UNIVERSITY OF STAVANGER

BACHELOR THESIS

DYNAMIC ALTERNATIVE BACKOFF WITH ECN

Students
Dan Erik RAMSNES
Erlend Moen AL-KASIM

Supervisor
Naeem KHADEMI

May 15, 2020

Abstract

The Internet has evolved beyond its original purpose to send mail or use the World Wide Web (WWW). Recent years have seen a rapid increase in the use of latency sensitive applications such as online gaming, video streaming and highly time-sensitive financial transactions. Despite this, the Internet today still rely on lost packets as feedback for moderating the transmission rate of a host. When reliable delivery is necessary, which is still the case today with the bulk traffic using Transmission Control Protocol (TCP), packet loss is directly responsible for reducing throughput and consequently rising the latency due to the additional time needed for retransmission.

Explicit Congestion Notification (ECN) is a solution to combat this and has existed for two generations, but only recently gained a swift and wide adoption. Alternative Backoff with ECN (ABE) is a recent proposal that increases performance by utilizing ECN. We propose Dynamic Alternative Backoff with ECN (DABE) that improves ABE by using a dynamic reduction factor.

Contents

Abstract i

1 Introduction 1
- 1.1 Background and Motivation 1
- 1.2 Goals and Research Questions . . . 2
- 1.3 Research Methodology 2
- 1.4 Contributions 2
- 1.5 Thesis Structure 2

2 Background 4
- 2.1 Network Delay and Latency 4
- 2.2 Transmission Control Protocol . . . 4
  - 2.2.1 Congestion Control 5
  - 2.2.2 Bufferbloat 7
  - 2.2.3 TCP Variants 8
- 2.3 Active Queue Management 9
  - 2.3.1 Controlled Delay 9
  - 2.3.2 Proportional Integral Controller Enhanced 10
- 2.4 Explicit Congestion Notification . 10
- 2.5 Alternative Backoff with ECN . . . 11

3 Methodology 13
- 3.1 System Overview 13
  - 3.1.1 The Raspberry Pi 3 Cluster 13
  - 3.1.2 Network Topology 14
  - 3.1.3 Zyxel GS1920-8HP 15
- 3.2 General Setup 16
  - 3.2.1 Gateway 16
  - 3.2.2 Router 19
  - 3.2.3 Hosts 21
- 3.3 TCP Experimentation with TEACUP 22
- 3.4 Congestion Control in FreeBSD . . 23
- 3.5 Achieving Low Latency with ABE 26
- 3.6 Dynamic Alternative Backoff with ECN 27

4 Experimental Evaluation 30
- 4.1 Experimental Design 30
- 4.2 Performance 30
  - 4.2.1 Comparing Against ABE . 30
- 4.3 Intra-Fairness 34
  - 4.3.1 Two Flows 34
  - 4.3.2 Four Flows 36
- 4.4 Mixed Flows 38
  - 4.4.1 DABE and ABE 38
  - 4.4.2 DABE and CUBIC 40
  - 4.4.3 DABE and NewReno 41
  - 4.4.4 DABE, ABE and CUBIC . . 43
- 4.5 Evaluation 45

5 Conclusion 47
- 5.1 Future Work 47
- 5.2 Setbacks 47

A The PI3-Cluster Testbed 49
- A.1 Installing an Operating System . . 49
- A.2 Keyboard Layout 49
- A.3 Root Account 49
- A.4 Updating the System 50
- A.5 Enable SSH 50
- A.6 Change Hostname 51
- A.7 Time Synchronization 51

B TEACUP 53
- B.1 Configurations 53
  - B.1.1 Performance 53
  - B.1.2 Mixed Flows 57
- B.2 Utility Bash Scripts 62

C FreeBSD CC Modules 65
- C.1 NewReno 65
- C.2 Dynamic ABE 73

Terms 83

References 87

Chapter 1

Introduction

This chapter aims at giving an introduction and overview of the thesis. It starts with a brief explanation of why Internet today still feels slow despite major advances in technology, followed up by establishing the goals and research questions for the thesis. To address the research questions, a small look into the research methodology is presented. In the final section, an outline of the thesis structure is given.

1.1 Background and Motivation

The Internet has evolved beyond its original purpose to send mail or use the WWW. New expectations from end users are rapidly increasing by the demand for interactive applications such as online gaming, audio or video streaming. A common ground for these services is that they are sensitive to latency. Despite the growing use of such services, the Internet is still suffering from poor latency and performance. The culprit is widely known as *bufferbloat* [1], the existence of excessively large and frequently full buffers inside the network causing the dreaded notion of *lag*.

A key factor for the frequently full buffers is the dominant use of the TCP as the main communication protocol for the Internet. Although TCP itself is not the problem, a part of it became a major contributor. One of the main reasons for TCP dominance is Congestion Control (CC) — a fundamental set of mechanisms for maintaining the stability and efficiency of the Internet. [2] However, most mechanisms in use today still rely on lost packets as feedback for moderating the transmission rate of a host. This has served to add fuel to the bufferbloat problem, as standard CC will fill up any buffer in the network until a packet loss is inferred. Constantly pushing excessively large buffers to its limits causes packets to become queued for long periods of time, resulting in high latencies.

To combat bufferbloat, the proposal of active queue management (AQM) on intermediary devices was introduced in order to minimize the time packets spend enqueued at a bottleneck. By dropping a packet inside a buffer before it becomes full, the CC would trigger a rate reduction before pushing the buffer to its limits. Building upon AQM came the ability to explicitly signal congestion in the network, called ECN. Although ECN was standardized two decades ago, it has not seen much support due to compatibility issues with existing network equipment. However, recent years have shown a rapid change on this matter as the adoption of ECN on end-systems has accelerated and is now supported on the majority of servers. [3]

The benefits of ECN are evident. Instead of reacting to network congestion after the fact (i.e. after packet loss), one can take measures before it happens. When reliable delivery is necessary, packet loss is directly responsible for reducing throughput and consequently rising the latency due to the additional time needed for retransmission. While the reason for a packet loss can be of multiple sources besides a full buffer, explicit feedback using ECN serves as a clear signal for the inevitable impending of network congestion.

ABE [4] is a recent proposal that clearly shows the benefits of using explicit congestion feedback — by reducing the sender's transmission rate upon the receipt of an ECN mark, a packet loss can be avoided.

In addition, the earlier congestion feedback allows for a less aggressive reduction factor, yielding a higher sustained throughput and lower latency while also maintaining the benefit of avoiding packet loss.

1.2 Goals and Research Questions

In this section, the goal and research question for the thesis is presented along with a brief motivation.

> **Goal statement:**
> *Designing and implementing an improved version of ABE on FreeBSD.*

The goal of the thesis is to improve ABE by making it more responsive to different network conditions. Currently, ABE shares the same behaviour as NewReno when responding to a congestion signal, which is to reduce a sender's transmission rate by a constant factor. The only difference is that ABE reduces by less. Hence, the following research question defined below will be investigated.

> **Research question:**
> *ABE reduces a sender's transmission rate by a constant factor in the event of an ECN congestion signal. Can this factor be made more dynamic such that it reduces less when there is moderate congestion, and more when there is heavy congestion?*

Since ABE backs off by a constant factor, the thesis will look into techniques to make this reduction factor more dynamic. The ECN congestion response should back off less when there is moderate congestion, and back off more when there is heavy congestion.

1.3 Research Methodology

To address the research question outlined above, a cluster consisting of eight Raspberry Pi machines has been set up. The cluster serves as a physical testbed for conducting various TCP experiments in order to validate the results from our modifications to ABE.

1.4 Contributions

The thesis' main contribution is improving ABE by making it more responsive to various network conditions through the use of a more dynamic reduction factor in the event of an ECN congestion signal. We have also set up a physical testbed using a cluster of Raspberry Pi machines in order to test the validity of our improvements to ABE. In addition, we have documented the set up for others who are interested in making the TCP Experiment Automation Controlled Using Python (TEACUP) work on an ARM architecture.

1.5 Thesis Structure

The thesis consists of five chapters. This chapter presents the introduction and motivation for the research question. Chapter 2 gives an insight into the background knowledge that the thesis builds upon, and serves as an aid for those unfamiliar with the topic. Chapter 3 describes how the physical testbed has been set up, how our work has been done and the implementation details of it. Chapter 4 presents the results from our work along with an evaluation of the findings. Chapter 5 concludes the thesis with

an outline of future considerations and a final section discussing the struggles we encountered during the project.

Chapter 2

Background

This chapter presents the background knowledge that the thesis builds upon. We start by introducing the various components of network delay, and follow up by presenting the main protocol used today for transmitting data on the Internet. The later sections goes into both somewhat old and more recent technologies that has emerged in order to improve the performance of Internet.

2.1 Network Delay and Latency

The time it takes for a bit of data to travel across the network from one communication endpoint to another is known as *delay*. The process for such a transmission involves many components. A typical example where the data traverses an intermediate device before reaching destination follows. First, the data to be sent is usually created by an application. The data will then be handed over to the Operating System (OS) which passes it further down to the network interface card. From there, the data will be encoded and transmitted over a physical medium and eventually received by an intermediate device, such as a router. The router will then analyze the data and retransmit it over another medium that points to the destination. Finally, the data reaches the receiver. The whole process can happen in either multiples or fractions of seconds.

Network delay is therefore divided into the following four parts:

- Processing delay — time it takes a router to process the packet header
- Queuing delay — time the packet spends in routing queues
- Transmission delay — time it takes to push the packet's bits onto the link
- Propagation delay — time for a signal to reach its destination

It is common to notify the sender that the receiver actually got the data. This is done by sending a signal from the receiver back to the sender, known as an acknowledgement (ACK). The total time it takes for a sender to send data *and* receive back an ACK is known as *latency* or Round Trip Time (RTT).

In this thesis, we are mainly concerned with the *queuing* delay part.

2.2 Transmission Control Protocol

Whenever a user sends an email, or any data over the network for that matter, one should expect some kind of assurance that the delivery of the data was successful. This notion is known as *reliability*, and is one of the key components of the TCP and why it is one of the main protocols for transmitting data on the Internet. In essence, TCP is a communication protocol that provides reliable, ordered, and error-checked delivery of data between applications such as WWW, email and file transfer.

The main job of the TCP is to package data into *segments* or *packets*, send them and then reassemble them on the receiving side. To ensure the reliability of these segments, a *sequence number* is contained

in each of them so that the receiver side can reassemble them in correct order. The sequence numbers also act as a safeguard when some packet has been lost, as the receiver will know what is missing. If the sender has not received an ACK within what is known as a *timeout* interval, the missing data will be retransmitted.

Another distinguishing feature of TCP is the notion of *flow control*. Network devices have finite resources, and so some will be much faster than others. One device may be sending data at a much faster rate than the receiver can handle. TCP solves this by using a *sliding windows* mechanism which the sole purpose of controlling the flow of data between two devices such that one is not overwhelmed.

2.2.1 Congestion Control

In the same sense that traffic on the road can come to a halt, the same is true for traffic on a network. This is known as *congestion*, and is usually caused by overutilization. That is, network devices such as a router have finite resources, and thus too much traffic will cause the device to carry more data than it can handle which leads to congestion on the network. Typical effects include queueing delay, packet loss or the blocking of new connections.

The notion of congestion led to another key compononent of TCP, which is the ability to either prevent congestion or mitigate it after it occurs. The means of applying Congestion Control (CC) is simple — ensure that the sender does not overflow the network. In other words, the sender's rate needs to be adjusted based on the condition of the network. Now to the hard part — how to adjust it?

To shed some light on this, TCP includes a state variable called Congestion Window (CWND) which limits the amount of data that a sender can send before receiving an ACK back. This variable is known only to the sender, and serves as the key role for attaining CC. The means of adjusting CWND is referred to as an *congestion-control algorithm*, and consists of four intertwined parts:

- Slow start — gradually increase the amount of data transmitted until the capacity of the network has been reached.
- Congestion avoidance — slowly probe for available bandwidth.
- Fast retransmit — perform retransmission isntead of waiting for a timeout.
- Fast recovery — enter congestion avoidance phase instead of slow start after a fast retransmit.

The following sections will describe each part in more detail.

Slow Start and Congestion Avoidance

Slow start and congestion avoidance are two independent algorithms with different objectives, but in practice are implemented together and so will be discussed together.

Upon a TCP connection, the CWND value is initialized to a single *segment* with a specified size. This size is either announced by the the other end, known as the Receiver Window (RWND), or set to a typical default [5]. In the same sense that CWND is a sender-side limit, RWND is a receiver-side limit. Together, the minimum of CWND and RWND governs the data transmission in slow start. However, at the beginning of a transmission, the network conditions are unknown, so the goal of the slow start phase is simple — slowly probe the network to determine the available capacity.

To find the capacity, slow start works as follows — the sender starts by transmitting one segment. For every received ACK, increment the CWND by another segment, effectively doubling it every RTT. Re-

peat this process until a timeout occurs. That is, until the capacity has been reached that is signaled by a router starting to drop packets, which tells the sender that its CWND has grown too large.

As a response to hitting the threshold, TCP will now halve the CWND since that is the last known value to not induce a timeout. This new value is known as the slow start threshold (ssthresh), and is another TCP state variable used to determine whether the slow start or congestion avoidance algorithm is used to control the data transmission. In other words, the slow start phase ends when CWND reaches or exceeds ssthresh.

The goal of congestion avoidance is also simple — avoid congestion, as the name implies. But the challenge here is to maintain a transmission rate that it not too low, otherwise the link becomes underutilized, and at the same time probe for available bandwidth without overflowing the network too quickly. To do so, a feedback control algorithm known as additive-increase/multiplicative-decrease (AIMD) is used.

Let $w(t)$ be the sending rate during time slot t, let a be a positive number and b a number between 0 and 1. The sending rate can then be modeled as AIMD by

$$w(t+1) = \begin{cases} w(t) + a, & \text{if congestion is not detected,} \\ w(t) \times b, & \text{if congestion is detected.} \end{cases} \tag{2.1}$$

where a is the additive increase factor, and b is the multiplicative decrease factor.

