extra Delay Background Transport (LEDBAT)*, RFC 6817 (Experimental), RFC, Fremont, CA, USA: RFC Editor, 2012. DOI: 10.17487/RFC6817. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6817.txt.

[60] J. Chu, N. Dukkipati, Y. Cheng, and M. Mathis, *Increasing TCP's Initial Window*, RFC 6928 (Experimental), RFC, Fremont, CA, USA: RFC Editor, 2013. DOI: 10.17487/RFC6928. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6928.txt.

[61] J. Postel, *Internet Protocol*, RFC 791 (Internet Standard), RFC, Updated by RFCs 1349, 2474, 6864, Fremont, CA, USA: RFC Editor, 1981. DOI: 10.17487/RFC0791. [Online]. Available: https://www.rfc-editor.org/rfc/rfc791.txt.

[62] ——, *Transmission Control Protocol*, RFC 793 (Internet Standard), RFC, Updated by RFCs 1122, 3168, 6093, 6528, Fremont, CA, USA: RFC Editor, 1981. DOI: 10.17487/RFC0793. [Online]. Available: https://www.rfc-editor.org/rfc/rfc793.txt.

[63] T. Hoeiland-Joergensen, P. McKenney, D. Taht, J. Gettys, and E. Dumazet, *The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm*, RFC 8290 (Experimental), RFC, Fremont, CA, USA: RFC Editor, 2018. DOI: 10.17487/RFC8290. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8290.txt.

[64] J. Nagle, *On Packet Switches With Infinite Storage*, RFC 970, RFC, Fremont, CA, USA: RFC Editor, 1985. DOI: 10.17487/RFC0970. [Online]. Available: https://www.rfc-editor.org/rfc/rfc970.txt.

[65] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, and R. Scheffenegger, *CUBIC for Fast Long-Distance Networks*, RFC 8312, RFC, 2018. DOI: 10.17487/RFC8312. [Online]. Available: https://rfc-editor.org/rfc/rfc8312.txt.

[66] D. Rossi, C. Testa, S. Valenti, and L. Muscariello, "LEDBAT: The new BitTorrent congestion control protocol," *Proceedings - International Conference on Computer Communications and Networks, ICCCN*, 2010, ISSN: 10952055. DOI: 10.1109/ICCCN.2010.5560080.

[67] G. Stein, *This Is How Your BitTorrent Downloads Move So Fast*, 2013. [Online]. Available: https://www.fastcompany.com/3014951/why-your-bittorrent-downloads-move-so-fast (visited on 06/14/2018).

[68] D. E. L. Steven H. Low, "Optimization Flow Control-I-Basic Algorithm and Convergence," *Ieee/Acm Transactions on Networking*, vol. vol 7, NO. No. 6, pp. 861–874, 1999, ISSN: 1063-6692. DOI: http://dx.doi.org/10.1109/90.811451.

[69] I. Stoica, H. Zhang, and T. S. E. Ng, "A hierarchical fair service curve algorithm for link-sharing, real-time, and priority services," *IEEE/ACM Transactions on Networking*, vol. 8, no. 2, pp. 185–199, 2000, ISSN: 1063-6692. DOI: 10.1109/90.842141.

[70] L. E. Storbukås, "Implementing less than best effort with deadlines," Master's thesis, 2018, p. 82. [Online]. Available: http://urn.nb.no/URN:NBN:no-64318.

[71] J. Sun, S. Chan, and M. Zukerman, "IAPI: An intelligent adaptive PI active queue management scheme," *Computer Communications*, vol. 35, no. 18, pp. 2281–2293, 2012, ISSN: 01403664. DOI: 10.1016/j.comcom.2012.07.013. [Online]. Available: http://dx.doi.org/10.1016/j.comcom.2012.07.013.

[72] A. Tang, X. Wei, S. H. Low, and M. Chiang, "Equilibrium of heterogeneous congestion control: Optimality and stability," *IEEE/ACM Transactions on Networking*, vol. 18, no. 3, pp. 844–857, 2010, ISSN: 1063-6692. DOI: 10.1109/TNET.2009.2034963.

[73] Tcpdump, "Tcpdump/Libpcap," [Online]. Available: http://www.tcpdump.org/.

[74] The Linux Foundation, *Linux Foundation Wiki*, 2018. [Online]. Available: https://wiki.linuxfoundation.org/ (visited on 05/10/2018).

[75] ——, *Linux Kernel Documentation*. [Online]. Available: https://www.kernel.org/doc/Documentation/ (visited on 05/08/2018).

[76] N. Trichakis, A. Zymnis, and S. P. Boyd, *Dynamic network utility maximization with delivery contracts*, 1 PART 1. IFAC, 2008, vol. 17, pp. 2907–2912, ISBN: 9783902661005. DOI: 10.3182/20080706-5-KR-1001.0323. [Online]. Available: http://dx.doi.org/10.3182/20080706-5-KR-1001.00489.

[77] H. Wallenburg, *mpg_papers / thesis-2018-hugowallenburg – Bitbucket*, 2018. [Online]. Available: https://bitbucket.org/mpg%7B%5C_%7Dpapers/thesis-2018-hugowallenburg/src/master/ (visited on 07/02/2018).

[78] J. Wang, J. Yang, G. Xie, Z. Li, and M. Zhou, "On-line estimating skew in one-way delay measurement," in *Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2003, pp. 430–436. DOI: 10.1109/PDCAT.2003.1236339.