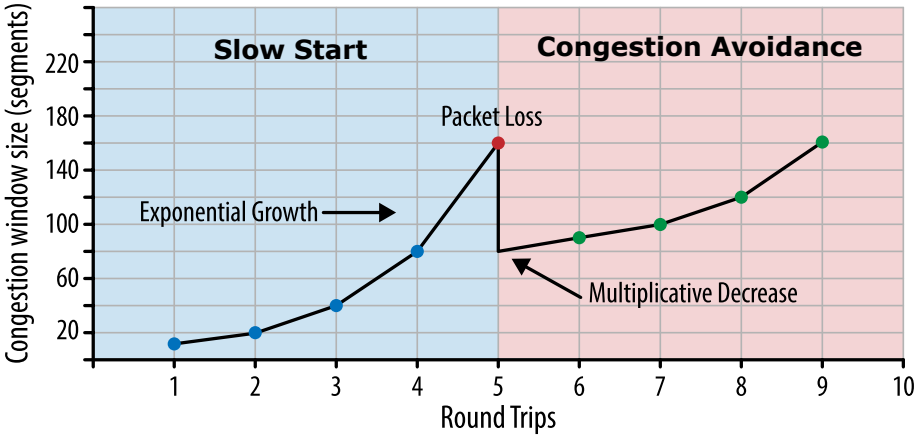


Figure 2.1: The two main parts of CC — the slow start phase where CWND grows exponentially, and the congestion avoidance phase where the transmission rate is increased more conservatively. (*source*)

AIMD provides linear growth of the CWND when probing for available bandwidth, and a multiplicative reduction when congestion is detected.

Fast Retransmit and Fast Recovery

Fast retransmit and fast recovery are two algorithms that aims to speed up the recovery process of a TCP connection in the event of a segment loss, but without re-entering the slow start phase. As with

slow start and congestion avoidance, both are in practice implemented together and so will be discussed together.

In a TCP connection, the sender uses a simple timer to recognize lost segments. That is, if an ACK is not received within a specified time, the sender will assume that the segment has been lost and so will retransmit it. Fast retransmit is an enhancement to TCP that reduces the time a sender waits before retransmitting a lost segment. The duration is determined by the amount of duplicate ACKs, which the receiver will keep sending to indicate the next expected sequence number. Generally, if only one or two duplicate ACKs are experienced, it is assumed that a simple reordering of the segments will solve the problem. But when three or more duplicate ACKs are received in a row, it signals a strong indication that a segment has been lost. In such a case, TCP will perform a retransmission instead of waiting for a timeout, known as a *fast transmit*.

When a fast retransmit has occured, the sender will transmit what appears to be the missing segment. However, the slow start phase will not be performed after this, but rather the congestion avoidance phase. This is known as the *fast recovery* algorithm. Since only a segment seems to be missing, it would be wasteful to redue the traffic flow abruptly by going into slow start. Instead, a fast recovery is initiated, allowing for sustained high throughput under moderate congestion.

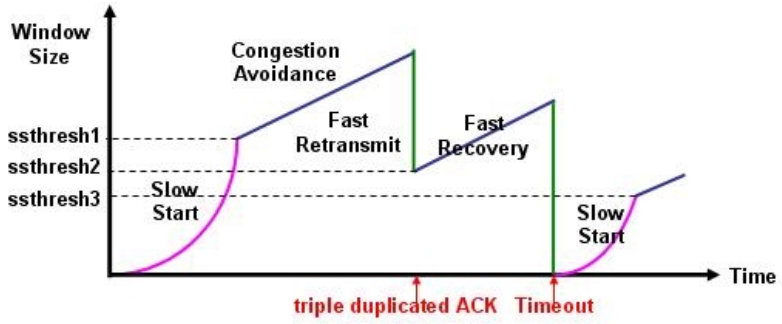


Figure 2.2: A more complete illustration of a congestion-control algorithm including all four parts — slow start, congestion avoidance, fast retransmit and fast recovery. (*source*)

2.2.2 Bufferbloat

It is easy to fall into the trap of believing that bigger buffers would increase network performance. After all, why drop a packet when you can avoid it by a buffer that never gets full in the first place. Likewise, it is easy to see why a big buffer is bad — it takes time to drain the queue. If more packets come in than can be transmitted, the longer the queue gets. As a result, the latency spikes. Excess buffering of packets is referred to as *bufferbloat*, causing high latency and reducing the overall network throughput.

Still, some buffering is clearly needed so that a short burst of packets can be absorbed without significant loss. But how much? Ideally, for a particular network, we would want to keep the bottleneck link as busy as possible. A widely used formula in determining the buffer size B is the bandwidth delay product (BDP) [6]:

$$B = RTT_{avg} \times C \tag{2.2}$$

where RTT_{avg} is the average RTT of the flow passing through the link and C is the capacity of the link.

However, if the average RTT is relatively high such as 250 ms and a router has a Gigabit Ethernet interface, then the buffer size would at least be $B = 250\,\text{ms} \times 1\,\text{Gbit} = 32\,\text{MB}$. Such large buffers have been shown to greatly contribute to the bufferbloat problem, especially on backbone routers that uses slow, off-chip DRAMs. Nick McKeown et. al. from [6] showed that dividing the BDP by the square root of number of flows yielded a radical better estimate for buffer sizing, so far as a 99% reduction in buffer size with negligible difference in throughput. In other words, the better formula for estimating the buffer size is then

$$B = \frac{RTT_{avg} \times C}{\sqrt{n}} \tag{2.3}$$

where n is the amount of flows on a given link.

2.2.3 TCP Variants

The two first TCP CC algorithms that became widely used was called Tahoe and Reno. Both uses packet loss as a congestion signal, and both react by halving their CWND. With the increase in network speeds, other loss-based algorithms were proposed. One of these were called NewReno, which improved the retransmission during the fast recovery phase of Reno. NewReno became widely popular, and is currently the default CC algorithm used on FreeBSD today. NewReno follows the behavior described in Section 2.2.1 and produces the classical *sawtooth* pattern shown in the following Figure 2.3.

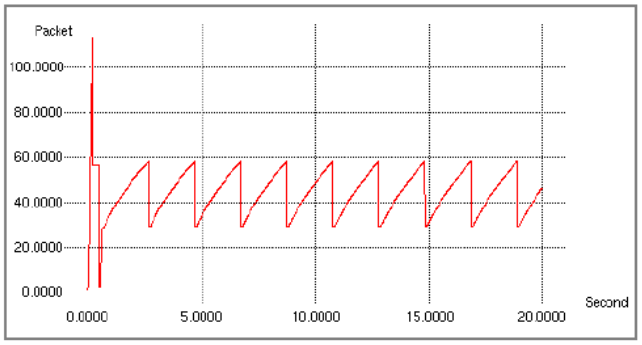


Figure 2.3: The classic *sawtooth* pattern of CWND over time using the TCP congestion control algorithm NewReno.

Another CC algorithm that became popular was CUBIC, aiming to optimize the congestion control in high bandwidth networks with high latency. CUBIC was implemented and used by default in Linux kernel versions 2.6.19 and above, which persists to this day. As its name implies, CUBIC does not use a linear function to adjust its CWND, but rather a cubic function as shown in the following Figure 2.4.

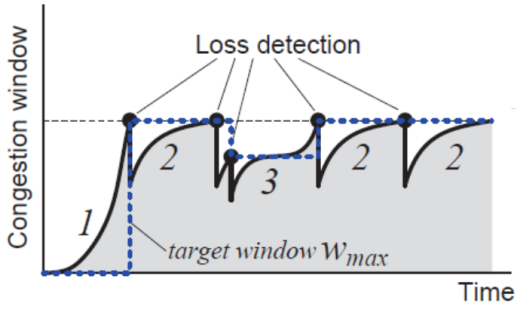


Figure 2.4: The behavior of the CUBIC congestion control algorithm.

2.3 Active Queue Management

When a packet arrives at the incoming port of a router, it is often the case that there is not enough memory to buffer it, and so an important decision must be made — should the newly arrived packet simply be dropped, or should one of the already-queued packets be dropped to make room for the new one? In the early days of networking, the queue management algorithm called *tail drop* was used, simply dropping newly arrived packets when the queue was filled. However, this approach is considered unfair among traffic flow due to its *first come, first serve* basis, leaving little room for new participants since a single connection could monopolize the queue space. [7] Tail drop also lead to what is known as *TCP global synchronization*, where all connections would both simultaneously "hold back" and then step forward, causing networks to become underutilized and flooded in alternate waves.

Over the years, many new suggestions and algorithms were made. They collectively became known as active queue management (AQM) algorithms — the policy of dropping packets *before* a buffer becomes full. One of the first and most widely implemented algorithm to surface was random early detection (RED), addressing the issues of tail drop. By preemptively dropping packets *probabilistically* before the buffer became full, RED would avoid both global synchronization and the problem of new bursty connections being penalized. [8]

Widespread usage of RED also revealed a core problem — it was difficult to configure, requiring complex fine-tuning in order to achieve significant performance gains. Variants of RED were developed to accommodate for this, such as BLUE, and later many new AQM schemes. A few selected will be discussed briefly in the next sections.

2.3.1 Controlled Delay

Controlled Delay (CoDel) is a more recent AQM scheme developed by Van Jacobsen. It is designed to overcome bufferbloat in modern networking environments by limiting, or *controlling*, the delay that network packets experience when passed through buffers in networking hardware. An implementation of CoDel for the Linux kernel was written in 2012 by Dave Täht and Eric Dumazet. It has since been adopted as the standard AQM for the widely known OpenWrt OS used by many routers.

Van Jacobsen himself asserted in 2006 that existing AQMs have been using incorrect means of recognizing bufferbloat. [9] Algorithms such as RED measures the average queue length and considers it a case of bufferbloat if the average grows too large. Jacobsen demonstrated in 2006 that this is not necessarily the case, as a quick burst also exhibits the same behaviour. In fact, Jacobsen would say that queue

length was no indicator at all about packet demand or network load, [9, 10] suggesting instead that the minimum queue length during a sliding time window would be a better metric. [10]

Jacobsen would therefore properly combat bufferfloat in CoDel by distinguishing between two types of queue — a *good* queue that exhibits no bufferbloat, and a *bad* queue that actually exhibits bufferbloat. The CoDel approach, in summary, is designed to meet the following goals: [11]

- Make AQM parameterless for normal operation.
- Be able to distinguish a good queue from a bad one in order to keep delay low while allowing necessary bursts of traffic.
- Control delay while insensitive to RTT delays, link rates, and traffic loads — all without harming the network.
- Adapt to dynamically changing link rates with no negative impact on utilization.
- Allow simple and efficient implementation.

2.3.2 Proportional Integral Controller Enhanced

Proportional Integral Controller Enhanced (PIE) is another recent AQM algorithm to control queuing latency directly in order to address the bufferbloat problem. It was made by a group of authors from Cisco Systems as a lightweight alternative to CoDel, claiming that CoDel cannot practically be implemented in hardware — first, CoDel requires packets to be timestamped when enqueued, and second, the drop decision happens on dequeue, so all packets have to be queued. PIE, on the other hand, performs the drop decision on enqueue, similar to RED and most other AQM schemes. Hence, PIE incurs very little overhead and is simple enough to implement in both hardware and software.

The PIE approach, much like CoDel, is designed to meet the following goals: [12]

- Control queuing latency instead of queue length to combat bufferbloat.
- Attaining a delicate balance between high link utilization and low latency.
- Simple to implement and easily scalable in both hardware and software by striving to maintain similarity to that of RED.
- Ensure system stability for various network topologies and scale well across an arbitrary number of streams.
- Design parameters should be set automatically. Users only need to set performance-related parameters such as target queue latency, not design parameters.

2.4 Explicit Congestion Notification

Conventially, the means of detecting network congestion has been by packet loss or a long enough delay inducing a timeout. Then the addition of AQM came to the Internet infrastructure, adding the ability for routers to detect congestion before the queue overflows. Both loss and delay are *implicit* signals of congestion, and in 2001, an extension to TCP and Internet Protocol (IP) was defined [13] to allow for *explicit* congestion signaling — called ECN — enabling end-to-end notification of network congestion without dropping packets.

ECN is a form of network-assisted CC, meaning that the entire underlying network infrastructure must support it. For this to work, the new ECN mechanism consists of two components:

- Two new bits has been added to the IP header in order to explicitly communicate ECN-capability and indication of congestion on the network (see table 2.1).
- Two new flags has been added to the TCP header — the ECN-Echo (ECE) flag so that the receiver can inform a sender when a Congestion Encountered (CE) packet has been received, and the Congestion Window Reduced (CWR) flag so that a sender can inform the receiver that the CWND has been reduced.

ECN Code Points

Code	Description	Shorthand
00	Non ECN-Capable Transport	Non-ECT
01	ECN Capable Transport	ECT(0)
10	ECN Capable Transport	ECT(1)
11	Congestion Encountered	CE

Table 2.1: The four different code points for ECN that is encoded in the IP header.

An ECN-capable network works as follows. A sender will transmit IP packets with either the ECT(0) or ECT(1) bit set. Then, when there is congestion, a router (in conjuction with AQM) will start marking the packets by setting the CE bit. The receiver will then see this, and echo back the congestion indication to the sender with an ECE packet. The sender will then reduce its CWND value and respond with an CWR packet.

ECN was standardized in 2001 as a direct replacement for the use of packet loss as congestion signaling. [13] Despite its positive early impact [14], it never saw widespread usage. [3] A series of unfortunate incidents managed to cripple its early deployment — rather than responding properly or ignoring the ECN bits, some firewalls on faulty network equipment would instead block or mangle packets with ECN bits set. This incorrect behaviour led to failure rate of 8% in ECN-enabled TCP connections for web servers tested in 2004. [15]

Due to the unfortunate early struggles of ECN deployment, the Internet Engineering Task Force (IETF) required that the response to an ECN signal should be the same as the reaction to a loss. [13] However, this requirement has recently been relaxed, [16] sparking a regained interest in the field and giving rise to new experimental ECN proposals. As of 2015, measurements show that the failure of ECN-enabled TCP connections has reduced to less than 1%. [3]

2.5 Alternative Backoff with ECN

Alternative Backoff with ECN (ABE) [4] is a recent proposal utilizing ECN in congestion control. It is a simple sender-side only modification that can be summarized as follows:

- Upon a packet loss, a TCP sender should reduce its CWND as usual (e.g. 50% with NewReno or 30% with CUBIC).
- Upon the receipt of an ECN mark, a TCP sender should reduce *less* than the usual response for loss.

The less aggressive multiplicative decrease factor yielded significant performance gains in lightly-multiplexed scenarios. Specifically, a 20% reduction factor was found to give a good latency-vs-utilization trade-off.

As of 2018, ABE has been formalized by the IETF as an experimental Request for Comments (RFC) [17] and been implemented in mainline FreeBSD OS kernel.

Chapter 3

Methodology

This chapter describes the architecture and methods that were needed in order to investigate the research question presented in Section 1.2. An overview of the physical testbed will given, followed by the general setup for each machine and how the TCP experimentation have been conducted using the tool TEACUP. The later sections briefly explain the internals of congestion control development on FreeBSD, how to use ABE and the implementation details for our work upon improving ABE.

3.1 System Overview

The objective of the system is to perform automated TCP experiments in order to validate the reproducibility and significance of our results. This section aims to give a brief overview of the system as a whole and its individual components.

3.1.1 The Raspberry Pi 3 Cluster

The system consists of eight connected Raspberry Pi machines. The Raspberry Pi machine is a series of small single-board computers developed in the UK by the Raspberry Pi Foundation to promote teaching of basic computer science in schools and in developing countries. The Raspberry Pi 3 Model B+ was released in 2018, and is the model used in this cluster.

All the machines are hooked up to a single Zyxel switch. The switch provides interconnectivity, and in addition powers the machines through Power over Ethernet (PoE).

Figure 3.1: The Raspberry Pi 3B+ cluster all hooked to a Zyxel switch.

Raspberry Pi 3 Model B+	
CPU	Broadcom BCM2837B0 Cortex-A53 (ARMv8) 64-bit SoC 1.4GHz Quad-core
RAM	1GB LPDDR2 SDRAM
NICs	Gigabit Ethernet over USB 2.0 Dual IEEE 802.11ac WiFi, Bluetooth 4.2
FS	32GB Micro-SD
USB	4 USB 2.0 ports

Table 3.1: Raspberry Pi 3 Model B+ hardware specifications.

3.1.2 Network Topology

The testbed consists of two networks — the *controller* network in which all machines are connected directly, and the *experimental* network separated by two subnets with a router in-between.

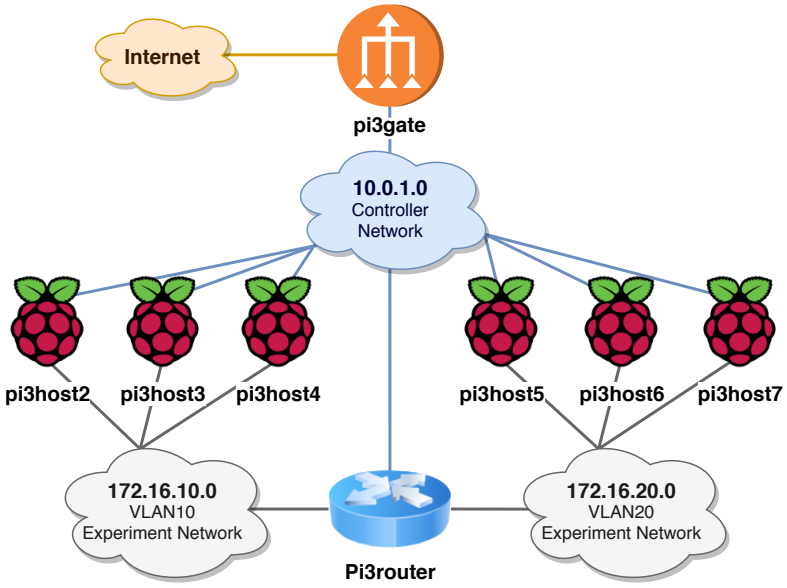


Figure 3.2: The logical network topology for the Raspberry Pi 3B+ cluster testbed.

Since the entire testbed is connected to a single switch only, and each Raspberry Pi 3B+ machine has only one Ethernet interface, both virtual LAN (VLAN) and virtual interfaces have been used to achieve the desired network topology presented above. In addition, one USB-to-Ethernet adapter was necessary on the router in order to conduct network experimentation with TEACUP. The VLAN setup on the switch is explained in Section 3.1.3 while the general network setup for every machine is explained in the remaining sections.

A summary table of the network setup for each machine follows.

Summary Network Setup		
Hostname	**IP address**	**OS**
pi3router	10.0.1.1 — eth0 172.16.10.1 — eth0:10 (virtual) 172.16.20.1 — eth1 (USB-to-eth)	Raspbian Buster (Linux)
pi3host2	10.0.1.2 — eth0 172.16.10.2 — eth0:10 (virtual)	FreeBSD 12.1
pi3host3	10.0.1.3 — eth0 172.16.10.3 — eth0:10 (virtual)	FreeBSD 12.1
pi3host4	10.0.1.4 — eth0 172.16.10.4 — eth0:10 (virtual)	FreeBSD 12.1
pi3host5	10.0.1.5 — eth0 172.16.20.5 — eth0:20 (virtual)	FreeBSD 12.1
pi3host6	10.0.1.6 — eth0 172.16.20.6 — eth0:20 (virtual)	FreeBSD 12.1
pi3host7	10.0.1.7 — eth0 172.16.20.7 — eth0:20 (virtual)	FreeBSD 12.1
pi3gate	DHCP — eth0 10.0.1.254 — eth0:1 (virtual)	Raspbian Buster (Linux)

Table 3.2: The hostnames, IP addresses and OS for each Raspberry Pi 3B+ machine in the cluster.

The table above lists the configured Raspberry Pi 3B+ machines in left to right order as seen from 3.1. That is, the left most machine is set up as the router, and the right most machine is set up as the gateway.

3.1.3 Zyxel GS1920-8HP

The Zyxel GS1920-8HP model is a managed switch with 10 available ports for interconnectivity. The Zyxel switch is what powers every Raspberry Pi 3B+ machine using PoE.

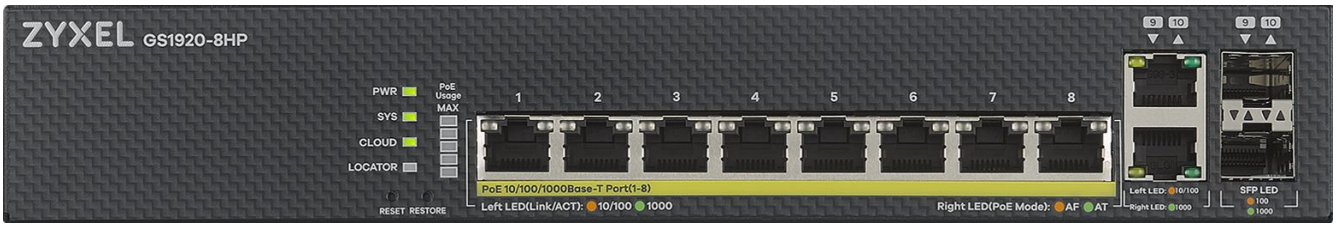
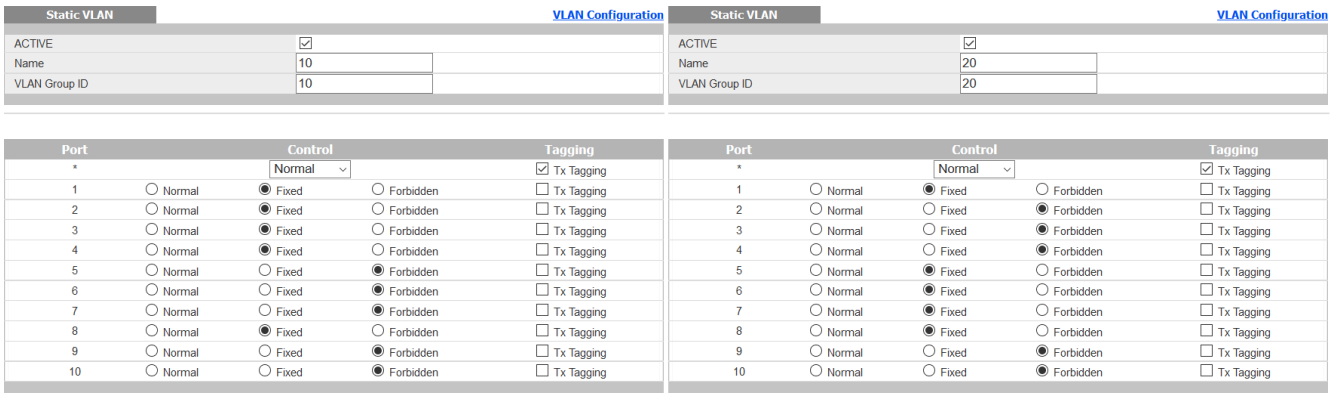


Figure 3.3: The front of the Zyxel GS1920-8HP managed switch.

In order to logically separate two subnets on a single switch, as illustrated in 3.2, two VLANs must be set up. This can easily be done through the switch's graphical user interface that can be accessed in several ways as shown in the official quick start guide [1]. In short, connect a computer to the Zyxel switch with an Ethernet cable, set a static IP of 192.168.1.10, and access the graphical user interface in a web browser by visiting the address 192.168.1.1.

[1] https://www.zyxel.com/products_services/8-24-48-port-GbE-Smart-Managed-Switch-GS1920-Series

The default username and password for Zyxel GS1920-8HP is admin and 1234, respectively. Once logged in, go to Advanced Application -> VLAN -> VLAN Configuration -> Static VLAN Setup.



(a) VLAN 10 **(b)** VLAN 20

Figure 3.4: The configuration for the two VLANs on Zyxel GS1920-8HP.

Create two VLANs as shown above, confirming each creation with the Add button. When done, persist the changes with the Save button in the top right corner.

3.2 General Setup

This section serves as a guide for reproducing the general setup and configuration for the testbed. For the next sections, the following common tasks are assumed to have already been done on each Raspberry Pi 3B+ machine:

- Installed an OS as specified in A.1. Raspbian Buster is used on the gateway and router, and FreeBSD on the hosts.
- Changed the keyboard layout as specified in A.2.
- Created a root user account as specified in A.3.
- Updated the system as specified in A.4.
- Enabled SSH as specified in A.5.

In addition, any given command is assumed to be run as root user.

3.2.1 Gateway

This section describes how the machine with direct Internet access has been set up, known as the *gateway*, and how it provides access and Internet for the other machines through Network Address Translation (NAT). The gateway is also known as the *controller*, as this is where TEACUP experiments are both configured and conducted from.

Network

The gateway is the only machine with direct Internet access through Dynamic Host Configuration Protocol (DHCP) on its main interface, with a virtual interface statically connected to the private controller

network in order to communicate with the rest of the machines. To set up this, add the following to the /etc/network/interfaces file:

```
# Main interface (Internet access)
auto eth0
iface eth0 inet dhcp

# Subinterface (controller-network)
auto eth0:1
iface eth0:1 inet static
address 10.0.1.254
netmask 255.255.255.0
```

In order for the machines on the internal network to gain Internet access through the gateway, both IP forwarding and NAT must be set up:

```
# Enable IP forwarding
echo 'net.ipv4.ip_forward=1' >> /etc/sysctl.conf

# NAT
update-alternatives --set iptables /usr/sbin/iptables-legacy
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
apt install iptables-persistent
```

The hostname has also been changed to pi3gate as specified in A.6, followed by a reboot to apply all network changes.

NTP Server

TEACUP requires that time is synchronized on all machines when running experiments. In order for the internal machines to synchronize their clock, an Network Time Protocol (NTP) server must be set up. The gateway will provide this service, and is set up as specified in A.7.

TEACUP

Every network experiment orchestrated by TEACUP is initiated from the gateway, and so the majority of the tools for TEACUP to work are installed on this machine. As described from the official install guide [2], with some slight adjustments, the following commands will install TEACUP properly:

```
# R
apt install -y r-base

# PDFJAM
apt install -y texlive-extra-utils
```

[2] http://caia.swin.edu.au/tools/teacup/TEACUP-0.9_INSTALL.txt

```
# SPP
apt install -y mercurial libpcap-dev build-essential
hg clone https://bitbucket.org/caia-swin/spp
cd spp
make
mkdir /usr/local/man/man1
make install
cd ..

# Fabric
apt install -y fabric python-pip python-dev libffi-dev libssl-dev
pip install fabric3
pip install -Iv pexpect==3.2

# TEACUP
wget https://sourceforge.net/projects/teacup/files/teacup-1.1.tar.gz
tar -xf teacup-1.1.tar.gz
```

To verify that TEACUP has been properly installed, a simple check can be done as follows:

```
mkdir experiment
cp teacup-1.1/example_configs/config-scenario1.py experiment/config.py
cp teacup-1.1/run.sh experiment/
cp teacup-1.1/fabfile.py experiment/
```

Go into the experiment folder and prepare to edit the config.py file. Find the line containing TPCONF_script_path = '/home/teacup/teacup-0.8' and change the path to where you extracted teacup-1.1. Finally, verify TEACUP installation with the command fab check_config. If no errors appear, all is good.

Modifications to TEACUP

Due to the restricted physical set up as shown in 3.1 and discussed in 3.1.2, virtual interfaces have been used extensively, particularly on the router. This introduced a conflict to the TEACUP codebase, as it assigns a name to each interface before performing network logging with tcpdump. However, this name is not unique since both the controller and experiment network shares the same interface, resulting in a duplicate handle error from TEACUP. A slight adjustment to the codebase was therefore necessary.

Applying the following "band-aid" to the loggers.py file in TEACUP should fix the error:

```
--- teacup-1.1/loggers.py
+++ teacup-1.1-modified/loggers.py
@@ -41,6 +41,7 @@
    from getfile import getfile
    from runbg import runbg

+   import random

    ## Collect all the arguments (here basically a dummy method because we
```

```
    ## dont used the return value)
@@ -505,7 +506,7 @@
                     snap_len, interface, file_name, tcpdump_filter)
         pid = runbg(tcpdump_cmd)

-        name = 'tcpdump-' + interface
+        name = 'tcpdump-' + interface + str(random.randint(0, 50000))
         #bgproc.register_proc(env.host_string, name, '0', pid, file_name)
         bgproc.register_proc_later(
             env.host_string,
```

In additon, a few changes to the hostsetup.py file in TEACUP must be done. TEACUP checks if the sysctl variable net.inet.tcp.reass.overflows exists, which it does not for the FreeBSD version that we are using. The default sender and receiver buffer on FreeBSD hosts must also be set to a higher value, otherwise experiments would experience a capped CWND value. The changes are easily done as follows:

```
--- teacup-1.1/hostsetup.py
+++ teacup-1.1-modified/hostsetup.py
@@ -858,7 +858,7 @@

     if htype == 'FreeBSD':
         # record the number of reassembly queue overflows
-        run('sysctl net.inet.tcp.reass.overflows')
+        #run('sysctl net.inet.tcp.reass.overflows')

         # disable auto-tuning of receive buffer
         run('sysctl net.inet.tcp.recvbuf_auto=0')
@@ -869,6 +869,8 @@
         # send and receiver buffer max (2MB by default on FreeBSD 9.2 anyway)
         run('sysctl net.inet.tcp.sendbuf_max=2097152')
         run('sysctl net.inet.tcp.recvbuf_max=2097152')
+        run('sysctl net.inet.tcp.recvspace=655360')
+        run('sysctl net.inet.tcp.sendspace=327680')

         # clear host cache quickly, otherwise successive TCP connections will
         # start with ssthresh and cwnd from the end of most recent tcp
```

Finally, the tool spp does not run on the Raspberry Pi 3 or ARM architecture at all, only displaying the helpful message "aborting" when running it. TEACUP uses this tool to calculate RTT, and fortunately is only needed when analyzing the results. In other words, the experiment data can easily be copied over to an x86 machine where spp works, and then be analyzed from there. That is, follow the install instructions above from 3.2.1 again as normal, but on an x86 machine instead.

3.2.2 Router

This section describes how the router has been set up.