[79] *Writing R Extensions - System and foreign language interfaces*, 2018. [Online]. Available: https://cran.r-project.org/doc/manuals/r-release/R-exts.html#System-and-foreign-language-interfaces (visited on 05/01/2018).

[80] L. Xu, K. Harfoush, and I. Rhee, "Binary increase congestion control (BIC) for fast long-distance networks," *Proceedings - IEEE INFOCOM*, vol. 4, pp. 2514–2524, 2004, ISSN: 0743166X. DOI: 10.1109/INFCOM.2004.1354672.

Appendices

Appendix A

Interface details

A.1 Initializing the Library

DA-LBE Init

Signature

```
int dalbe_init(char *err)
```

Parameters

`char *err`
Destination for human-readable error messages generated by the library.

Return value

Zero if the function succeeded.

A negative value if initialization fails. This will only happen if the creation of the daemon thread fails, in which case the result of the thread creation system call will be returned to aid in debugging.

Operation in detail

Initializes the library: Memory is allocated on the heap for the (linked) list of sockets, locks required for the shared memory constructs are initialized, and the daemon thread is created. The only reasonable source of errors, short of a memory error during allocation, is if the creation of the daemon thread fails; a situation which should not occur during typical working conditions[1].

A.2 Disposing of the Library Resources

DA-LBE Dispose

Signature

[1] This would imply a programming error in the library or unsafe memory writes resulting in a deadlock or other undefined error states.

```
int dalbe_dispose(char *err)
```

Parameters

char *err
Destination for human-readable error messages generated by the library.

Return value
Zero if the function succeeded.

Operation in detail
All sockets that are still open are closed and disposed of, after which the daemon is shut down and all allocated memory is freed. The user need not call this function if they have already closed all open sockets and the application is about to shut down gracefully. Libdalbe does not create resources that would persist between runs should the dispose function not be called.

The dispose function will only fail under extraordinary circumstances where the tearing down of synchronization primitives would fail; a situation which should occur only if the memory allocated for these resources is tampered with. For this case there is no clean course of action.

A.3 Opening a DA-LBE Socket

DA-LBE Open

Signature

```
int dalbe_open(int domain,
    const char *proto,
    size_t size,
    time_t deadline,
    void (*handle_cc_func)(struct socket *),
    void (*init_cc_func)(struct socket *),
    size_t user_data_size,
    char  *err);
```

Parameters

int domain
The communication domain.

Passed directly to the underlying system call to create a socket, the `domain` decides which protocol family to use for communication. Because the DA-LBE control functionality is only implemented for TCP, the domain must be one that supports TCP. This is however not checked as valid domains may differ between environments.

const char *proto

The protocol which to control.

`proto` Decides which of the supported protocols to control. With Cubic and Vegas being the only protocols supported, the only valid arguments are `"cubic"` and `"vegas"`[2].

`size_t size`
The estimated size of the data transfer.

The `size` parameter is used by functions that rely on it being a positive value; as `size_t` is defined to be unsigned, 0 is the only invalid value.

`time_t deadline`
The deadline of the data transfer, as a delta from the current time.

The deadline must be strictly longer than one update interval; a deadline shorter than this would render DA-LBE functionality irrelevant, as by the time the control function is called for the first time the deadline has already passed.

`void (*handle_cc_func)(struct socket *)`
The congestion controller function for the daemon to call on each update interval. May be `NULL`.

The function `handle_cc_func` is called by the daemon on each update interval, after having updated the DA-LBE statistics required to perform metacontrol. If the argument is `NULL`, an appropriate default controller for the given protocol `proto` is used.

The specifics of this function and its signature is elaborated on in section 3.5.

`void (*init_cc_func)(struct socket *)`
A function with which to initialize the data structure used by the supplied function `handle_cc_func`. May be `NULL`.

The function `init_cc_func` is called after the necessary control structures for the socket in Libdalbe have been created. The supplied function will initialize the DA-LBE control structures to a reasonable initial state for the controller function `handle_cc_func`.

If the argument is `NULL` the data structure is not initialized.

`size_t user_data_size`
Size of the data area to be used by the supplied control function `handle_cc_func`. May be zero.

A storage area meant for use by the supplied control function is allocated and kept track of by Libdalbe, as described in section 3.5. This will typically be the size of the struct used for control information in the control function.

`char *err`
Destination for human-readable error messages generated by the library.

Return value
Zero if the function succeeded.

Returns negative if any of the parameters failed validation, in which case -1 is returned, or if the `socket` system call fails, in which case an error message is copied to `err` and the return value of `socket` is returned for further debugging.

Operation in detail

The new socket is put into the list of sockets shared with the daemon thread so that it will perform the regularly scheduled metacontrol operations. All settable DA-LBE parameters for the socket are set to sane default values: *off* for boolean parameters (typically turning a mode on or off), *zero* for parameters whose operation relies on a positive value, and *one* for scaling factors. Internal mechanisms are thus explicitly initialized, but not given values which would alter the underlying TCP in any way. The socket is set to operate in DA-LBE mode, that is `DA_LBE_MODE` is set to `DA_LBE_ENABLED` as per section 2.6.6. This results in the gathering of DA-LBE statistics and sets the congestion control up for metacontrol.

A.4 Closing a DA-LBE socket

DA-LBE Close

Signature

```
int dalbe_close(int socket, char *err)
```

Parameters

`int socket`
The socket to close. Must be a value previously returned by a call to `dalbe_open`.