Network

The router serves as the intermediate device for the hosts in order to conduct more realistic experiments. The main interface is statically connected to the controller network, with two additional interfaces (one virtual and one USB-to-Ethernet) that function as the default gateway for the two separate subnets that the hosts reside in. To set the router up as such, add the following to the /etc/network/interfaces file:

```
# Main interface (controller-network)
auto eth0
iface eth0 inet static
address 10.0.1.1
netmask 255.255.255.0
gateway 10.0.1.254

# Subinterface for VLAN 10 (experiment-network)
auto eth0:10
iface eth0:10 inet static
address 172.16.10.1
netmask 255.255.255.0
gateway 10.0.1.254

# USB-to-eth for VLAN 20 (experiment-network)
auto eth1
iface eth1 inet static
address 172.16.20.1
netmask 255.255.255.0
gateway 10.0.1.254
```

In order for the hosts to communicate through the router, IP forwarding must be enabled:

```
# Enable IP forwarding
echo 'net.ipv4.ip_forward=1' >> /etc/sysctl.conf
```

The hostname has also been changed to pi3router as specified in A.6, followed by a reboot to apply all network changes.

NTP Client

To synchronize time for the router against the gateway, set the router up as an NTP client as specified in A.7.

TEACUP

The router only needs two tools for TEACUP to work properly when controlled from the gateway, and that is tcpdump and ntp:

```
# TEACUP tools on router
apt install -y tcpdump ntp
```

3.2.3 Hosts

This section describes how each host has been set up.

Network

The hosts are separated into two subnets. Hosts 2, 3 and 4 are on the network 172.16.10.0/24 (VLAN 10), while hosts 5, 6 and 7 are on 172.16.20.0/24 (VLAN 20). The main interface on each host is statically connected to the controller network, with an additional virtual interface (or *alias* in FreeBSD) statically connected to the experimental network. To set up each host, run the following, but replace yy with the appropriate VLAN and x with the correct host number as illustrated in 3.1.2:

```
# Main interface (controller-network)
echo 'ifconfig_ue0="inet 10.0.1.x netmask 255.255.255.0"' >> /etc/rc.conf
echo 'defaultrouter="10.0.1.254"' >> /etc/rc.conf

# Alias (experiment-network)
echo 'ifconfig_ue0_alias0="inet 172.16.yy.x netmask 255.255.255.0"' >> /etc/rc.conf
```

In order for the hosts to communicate between each subnet, a static route must be added. Run the following to do so:

```
# Static route
echo 'static_routes="intnet"' >> /etc/rc.conf
echo 'route_intnet="-net 172.16.yy.0/24 172.16.xx.1"' >> /etc/rc.conf
```

Replace yy with the destination VLAN and xx with the source VLAN.

The hostname has also been changed to pi3hostX as specified in A.6, where X refers to the correct host number. A reboot will apply all network changes.

NTP Client

To synchronize time for each host against the gateway, set up each host as an NTP client as specified in A.7.

TEACUP

TEACUP requires four tools that must be installed on each host in order to run experiments, namely lighttpd, iperf, httperf and nttcp. The lighttpd tool can be installed from the package repository with pkg install lighttpd. A modified version of iperf is used, and so must be installed from source as follows:

```
# Installing modified iperf on FreeBSD
wget https://sourceforge.net/projects/iperf2/files/iperf-2.0.9.tar.gz --no-check-certificate
wget http://caia.swin.edu.au/tools/teacup/downloads/iperf-2.0.5-mod.tar.gz
tar -xf iperf-2.0.9.tar.gz
tar -xf iperf-2.0.5-mod.tar.gz
cd iperf-2.0.9
patch -p1 < ../iperf-2.0.5-mod/iperf-2.0.5.patch
./configure
make
make install
```

The patching process will produce a tiny conflict. This issue is easily resolved by manually applying the changes.

A modified version of httperf is also used, but installing from source has proven unsuccessful. Thus, an unmodified version is therefore installed with pkg install httperf. The last tool to install, nttcp, is also modified, and so must be installed from source as follows:

```
# Installing modified nttcp on FreeBSD
wget http://caia.swin.edu.au/tools/teacup/downloads/nttcp-1.47-mod.tar.gz
tar -xf nttcp-1.47-mod.tar.gz
cd nttcp-1.47-mod
make
cp nttcp /bin/
```

3.3 TCP Experimentation with TEACUP

TCP Experiment Automation Controlled Using Python (TEACUP) [3] is a software tool for running automated TCP experiments in a controlled physical tested. It was made as a research project by Centre for Advanced Internet Architectures (CAIA) in 2005 to gain a better understanding of the impact of different CC algorithms on TCP-based streaming flows. [18] The typical use-case involves a classical dumbbell topology where multiple hosts are connected on opposite sides of a bottleneck router. An essential part of the tool is the *control host* or *gateway* as we refer to it, which is where TEACUP itself runs and orchestrates the configuration of the end hosts and bottleneck router before a particular experiment is conducted.

The process of installing and setting up TEACUP is quite extensive. We experienced a lot of problems due to the ARM architecture on Raspberry Pi. However, the effort was well worth it. We ended up with a single OS on each machine, where the hosts are running FreeBSD and the router and gateway are running Raspbian Buster (Linux). With a single configuration file (see Appendix 3.2.1), TEACUP is able to perform experiments with multiple permuations — traffic generation (such as TCP bulk transfer), traffic shaping (such as bandwidth, delay and loss), traffic control (scheduling of packets with AQM) and executing custom host commands such as enabling ECN.

[3] http://caia.swin.edu.au/tools/teacup

For each experiment, TEACUP will collect a variety of metadata from the end hosts and bottleneck router. The resulting experiment data is then analyzed in order to generate graphical plots that can show either the throughput, RTT (including smoothed) or CWND over time. The use of TEACUP in our thesis serves to validate the results from our work, as every plot illustrated in Chapter 4 has been generated by it.

3.4 Congestion Control in FreeBSD

The FreeBSD networking stack saw a major enhancement with the release of version 9.0 — modular congestion control was added, allowing CC algorithms to be implemented as loadable kernel modules. The framework is referred to as mod_cc [4], and was the result of a research project called NewTCP from CAIA that began in 2005. It initially focused on independent, interoperable implementations of new "high speed" TCP CC algorithms in FreeBSD, and eventually received FreeBSD Foundation support in order to port its modular congestion control framework and five congestion control algorithms into the official FreeBSD tree. [5]

The available CC algorithms on FreeBSD can be listed with the following command:

```
# Show available CC algorithms on FreeBSD
sysctl net.inet.tcp.cc.available
```

By default, only NewReno is available. For others to be available, they must first be loaded with the dynamic kernel linker facility (kld):

```
# Load FreeBSD CC module with kld
kldload cc_cubic
```

FreeBSD comes with several CC kernel modules. They reside in /boot/kernel with the prefix cc_. On our system, the following modules (.ko-files) are found by default on the hosts:

```
# Output of 'ls /boot/kernel | grep cc_' on FreeBSD 12.1-RELEASE:
cc_cdg.ko
cc_chd.ko
cc_cubic.ko
cc_dctcp.ko
cc_hd.ko
cc_htcp.ko
cc_vegas.ko
```

Once a CC module has been loaded, it will show up as available and can be selected with:

```
# Select which CC algorithm to use on FreeBSD
sysctl net.inet.tcp.cc.algorithm=cubic
```

[4] https://www.freebsd.org/cgi/man.cgi?query=mod_cc&sektion=4&apropos=0&manpath=FreeBSD+12.1-RELEASE
[5] http://caia.swin.edu.au/urp/newtcp

In order to compile a FreeBSD kernel module, the kernel source is needed. The source must reflect the current installed system, which in our case means that it can be fetched as follows:

```
# Get FreeBSD kernel source
VERSION=12.1-RELEASE
ARCHITECTURE=arm64
fetch ftp://ftp.freebsd.org/pub/FreeBSD/releases/$ARCHITECTURE/$VERSION/src.txz
tar -C / -xzvf src.txz
```

With the kernel source in place, the implementation code for existing CC modules on FreeBSD can be found in /usr/src/sys/netinet/cc with its corresponding Makefile in /usr/src/sys/modules/cc in order to build it.

The mod_cc framework consists of the header file cc.h and implementation file cc.c. Once included in a kernel module, a set of hook functions becomes available that are called in various TCP events. These functions are encapsulated in a cc_algo structure in cc.h, and together represent a congestion control algorithm in FreeBSD.

HOOK FUNCTIONS IN CC.H

```
/*
 * Structure to hold data and function pointers that together represent a
 * congestion control algorithm.
 */
struct cc_algo {
    /* The name that uniquely identifies the algorithm */
    char name[TCP_CA_NAME_MAX];
    /* Init global module state on kldload. */
    int (*mod_init)(void);
    /* Cleanup global module state on kldunload. */
    int (*mod_destroy)(void);
    /* Init CC state for a new control block. */
    int (*cb_init)(struct cc_var *ccv);
    /* Cleanup CC state for a terminating control block. */
    void (*cb_destroy)(struct cc_var *ccv);
    /* Init variables for a newly established connection. */
    void (*conn_init)(struct cc_var *ccv);
    /* Called on receipt of an ack. */
    void (*ack_received)(struct cc_var *ccv, uint16_t type);
    /* Called on detection of a congestion signal. */
    void (*cong_signal)(struct cc_var *ccv, uint32_t type);
    /* Called after exiting congestion recovery. */
    void (*post_recovery)(struct cc_var *ccv);
    /* Called when data transfer resumes after an idle period. */
    void (*after_idle)(struct cc_var *ccv);
    /* Called for an additional ECN processing apart from RFC3168. */
    void (*ecnpkt_handler)(struct cc_var *ccv);
    /* Called for {get|set}sockopt() on a TCP socket with TCP_CCALGOOPT. */
    int (*ctl_output)(struct cc_var *, struct sockopt *, void *);
    /* Macro that declares a structure that connects the elements in the tail queue. */
```

```
    STAILQ_ENTRY (cc_algo) entries;
};
```

Listing 3.1: An excerpt from cc.h showing the set of hook functions that allows congestion control algorithms to be implemented as a dynamically loadable kernel module in FreeBSD.

Any mod_cc module must instantiate the cc_algo structure, but only the name field is required to be defined. The instantiation is provided by the DECLARE_CC_MODULE(ccname, &ccalgo) macro used to register a module with the cc_mod framework, where ccname specifies the name of the module, and &ccalgo points to a structure containing the function pointer assignments for the hook functions. A simple example follows.

SIMPLE MOD_CC EXAMPLE

```c
#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/malloc.h>
#include <sys/module.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/sysctl.h>
#include <sys/systm.h>
#include <net/vnet.h>
#include <netinet/tcp.h>
#include <netinet/tcp_seq.h>
#include <netinet/tcp_var.h>
#include <netinet/cc/cc.h>
#include <netinet/cc/cc_module.h>

/* Define functions to be used. */
static int example_mod_init(void);
static int example_mod_destroy(void);

/* Assign name and hook functions to the defined above. */
struct cc_algo example_cc_algo = {
    .name = "example",
    .mod_init = example_mod_init,
    .mod_destroy = example_mod_destroy
};

/* Private struct with members that can be used throughout the module. */
struct example {
    uint32_t value;
};

/* Implementation of the mod_init hook function. */
static int example_mod_init(void) {
    printf("Hello, world!\n");
    return (0);
}
```

```c
/* Implementation of the mod_destroy hook function. */
static int example_mod_destroy(void) {
    printf("Goodbye, world!\n");
    return (0);
}

/* Register module to the cc_mod framework and instantiate example_cc_algo struct. */
DECLARE_CC_MODULE(example, &example_cc_algo);
```

Listing 3.2: A simple mod_cc module example where the name, mod_init and mod_destroy have been defined. The module simply prints "Hello, world!" when loaded, and "Goodbye, world!" when unloaded, which can be viewed with the command dmesg.

To access TCP related variables, the macro CCV(ccv, what) is used, where ccv points to the cc_var structure defined in cc.h (passed by many hook functions from 3.2), and what refers to the variable to be retrieved. For instance, the call CCV(ccv, snd_cwnd) will return the current CWND. For complete examples of CC kernel modules in FreeBSD, see Appendix C.

3.5 Achieving Low Latency with ABE

Since ABE is already in mainline FreeBSD kernel, [17] and the hosts in our testbed are running FreeBSD, it is only a matter of enabling it with the following command:

```
# Enabling ABE on FreeBSD
sysctl net.inet.tcp.cc.abe=1
```

NewReno in FreeBSD utilizes the AIMD feedback control algorithm for adjusting the CWND, meaning that it grows linearly when in congestion avoidance, and experiences an exponential reduction when congestion is detected. The multiplicative decrease factor is specified as a percentage, and can be changed with the following commands:

```
# Set multiplicative window decrease factor in FreeBSD for lossy signaling (default 50)
sysctl net.inet.tcp.cc.newreno.beta=50

# Set multiplicative window decrease factor in FreeBSD for ECN signaling (default 80)
sysctl net.inet.tcp.cc.newreno.beta_ecn=80
```

Upon a congestion signal with ABE enabled, the static reduction factor in NewReno is updated as follows:

```c
static void newreno_cong_signal(struct cc_var *ccv, uint32_t type) {
    /* ... */
    if (V_cc_do_abe && type == CC_ECN)
        factor = beta_ecn; /* factor if ECN */
    else
        factor = beta; /* factor if loss */
```

```
    /* ... */
    cwin = max(((uint64_t)cwin * (uint64_t)factor) / (100ULL * (uint64_t)mss), 2) * mss;
```

The new CWND calculation follows the multiplicative part of the AIMD scheme from 2.1, which effectively becomes

$$\text{cwnd} = \begin{cases} \text{cwnd} \times 50/100, & \text{if loss,} \\ \text{cwnd} \times 80/100, & \text{if ECN} \end{cases} \tag{3.1}$$

where cwnd is the current CWND. The resulting value is then scaled up by the Maximum Segment Size (MSS) to get the amount in bytes. Finally, the result is applied in a **switch** statement where either only the ssthresh is set in the case of a timeout, or both the ssthresh and current CWND is set in the case of ECN:

```
case CC_NDUPACK: /* packet loss or timeout */
    if (!IN_FASTRECOVERY(CCV(ccv, t_flags))) {
        /* ... */
        if (!IN_CONGRECOVERY(CCV(ccv, t_flags)))
            CCV(ccv, snd_ssthresh) = cwin;
        ENTER_RECOVERY(CCV(ccv, t_flags));
    }
    break;
case CC_ECN: /* ECN */
    if (!IN_CONGRECOVERY(CCV(ccv, t_flags))) {
        CCV(ccv, snd_ssthresh) = cwin;
        CCV(ccv, snd_cwnd) = cwin;
        ENTER_CONGRECOVERY(CCV(ccv, t_flags));
    }
    break;
```

The complete implementation details for NewReno can be found in Appendix C.1.