`char *err`
Destination for human-readable error messages generated by the library.

Return value

Zero if the function succeeded.

Negative value if the socket is not one opened by Libdalbe, or if the closing of the underlying socket fails.

Operation in detail

The underlying socket file descriptor is closed, the socket is removed from the list of sockets, and associated resources are freed.

If the daemon was currently waiting on the deleted socket, it will skip to the next entry in the list.

[2] As given by the `name` field in the `tcp_congestion_ops` struct in the corresponding congestion control kernel module.

Appendix B

DA-LBE Framework Interface Details

The options listed here are values meant for the `optname` parameter of the `setsockopt` [35, `setsockopt(2)`] system call. With `level` set to `IPPROTO_TCP`, the option given by `optname` is set on an open TCP socket. The possible options are listed here for reference and for elaboration on points that are of importance to Libdalbe. See [70] for details.

B.1 Mid-Flow Control

The following are the options that control an arbitrary congestion control mid-flow:

DA-LBE Runtime Metacontrol Parameters

DA_LBE_INFO_ECN
Chance of triggering a phantom ECN for a received ACK.

DA_LBE_ECN_BACKOFF
Chance of ignoring a real ECN.

Can be used to tune a loss-based CC to compete more aggressively in competition with other ECN-enabled transports.

DA_LBE_CWND_BACKOFF
Chance for a delay-based CC to ignore `cwnd` backoff as a result of loss.

This flag is intended for delay-based congestion controls, specifically Vegas, to be able to compete better against aggressive loss-based CCs. Vegas is at a disadvantage against Cubic because they react differently to loss; Cubic backs off by a factor off β, typically 0.7^1[65], while Vegas is modeled after Reno which reduces the `cwnd` by a factor of 0.5 [52]. Setting this probability of ignoring backoff to a sensible value will let Vegas compete on par with Cubic.

DA_LBE_CONGESTION_PRICE
The factor by which the queuing delay is scaled.

Used to modify the measured queuing delay in order to control delay-based congestion controls. If `DA_LBE_BASE_RTT_BASED` is set, this scaling will only be performed on the extra queuing delay over the base RTT baseline.

DA_LBE_ECN_CONGESTION_DELAY
An integer scaling factor for the CC to determine whether it is not in competition with other traffic.

ECNs are not triggered if a long time has passed since the last real congestion indication, in an attempt to utilize free bandwidth if there is no competition. This scaling factor is the v in the calculation from section 2.6.3: $t_{\text{cong}} > v \times \tau_{\text{cong}}$.

DA_LBE_ECN_EWMA_WEIGHT

Weight factor for the exponentially weighted moving average being used to calculate the average time between congestion events.

The weight determines whether the moving average will react quickly to changing network conditions, or act more as a total average.

Though in real implementations the exact number will be one suited for integer arithmetic, $\frac{717}{1024}$ in the current Linux kernel [40, Version 4.16, `net/ipv4/tcp_cubic.c`, Line 47].

B.2 Flow Options

Some options are intended to be set once for every flow, to signal to the DA-LBE framework which type of congestion control is being run:

DA-LBE Mode Parameters

DA_LBE_MODE

Enable DA-LBE.

None of the control mechanisms are enabled if the mode of the flow is not set to `DA_LBE_ENABLED`. Some of the statistics are not gathered if this mode is off, making it nontrivial to use this as a toggle for DA-LBE behavior.

DA_LBE_BASE_RTT_BASED

Signal to the framework that the CC being controlled uses a base RTT measurement for determining congestion, i.e. it acts on extra queuing delay incurred on top of the base RTT.

Some congestion control schemes might want to adjust the entire measured RTT.

DA_LBE_DELAY_BASED_MODE

Signal to the framework that the CC being controlled is delay-based.

This enables the delay-tracking and adjustment mechanisms, as opposed to the default of only adjusting congestion signals by introducing phantom ECNs. Phantom ECNs can still be enabled on top of the delay control.

B.3 Available Statistics

Core statistics are tracked that facilitate DA-LBE control. These are structured such that the kernel does little or no extra calculations inside of the TCP implementation itself. The values meant to be averaged are reported as an aggregate value with a counter; the only statistic for which this is not the case is the moving average which is used frequently inside of the phantom ECN mechanism.

The available statistics are as follows:

DA-LBE Available Statistics

ECNs and phantom ECNs counts

RFC 3168 [53] states that TCP should not react to ECNs more than once per RTT. The receiving of real ECNs might compromise the effectiveness of phantom ECNs, as some congestion notifications might be ignored if arriving too close together.

Retransmissions

Extending upon the statistic offered by standard TCP, the framework provides counts of transmissions resulting from both fast and slow retransmission events, where fast retransmissions are *normal* packet retransmissions and slow retransmissions include a reset to slow start. Having access to different retransmission statistics can help more accurately gauge the state of network congestion.

Proportional difference in `cwnd` after reduction

A sum, as well as a count, of all of the differences in the congestion window after having been reduced due to excessive delay.

The model-based controller used for Vegas DA-LBE needs knowledge of the reduction in the `cwnd` per delay-based congestion event to be able to calculate the network congestion caused by delay. The controller, in the implementation provided with Libdalbe, lives in userspace and can utilize floating point arithmetic; the framework leaves this kind of heavy calculation up to the controller to save cycles in the TCP kernel code.

Average time between (real) congestion signals

This average is used to suppress phantom ECNs if there is little congestion.

Primarily used in the phantom ECN mechanism in the kernel, but may be used to infer the general state of network congestion long-term.

Appendix C

Test Setup

C.1 Cork Traceroute

```
$sender-a > traceroute -T cork
traceroute to cork (84.39.235.53), 30 hops max, 60 byte packets
```

			Probes		
#	Hostname	IP	#1	#2	#3
1	testbed	172.26.0.1	0.199 ms	0.175 ms	0.277 ms
2	10.174.0.1	10.174.0.1	0.578 ms	0.552 ms	0.522 ms
3	77.88.125.170	77.88.125.170	0.946 ms	1.523 ms	1.748 ms
4	81.175.33.17	81.175.33.17	0.976 ms	0.963 ms	1.083 ms
5	81.175.32.241	81.175.32.241	1.113 ms	1.099 ms	1.153 ms
6	ten-2-1-vl1504.00e121-070.as41572.net	81.175.32.226	1.786 ms	4.156 ms	4.097 ms
7	77.88.111.121	77.88.111.121	1.663 ms	3.508 ms	3.496 ms
8	ix-ge-1-0-1.hcore3.OS1-Oslo.as6453.net	80.231.89.29	11.744 ms	11.740 ms	11.728 ms
9	if-xe-8-3-1-0.tcore2.AV2-Amsterdam.as6453.net	80.231.152.41	23.318 ms	23.321 ms	23.378 ms
10	if-ae-2-2.tcore1.AV2-Amsterdam.as6453.net	195.219.194.5	23.605 ms	23.348 ms	22.918 ms
11	195.219.194.90	195.219.194.90	22.401 ms	22.396 ms	22.531 ms
12	xe-7-0-3.cro-lon8.ip4.gtt.net	141.136.105.113	30.174 ms	28.045 ms	30.109 ms
13	ip4.gtt.net	77.67.81.118	36.671 ms	37.127 ms	36.805 ms
14	91.103.0.202	91.103.0.202	38.481 ms	38.385 ms	38.347 ms
15	91.103.0.101	91.103.0.101	40.538 ms	40.392 ms	40.332 ms
16	84.39.235.50	84.39.235.50	38.109 ms	37.962 ms	38.092 ms
17	cork	84.39.235.53	41.415 ms	41.137 ms	41.203 ms

Table C.1: Traceroute from Fornebu, Norway to Cork, Ireland. Note the use of TCP for measurement – ICMP pings did not get replies from the entire route.

Appendix D

Analysis

D.1 Fairness Measurements

Cubic Cork	Vegas Cork	Cubic test bed	Vegas test bed
0.708	0.724	0.771	0.684
0.692	0.787	0.775	0.669
0.643	0.823	0.769	0.711
0.611	0.644	0.772	0.687
0.658	0.651	0.773	0.687
0.658	0.635	0.773	0.663
0.685	0.678	0.772	0.693
0.633	0.666	0.776	0.670
0.635	0.717	0.773	0.687
0.654	0.662	0.772	0.660
0.753	0.671	0.774	0.665
0.772	0.697	0.774	0.707
0.683	0.691	0.772	0.674
0.685	0.750	0.769	0.692
0.667	0.623	0.774	0.697
0.675	0.670	0.774	0.654
0.660	0.640	0.774	0.670
0.694	0.723	0.774	0.685
0.671	0.696	0.774	0.680
0.675	0.627	0.773	0.691
0.671	0.627	0.771	0.674
0.643	0.649	0.774	0.681
0.622	0.692	0.774	0.647
0.696	0.617	0.773	0.672
0.681	0.651	0.775	0.722
0.667	0.651	0.772	0.723
0.693	0.649	0.772	0.703
0.703	0.654	0.771	0.665
0.681	0.622	0.766	0.669
0.649	0.674	0.775	0.678
0.691	0.670	0.775	0.688
0.690	0.645	0.772	0.733
0.711	0.652	0.770	0.709
0.682	0.682	0.773	0.684
0.681	0.637	0.773	0.689
0.743	0.681	0.769	0.722
0.682	0.672	0.771	0.680
0.733	0.700	0.773	0.672
0.654	0.695	0.770	0.671
0.679	0.669	0.771	0.665
0.703	0.698	0.774	0.723
0.759	0.662	0.775	0.671
0.726	0.698	0.775	0.652
0.675	0.672	0.774	0.669
0.674	0.698	0.772	0.690
0.735	0.699	0.774	0.686
0.675	0.656	0.771	0.686
0.707	0.680	0.768	0.699
0.689	0.655	0.769	0.704
0.618	0.628	0.775	0.725