3.6 Dynamic Alternative Backoff with ECN

Dynamic Alternative Backoff with ECN (DABE) represents our work that builds upon ABE. Instead of reducing the CWND by a constant factor in the event of an ECN congestion signal, we reduce by the ratio of the shortest RTT measured and the most recent one. Hence, if the RTT rises, the resulting backoff is also greater, but if the RTT remains stable, the backoff is less. The shortest, or *minimum*, RTT serves as the optimal point in which the network experiences no congestion.

Our work follows ABE's spirit in that it is a simple sender-side only modification. In essence, we are only updating the ECN congestion response. From the multiplicative part of the AIMD scheme from 2.1, the new CWND value now becomes

$$\text{cwnd} = \begin{cases} \text{cwnd} \times 0.5, & \text{if loss,} \\ \text{cwnd} \times \dfrac{\text{RTT}_{min}}{\text{RTT}}, & \text{if ECN} \end{cases} \tag{3.2}$$

where RTT_{min} is the minimum RTT that has been measured and is continually updated, and the RTT is the most recent measure. In order to implement this, we need RTT measurements. This is true for any delay-based CC algorithm, and so FreeBSD comes included with such functionality — the Enhanced Round Trip Time (ERTT) *Khelp* module, referred to as h_ertt [6]. Khelp is another kernel framework in FreeBSD with the aim to provide a structured way to dynamically extend the kernel at runtime. The h_ertt works within the khelp framework to provide for enhanced RTT estimates for all new TCP connections created after the time at which the CC module was loaded. Both khelp and its h_ertt module first appeared in FreeBSD 9.0, made by the same people who worked on the NewTCP research project which brought modular congestion control to FreeBSD.

The h_ertt module provides several key pieces of data. In particular for us is the most *recent* RTT measurement and the *shortest* RTT measurement that has been taken.

THE ERTT STRUCTURE

```
/* Structure used as the ertt data block. */
struct ertt {
    /* Information about transmitted segments to aid in RTT calculation. (Private) */
    TAILQ_HEAD(txseginfo_head, txseginfo) txsegi_q;
    /* Bytes TX so far in marked RTT. (Private) */
    long bytes_tx_in_rtt;
    /* Final version of above. */
    long bytes_tx_in_marked_rtt;
    /* cwnd for marked RTT. */
    unsigned long marked_snd_cwnd;
    /* Per-packet measured RTT. */
    int rtt;
    /* Maximum RTT measured. */
    int maxrtt;
    /* Minimum RTT measured. */
    int minrtt;
    /* Guess if the receiver is using delayed ack. (Private) */
    int dlyack_rx;
    /* Keep track of inconsistencies in packet timestamps. (Private) */
    int timestamp_errors;
    /* RTT for a marked packet. (Private) */
    int markedpkt_rtt;
    /* Flags to signal conditions between hook function calls. */
    uint32_t flags;
};
```

Listing 3.3: An excerpt from h_ertt.h showing the fields for the ertt structure that is associated with each TCP connection. The private fields should not be manipulated by any code outside of the h_ertt implementation.

To make the h_ertt module available, the header files khelp.h and h_ertt.h must be included. To access the data from h_ertt, a unique ID must be generated and used. In addition, we must specify that the CC algorithm now depends on another module, which is done with the MODULE_DEPEND(name, moddepend, **int** minversion, **int** prefversion, **int** maxversion) macro. The implementation for this follows.

[6] https://www.freebsd.org/cgi/man.cgi?query=h_ertt&apropos=0&sektion=4&manpath=FreeBSD+12.1-RELEASE

```c
#include <sys/khelp.h>
#include <netinet/khelp/h_ertt.h>
/* ... */
static int32_t ertt_id;
/* ... */
static int newreno_dabe_init(void) {
    ertt_id = khelp_get_id("ertt");
    if (ertt_id <= 0) {
        printf("%s: h_ertt module not found\n", __func__);
        return (ENOENT);
    }
    return (0);
}
/* ... */
MODULE_DEPEND(newreno_abe, ertt, 1, 1, 1);
```

The h_ertt module is then used to fetch the minimum and most recent RTT in order to calculate the new CWND from 3.2 in the event of an ECN congestion signal. The implementation for this follows.

```c
static void newreno_abe_cong_signal(struct cc_var *ccv, uint32_t type) {
    /* ... */
    struct ertt *e_t = khelp_get_osd(CCV(ccv, osd), ertt_id);
    /* ... */
    switch (type) {
    /* ... */
    case CC_ECN:
        if (!IN_CONGRECOVERY(CCV(ccv, t_flags))) {
            CCV(ccv, snd_ssthresh) = CCV(ccv, snd_cwnd) * e_t->minrtt / e_t->rtt;
            CCV(ccv, snd_cwnd) = CCV(ccv, snd_ssthresh);
            ENTER_CONGRECOVERY(CCV(ccv, t_flags));
        }
        break;
    }
}
```

The CCV(ccv, snd_cwnd) call fetches the current CWND, and gets multiplied by e_t->minrtt / e_t->rtt which is the ratio of the minimum RTT and most recent RTT provided by the h_ertt module.

Chapter 4

Experimental Evaluation

In the following sections, all experiments conducted in order to resolve the research question in Section 1.2 will be presented, and later used as a basis for discussion in the final section.

4.1 Experimental Design

From our testbed, we will be using pi3host2, pi3host3 and pi3host4 as clients, with pi3host7 acting as a server. The clients will be sending bulk TCP data over the bottleneck router pi3router with a maximum bandwidth of 10 mbit and a buffer size of 64 packets. To provide a fair share of the bandwidth for all flows during experiments, we will use the Fair Queuing version of CoDel as the AQM on the bottleneck router.

4.2 Performance

In this section we will look at the performance of DABE by comparing it directly against ABE and CUBIC. Each experiment will only have a single flow so it can freely use the available bandwidth. The resulting data will then be plotted against each other in order to draw comparisons. The TEACUP configuration file can be found in Appendix B.1.1.

4.2.1 Comparing Against ABE

In this experiment we have run DABE and ABE individually ten times each with four different delays on the bottleneck router. The following Figure 4.1 shows the resulting CWND value over time for both DABE and ABE taken from separate experiments.

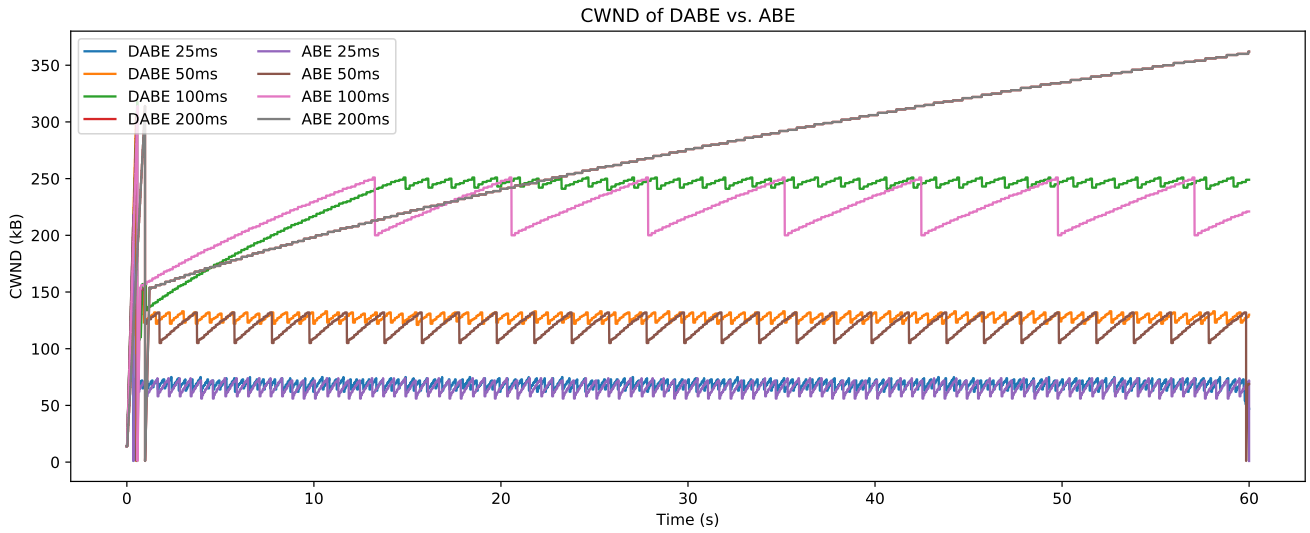


Figure 4.1: The CWND value for DABE vs. ABE taken from separate experiments under various delays.

Figure 4.2 shows the RTT value over time for both DABE and ABE taken from separate experiments.

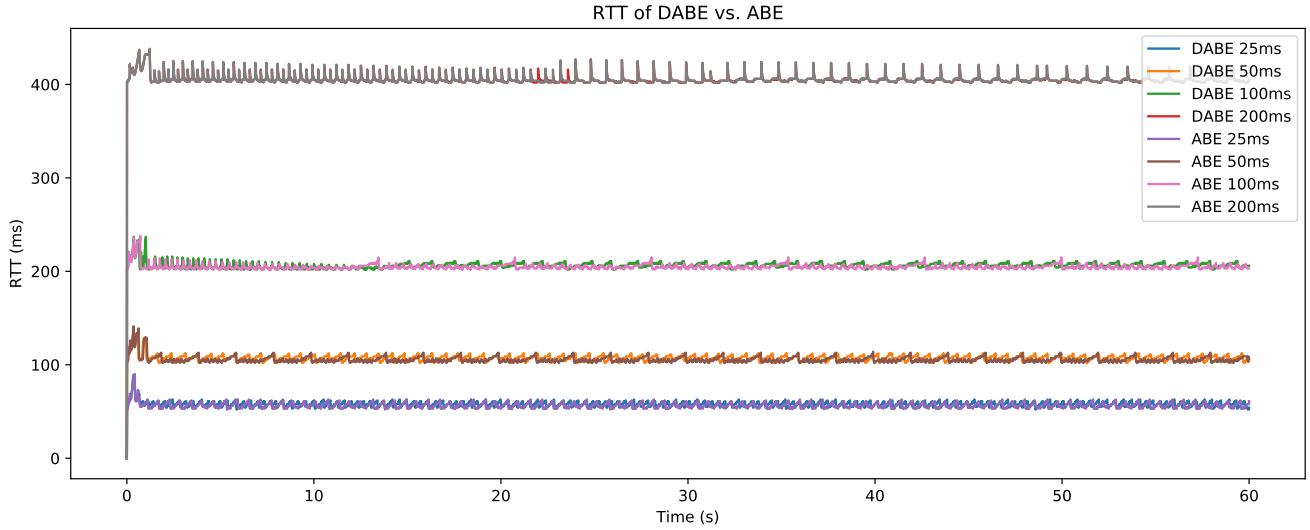


Figure 4.2: The RTT value for DABE vs. ABE taken from separate experiments under various delays.

In order to assess the overall CWND and RTT behavior for DABE and ABE, the following Figure 4.3 presents the average CWND and RTT values from the ten runs that has been conducted.

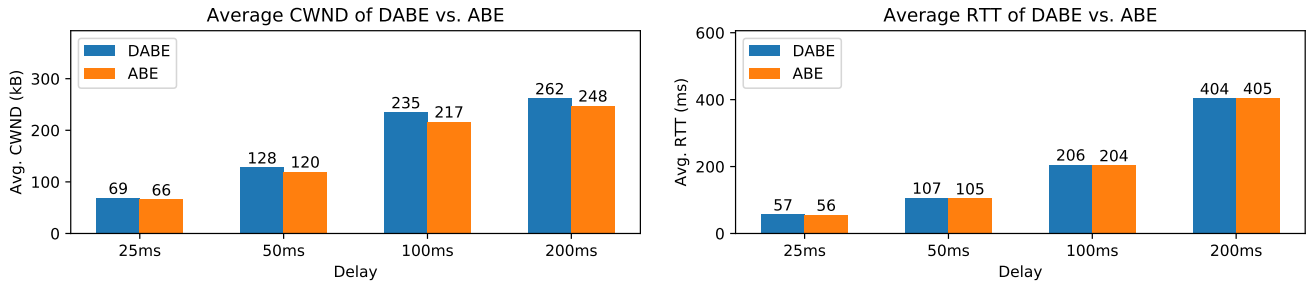


Figure 4.3: The average CWND and RTT values for DABE and ABE. The average is taken from running the experiment ten times.

Figure 4.4 shows the throughput in four different delays for DABE and ABE.

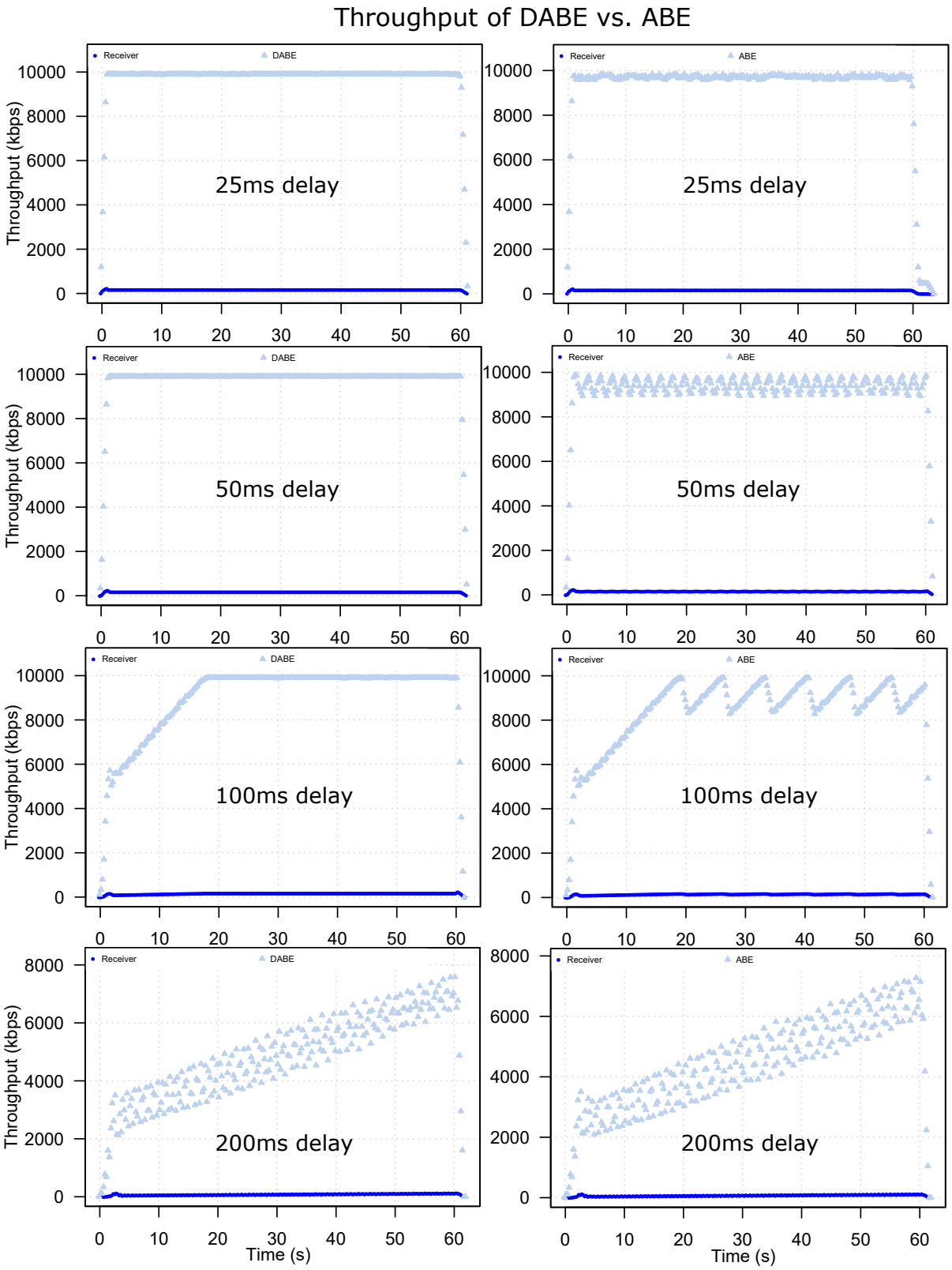


Figure 4.4: The throughput of DABE and ABE under four different delays.

4.3 Intra-Fairness

In this section we will look at intra-fairness of DABE by running experiments where it competes against itself.

4.3.1 Two Flows

In this section we have two flows of DABE sharing the same link in order to see how they coexist with each other. The following Figure 4.5 shows the resulting CWND value over time when it competes against another DABE flow.

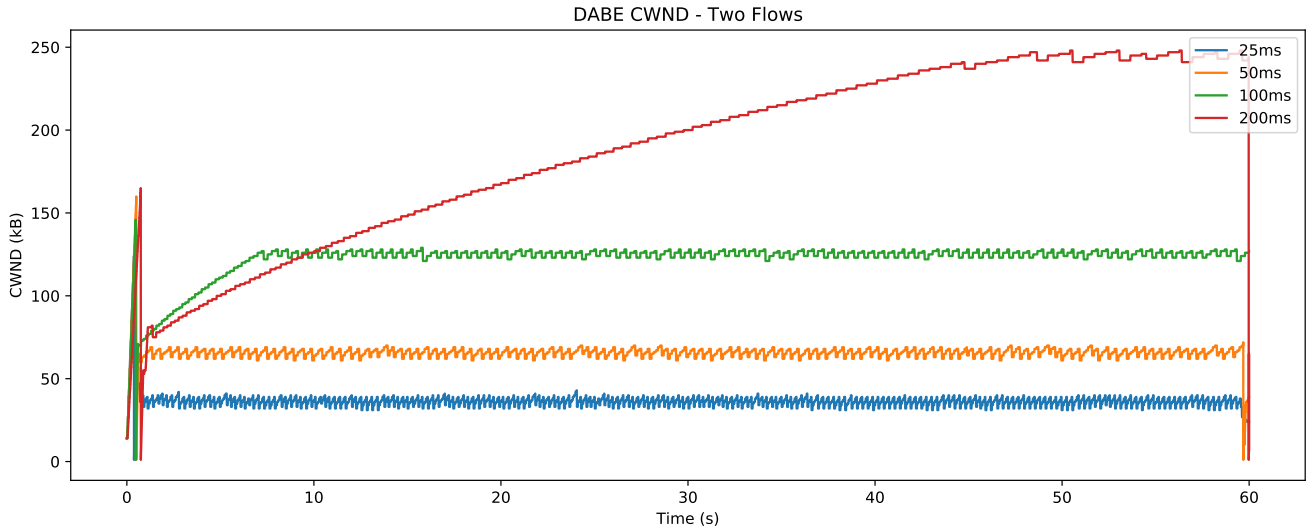


Figure 4.5: The CWND of DABE when competing against another DABE flow.

Figure 4.6 shows the RTT value over time when competing against another DABE flow.

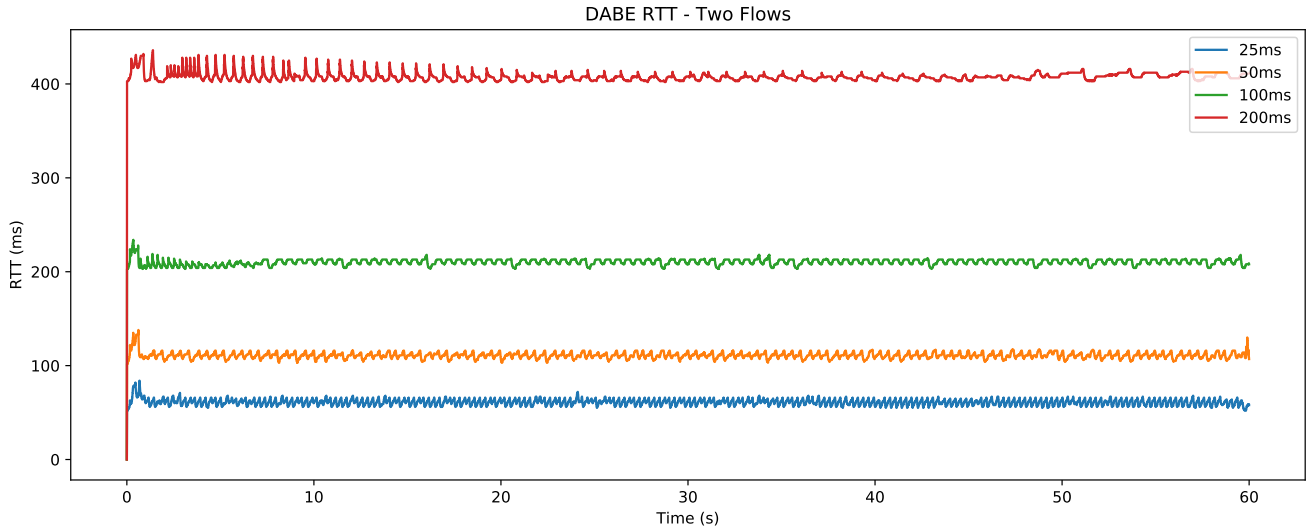


Figure 4.6: The RTT of DABE when competing against another DABE flow.

In order to assess the overall behavior when DABE competes against another DABE flow, the following Figure 4.7 presents the average CWND and RTT values from ten experiments that has been conducted.

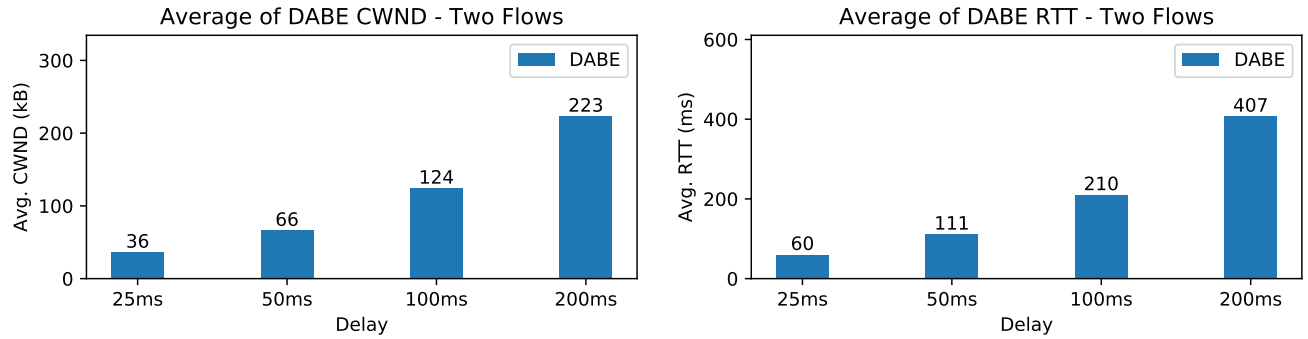


Figure 4.7: The average CWND and RTT when two flows of DABE compete against each others. The average is taken from ten experiments.

Figure 4.8 shows the throughput in four different delays when DABE and ABE coexists.

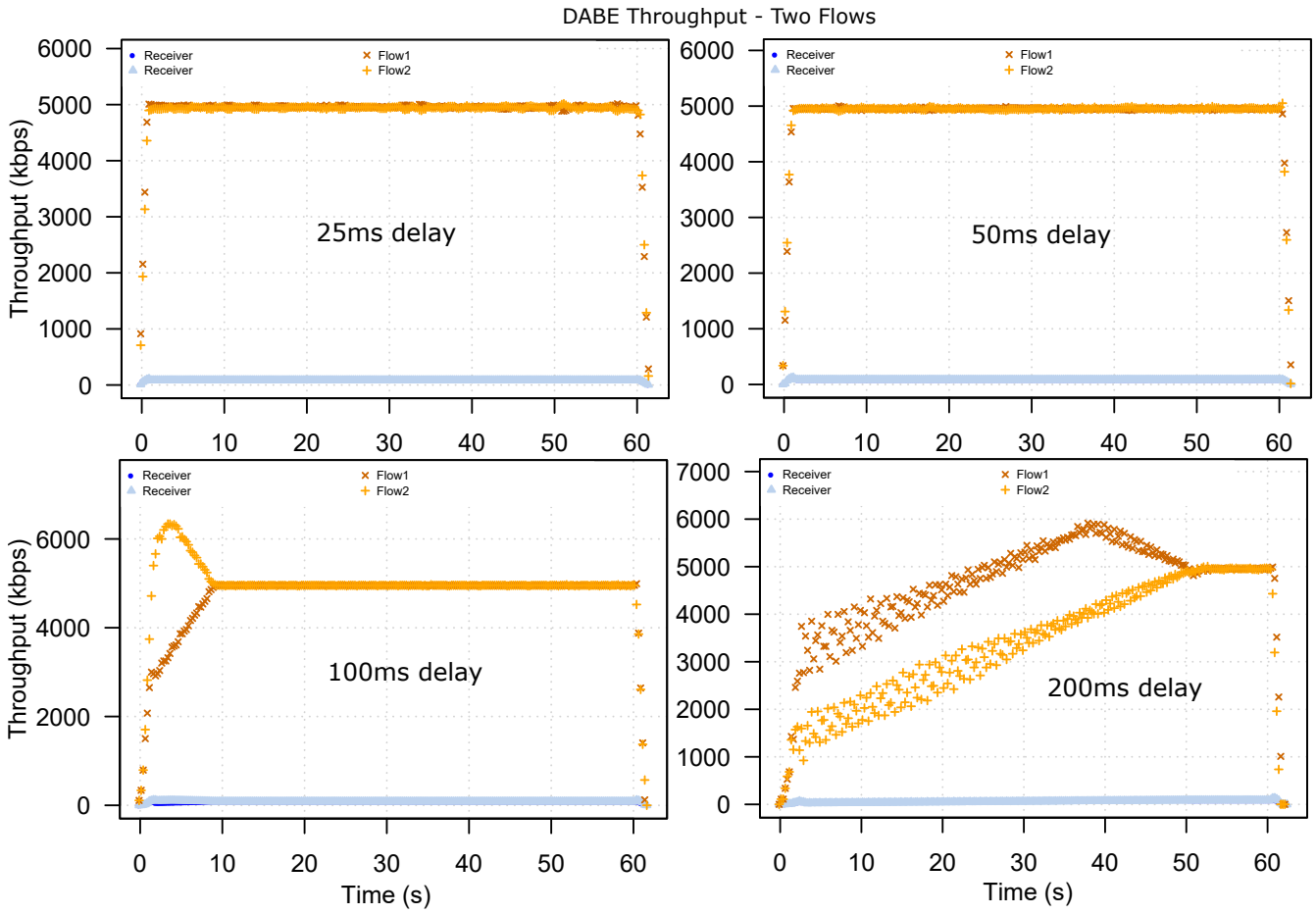


Figure 4.8: The throughput of DABE competing with another DABE flow under four different delays.

4.3.2 Four Flows

In this section we have four flows of DABE sharing the same link in order to see how they coexist with each other. The following Figure 4.9 shows the resulting CWND value over time when it competes against three other DABE flows.

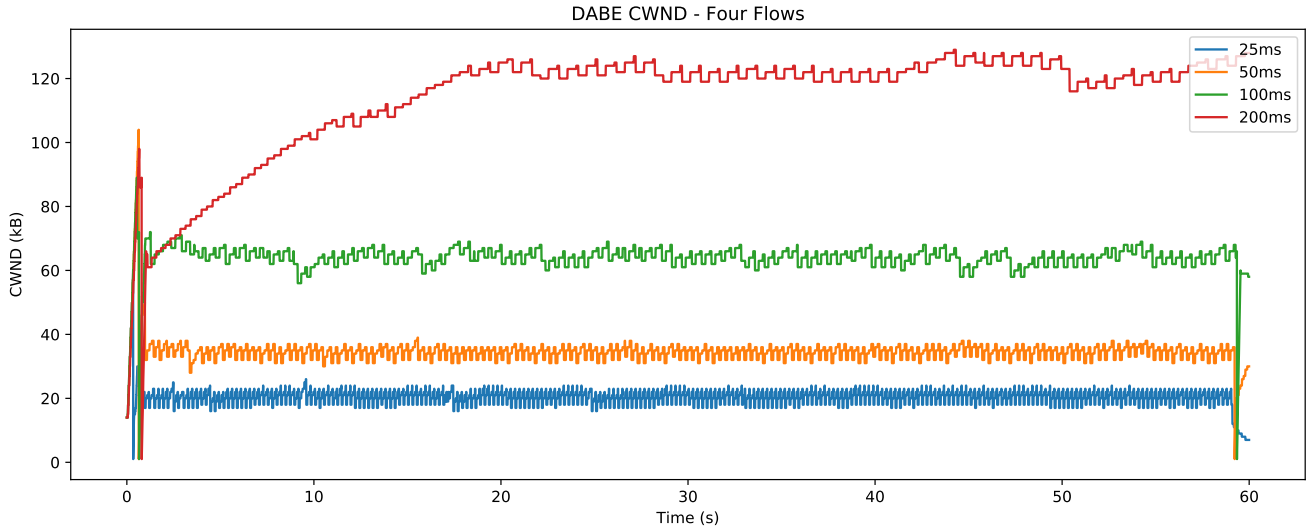


Figure 4.9: The CWND of DABE when competing against three other DABE flows.

Figure 4.10 shows the RTT value over time when competing against three other DABE flows.