Table D.1: All fairness measurements

D.2 Completion Times Measurements

Cubic Cork	Vegas Cork	Cubic test bed	Vegas test bed
−12.8	−0.7	−2.9	119.0
−11.8	−1.7	−9.7	119.0
−9.3	−2.2	1.5	119.0
−10.4	−4.2	−5.3	119.0
−11.9	−2.8	−6.3	119.0
−16.1	−3.3	−7.4	119.0
−12.2	20.6	−3.6	119.0
−11.5	−2.0	−13.3	119.0
−12.1	−2.4	−5.8	119.0
−13.5	−2.0	−4.3	119.0
−12.2	−1.1	−9.1	119.0
−11.2	−2.7	−8.2	119.0
−6.3	2.4	−5.5	119.0
−12.1	−37.2	0.7	119.0
−10.9	−1.7	−9.5	119.0
−8.7	−10.0	−9.5	119.0
−11.3	−2.4	−9.1	119.0
−8.5	−1.3	−8.2	119.0
−19.5	2.1	−8.6	119.0
−17.2	−1.0	−5.9	119.0
−11.3	−1.7	−3.3	119.0
−12.3	−2.8	−9.1	119.0
−14.6	−1.0	−7.7	119.0
−11.9	−1.0	−7.4	119.0
−11.9	1.4	−10.5	118.5
−15.5	−0.9	−5.4	115.4
−14.3	−2.8	−4.2	119.0
−11.7	−0.4	−2.4	119.0
−16.3	4.2	6.7	119.0
−16.6	−1.6	−9.7	119.0
−13.2	−0.1	−11.2	119.0
−16.3	−2.8	−5.1	85.4
−11.5	−1.7	−1.2	119.0
−12.7	−1.7	−5.7	119.0
−17.1	−1.1	−7.2	119.0
−13.3	−2.3	2.4	117.6
−11.8	−9.7	−1.9	119.0
−12.7	3.0	−6.1	119.0
−17.7	−2.0	−1.3	119.0
−10.8	−0.4	−3.5	119.0
−10.7	0.1	−9.4	115.2
−10.1	−2.1	−9.6	119.0
−13.2	−0.7	−11.1	119.0
−17.1	−2.3	−7.7	119.0
−5.8	1.5	−5.2	119.0
−14.8	−2.6	−8.7	119.0
−11.5	−1.4	−3.5	119.0
−17.5	−1.3	2.9	119.0
−19.2	−4.0	1.1	119.0
−11.2	−1.5	−9.6	106.3

Table D.2: All completion time measurements

D.3 Vegas Cork Completion Times Libdalbe Debug

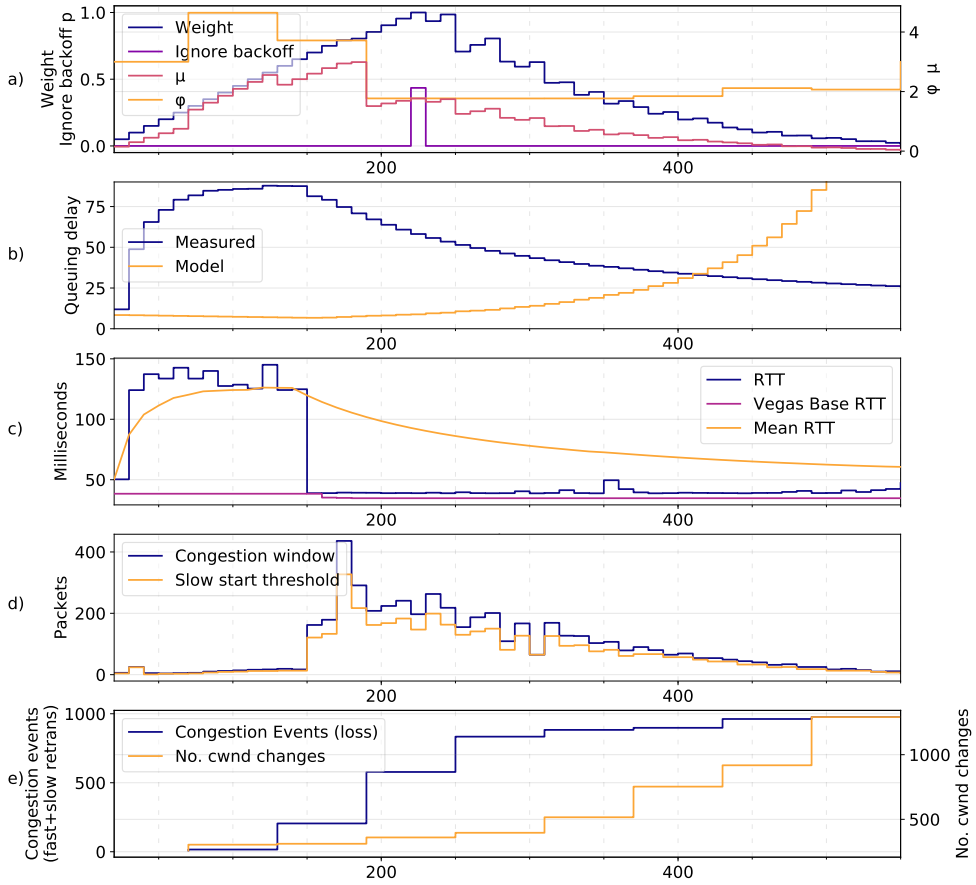


Figure D.1: Libdalbe debug graphs from the Vegas Cork test run that finished the earliest. A cutout from this graph is used in figure 5.14.