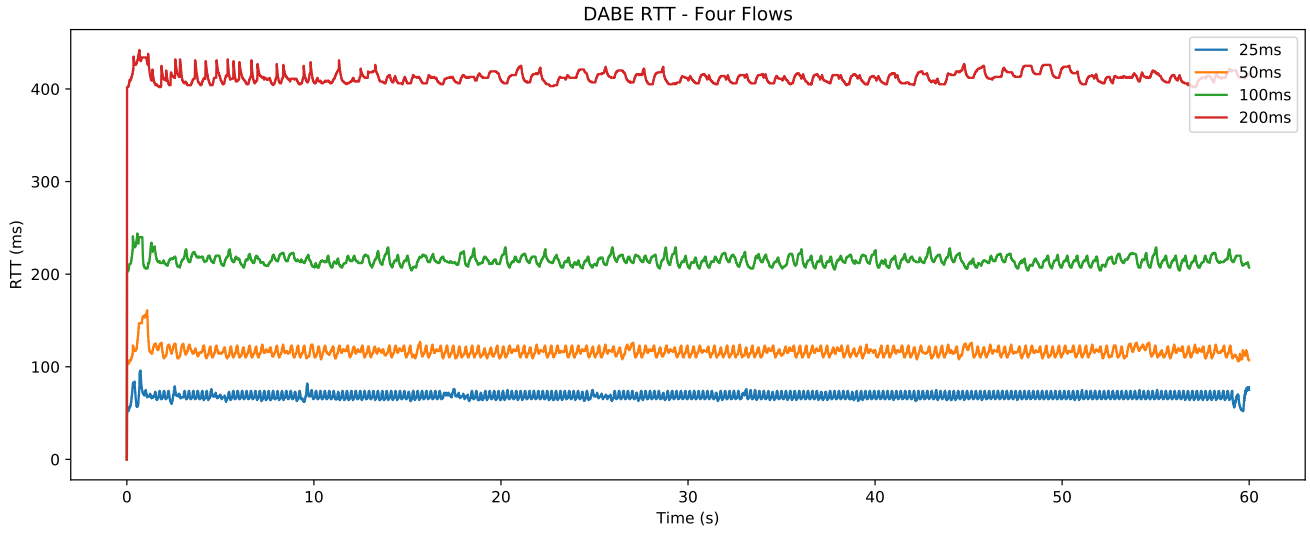


Figure 4.10: The RTT of DABE when competing against three other DABE flows.

In order to assess the overall behavior when DABE competes against three other DABE flows, the following Figure 4.11 presents the average CWND and RTT values from ten experiments that has been conducted.

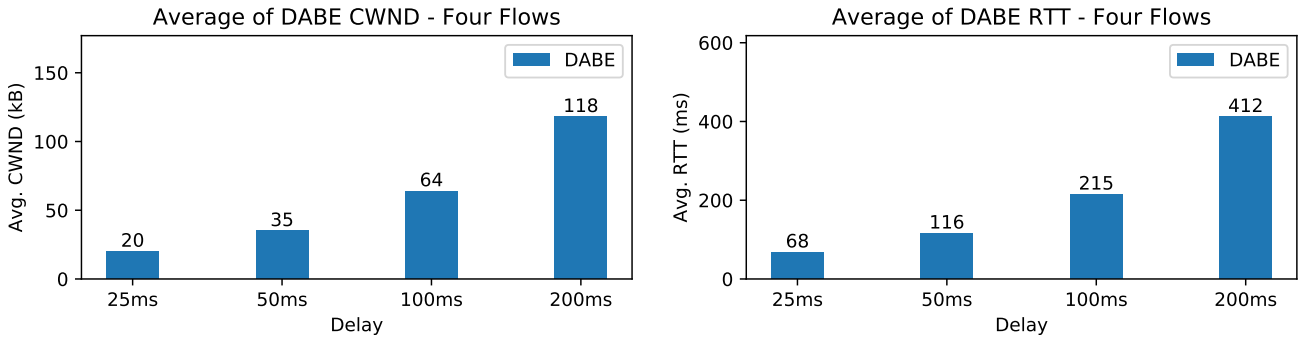


Figure 4.11: The average CWND and RTT when four flows of DABE compete against each others. The average is taken from ten experiments.

Figure 4.12 shows the throughput in four different delays when four DABE flows comepete against each other.

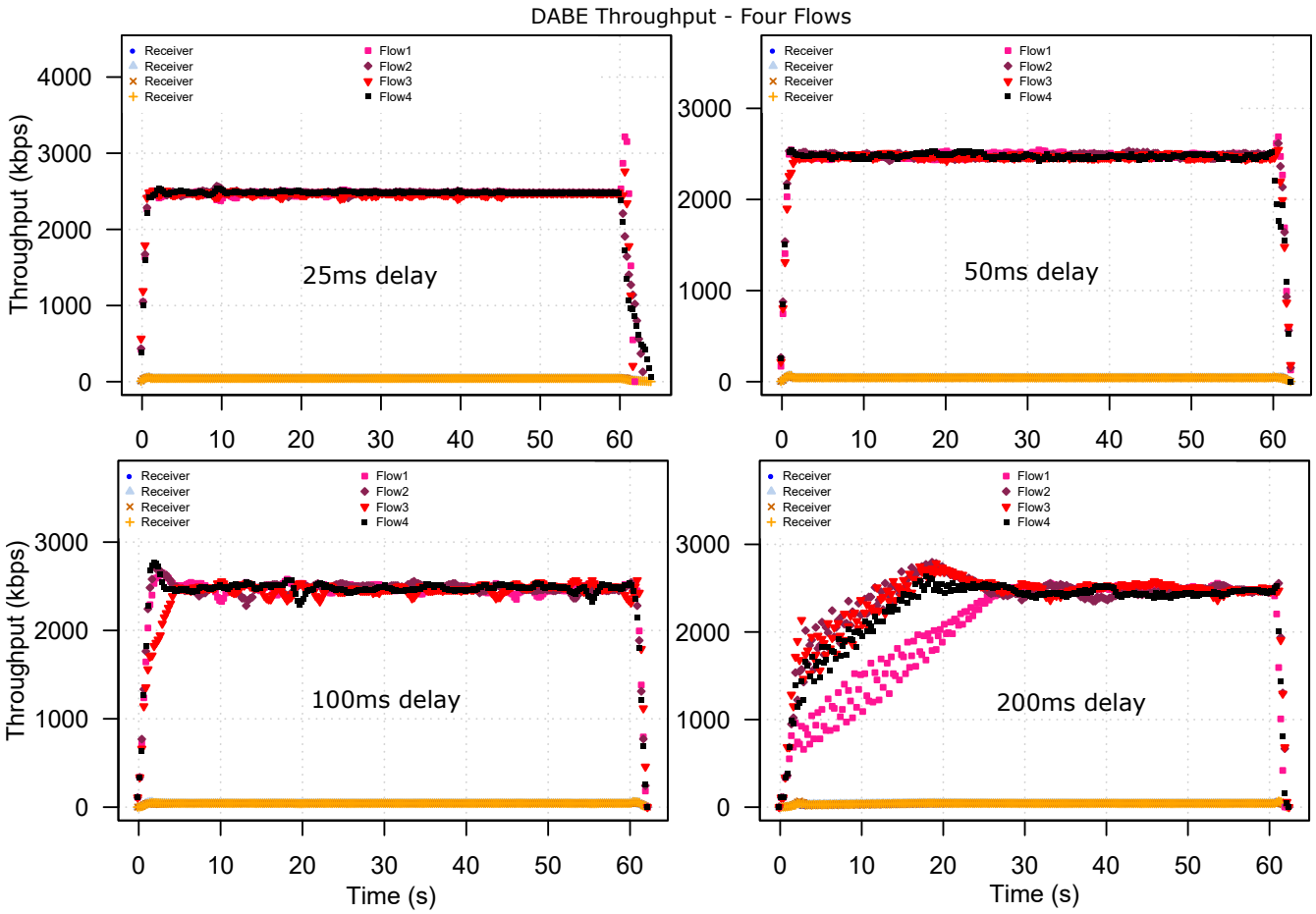


Figure 4.12: The throughput of four competing DABE flows under four different delays.

4.4 Mixed Flows

In this section we will look at how DABE coexists with other flows. The TEACUP configuration file for mixed flows can be found in Appendix B.1.2.

4.4.1 DABE and ABE

In this experiment we have generated one flow using DABE and another using ABE sharing the same link. Each experiment has been run ten times with four different delays. Figure 4.13 shows the CWND value over time with both flows present.

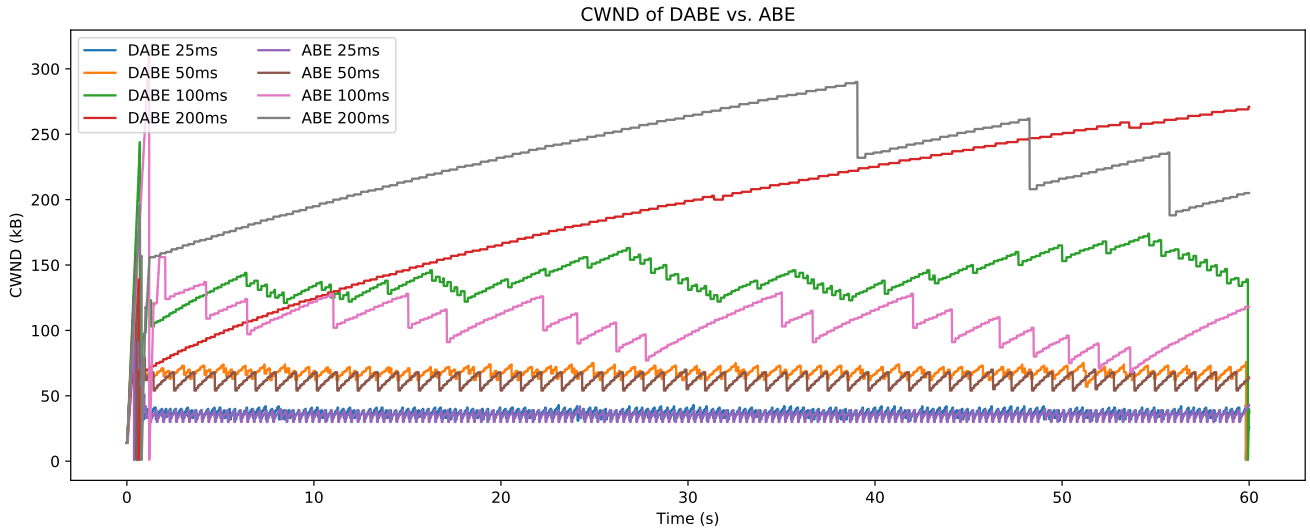


Figure 4.13: The CWND value over time for DABE and ABE when coexisting under four different delays.

Figure 4.14 shows the RTT value over time with both flows present.

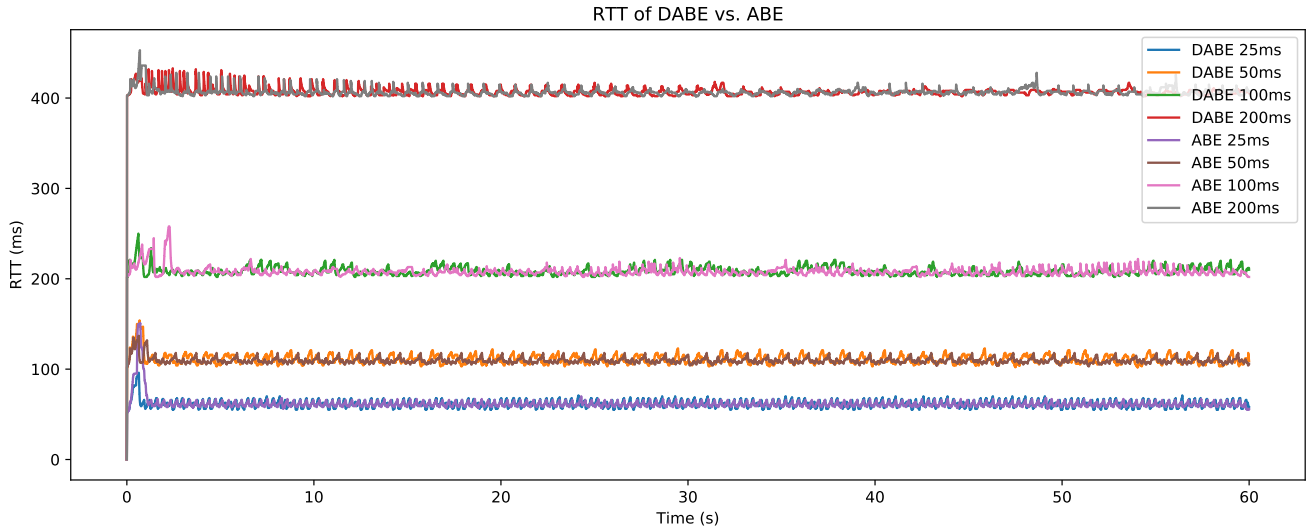


Figure 4.14: The RTT value over time for DABE and ABE when coexisting under four different delays.

In order to assess the overall behavior when DABE and ABE coexists, the following Figure 4.15 presents the average CWND and RTT values from the ten runs that has been conducted.

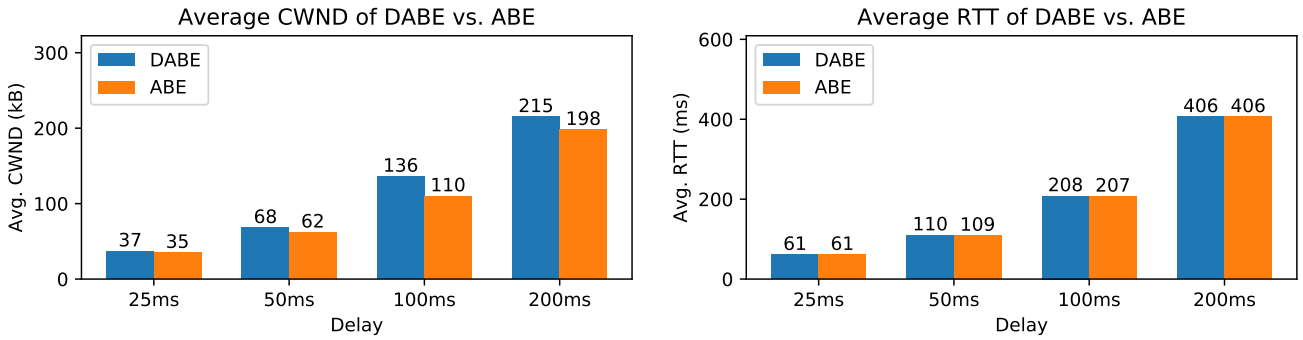


Figure 4.15: The average CWND and RTT values for DABE and ABE when mixed. The average is taken from running the experiment ten times.