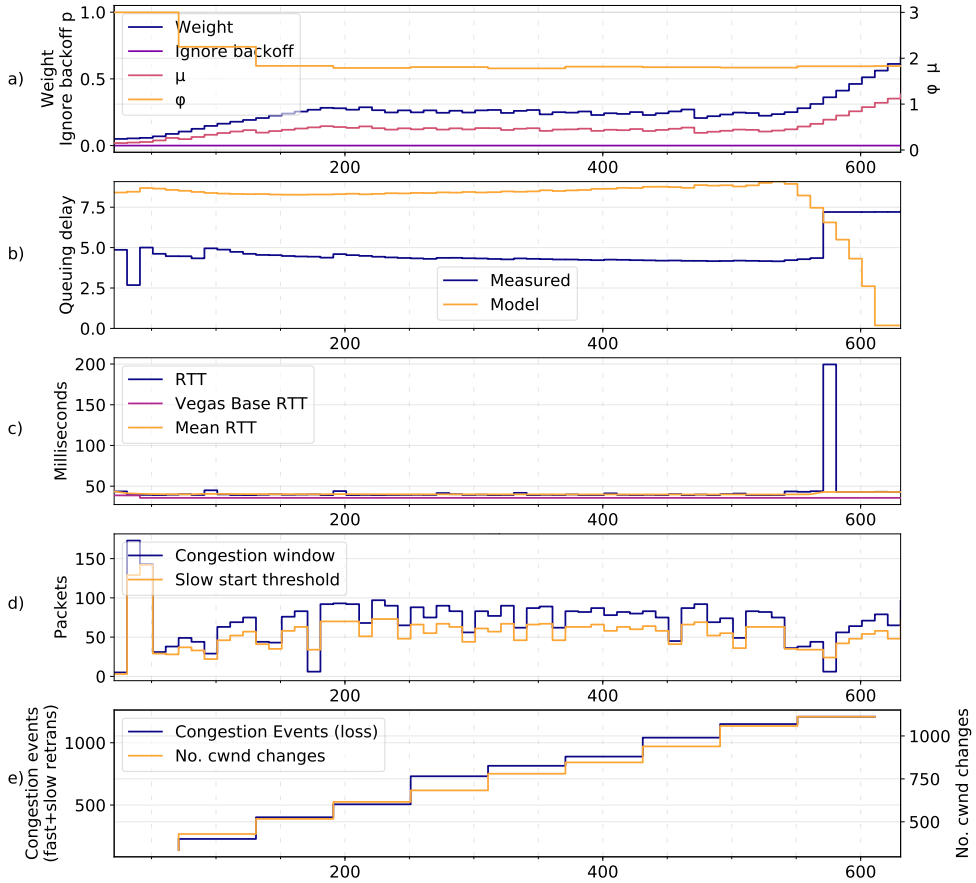


Figure D.2: Libdalbe debug graphs from the Vegas Cork test run that finished last. A cutout from this graph is used in figure 5.16.

D.4 Cubic Test Bed Completion Times Extra

The following graphs are all runs of the fairness experiment using Cubic on the test bed that finished after the deadline, excluding figure 5.17.

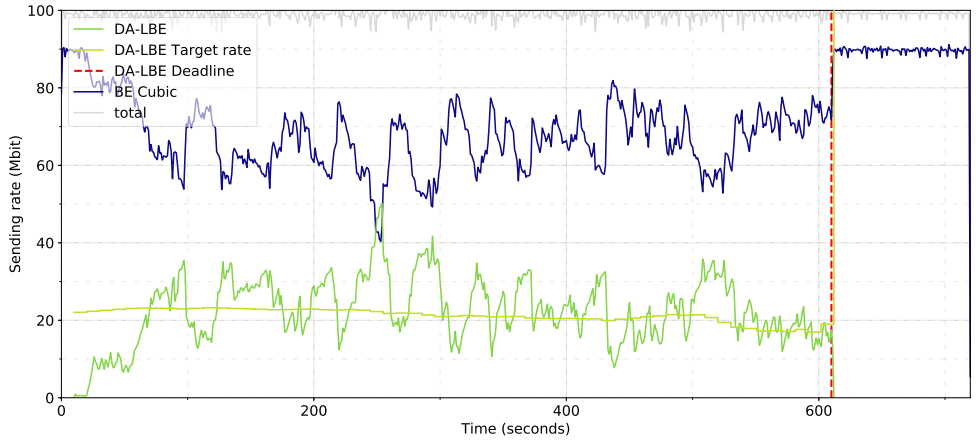


Figure D.3: Throughput graph of Cubic test bed fairness test no. 02.

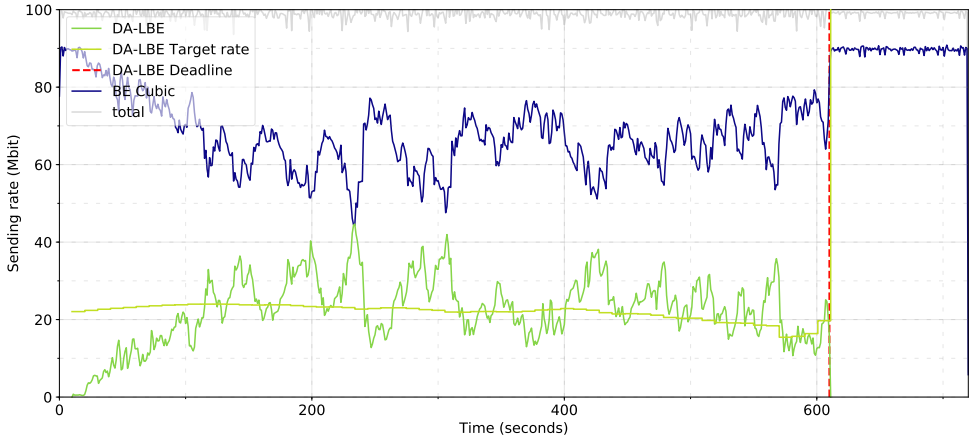


Figure D.4: Throughput graph of Cubic test bed fairness test no. 13.

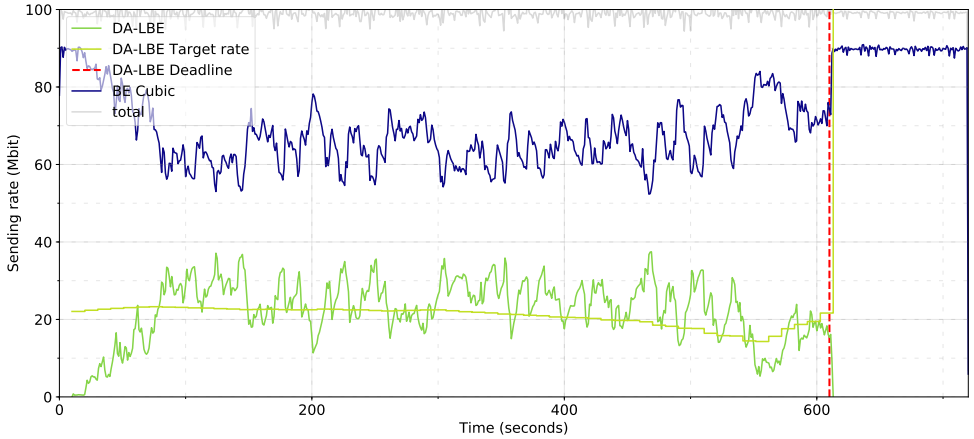


Figure D.5: Throughput graph of Cubic test bed fairness test no. 35.

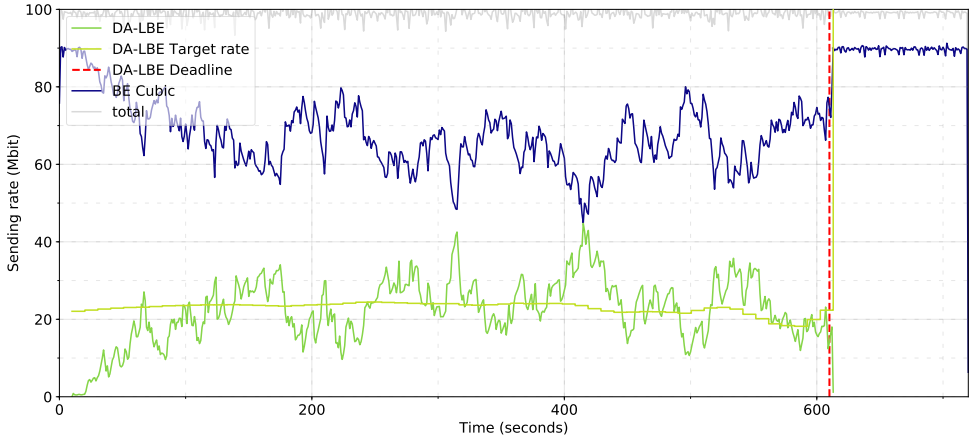


Figure D.6: Throughput graph of Cubic test bed fairness test no. 47.

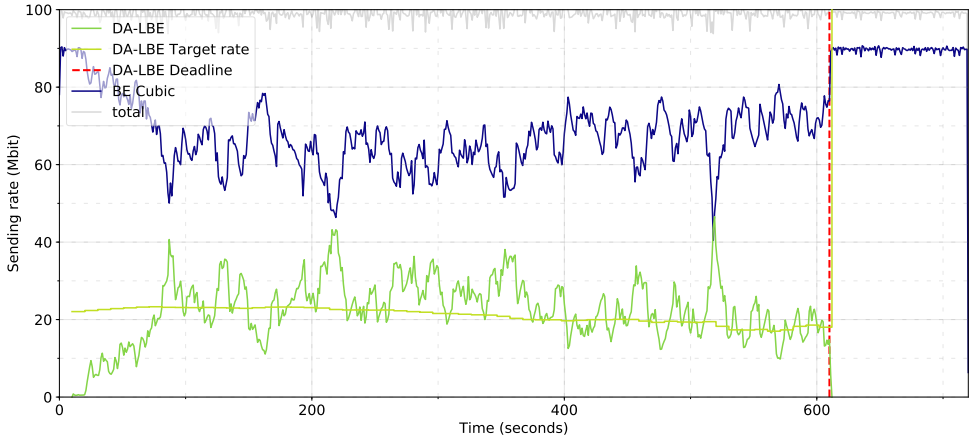
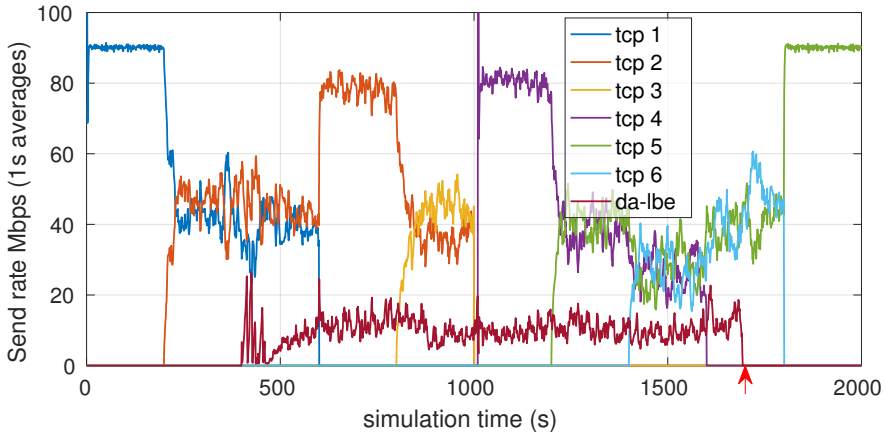


Figure D.7: Throughput graph of Cubic test bed fairness test no. 48.

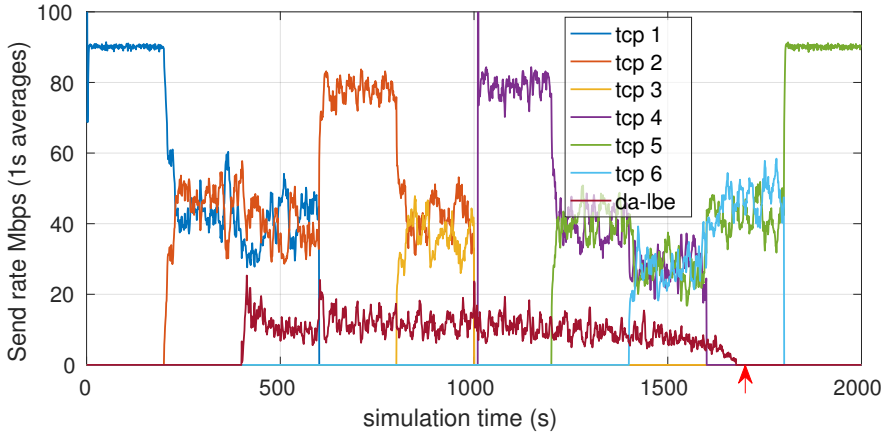
Appendix E

Graphs from Hayes *et al.*

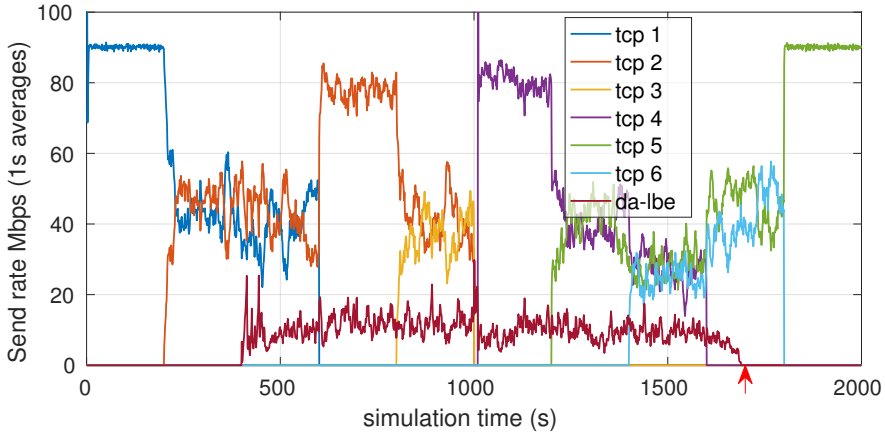
This appendix chapter contains figures copied from Hayes *et al.* [24] with permission.



(a) PID



(b) MBC



(c) MBC with w increase rate limiting

Fig. 4. NS2 simulation. Cubic TCP flows with a Cubic based DA-LBE phantom ECN flow. DA-LBE data size is based on 10% Capacity LBE rate.

Figure E.1: Figure 4 [24].

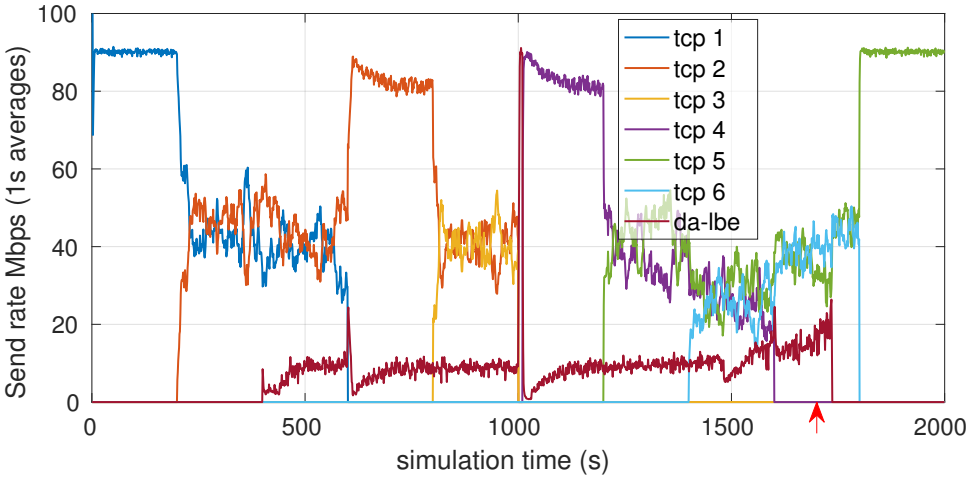


Fig. 8. Vegas MBC 10% Capacity LBE rate ($l_w = 0.05, l_\phi = 0.1$)

Figure E.2: Figure 8 [24].

Appendix F

Source Code for Libdalbe, Metacontrollers, and Test Orchestrator

Source code for Libdalbe, the metacontrollers, an example application, and the test orchestration suite can be found in a Bitbucket repository [77]. Access provided on request.