Figure 4.16 shows the throughput in four different delays when DABE and ABE coexists.

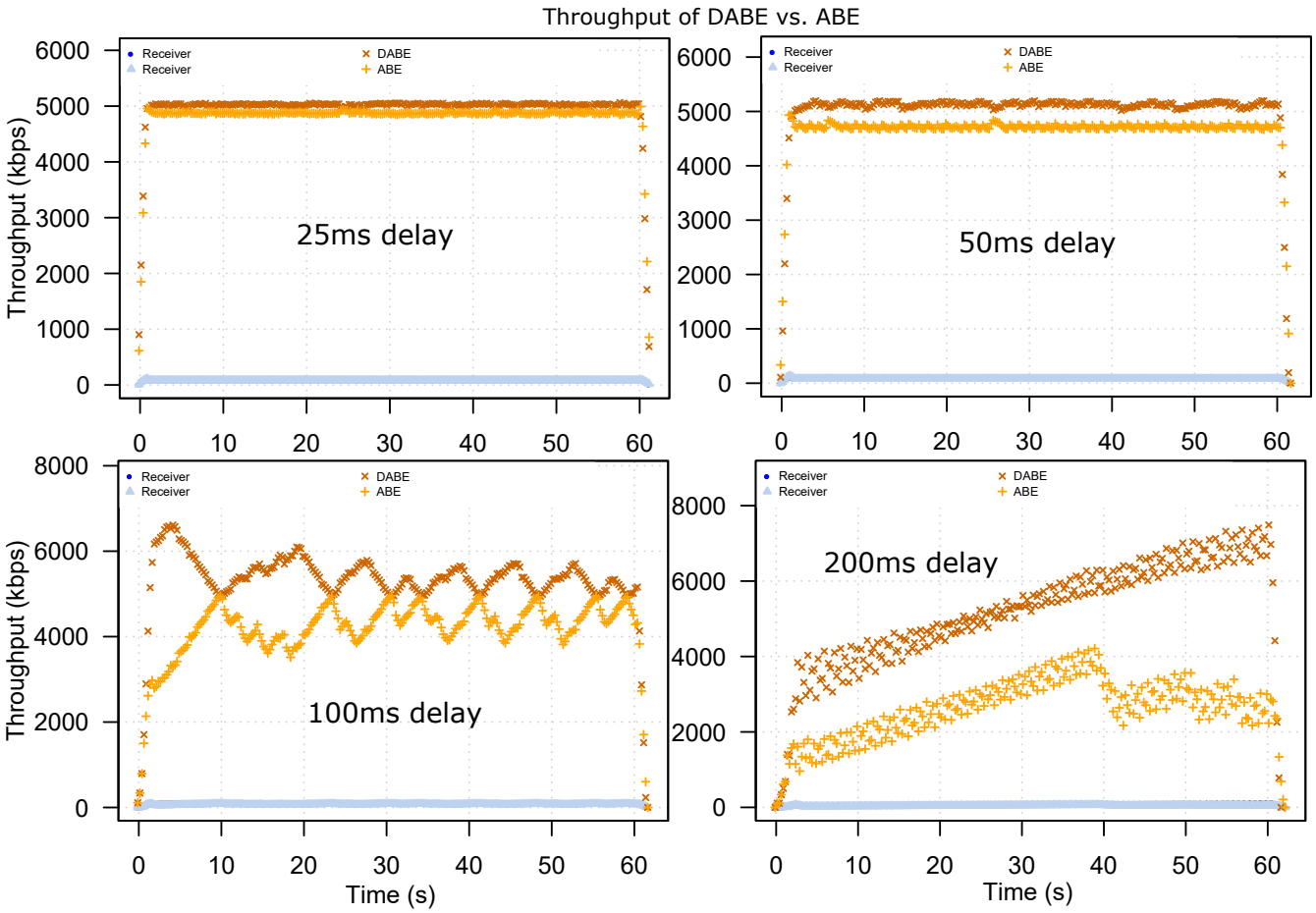


Figure 4.16: The throughput of DABE and ABE under four different delays when coexisting.

4.4.2 DABE and CUBIC

In this experiment we show the behavior of DABE and CUBIC when coexisting. Each experiment has been run ten times with four different delays. Figure 4.17 shows the CWND value over time with both flows present.

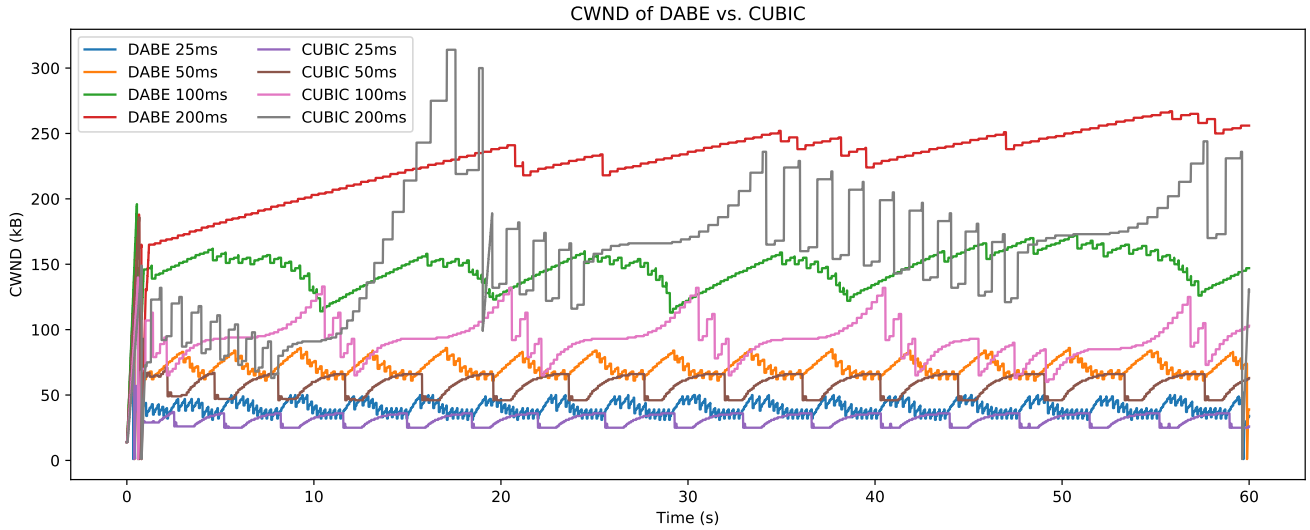


Figure 4.17: The CWND value over time for DABE and CUBIC when coexisting under four different delays.

Figure 4.18 shows the RTT value over time with both flows present.

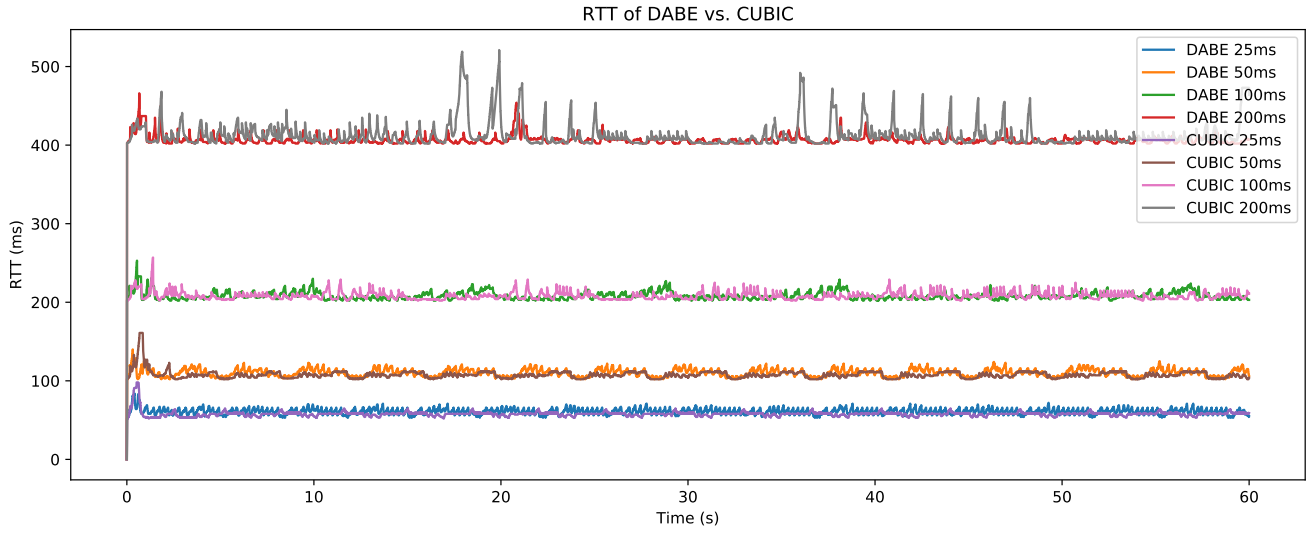


Figure 4.18: The RTT value over time for DABE and CUBIC when coexisting under four different delays.

In order to assess the overall behavior when DABE and CUBIC coexists, the following Figure 4.19 presents the average CWND and RTT values from the ten runs that has been conducted.

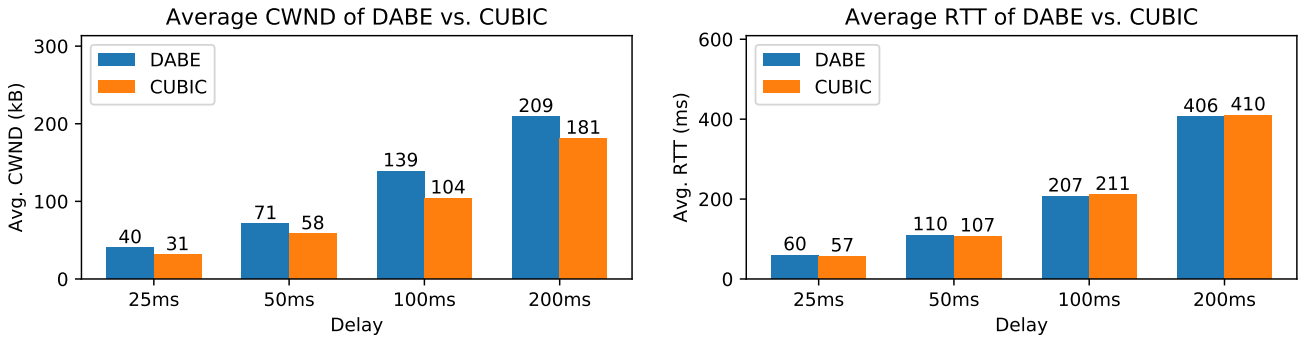


Figure 4.19: The average CWND and RTT values for DABE and CUBIC when mixed. The average is taken from running the experiment ten times.

Figure 4.20 shows the throughput in four different delays when DABE and CUBIC coexists.

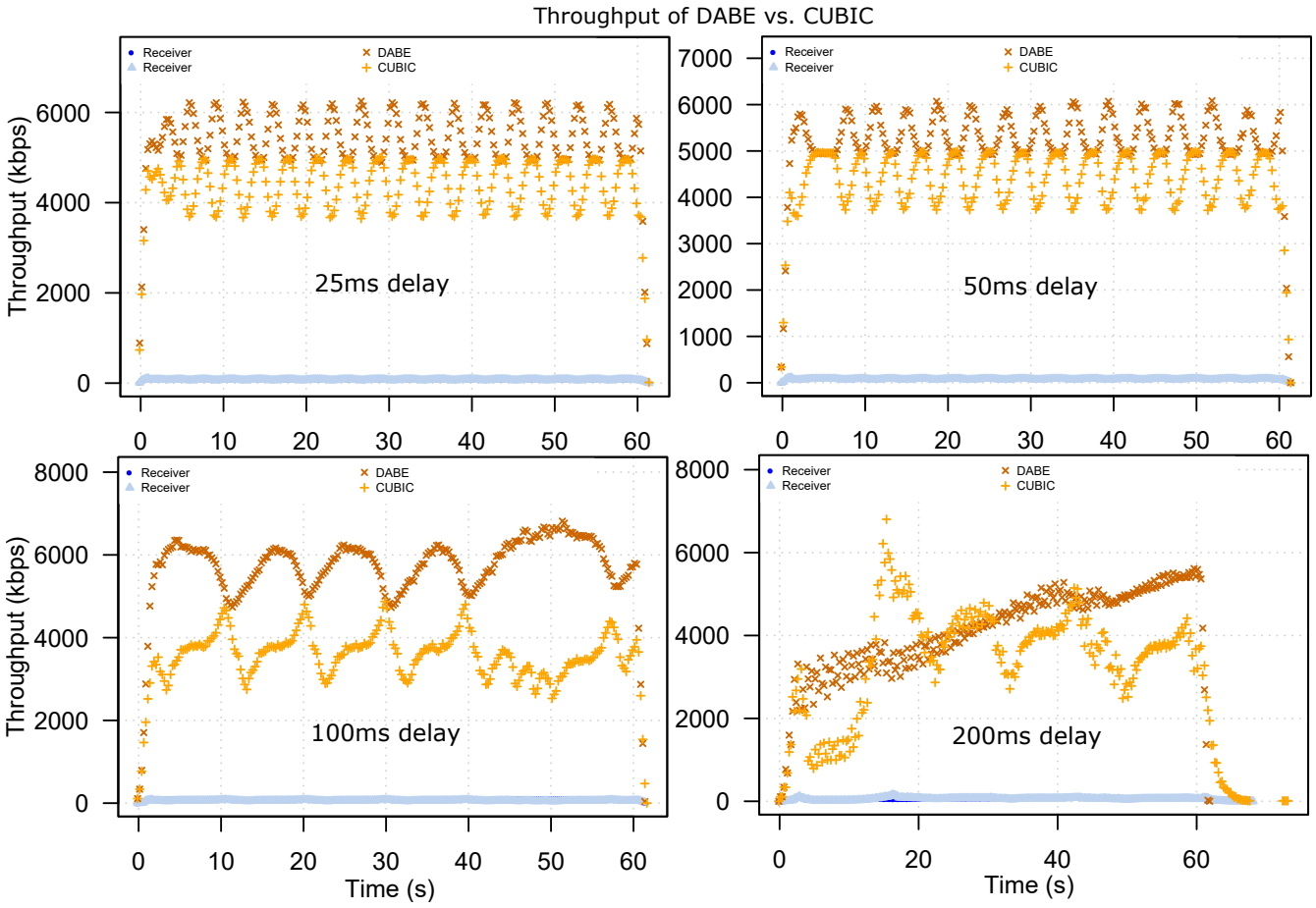


Figure 4.20: The throughput of DABE and CUBIC under four different delays when coexisting.

4.4.3 DABE and NewReno

In this experiment we show the behavior of mixing DABE and NewReno without ECN. That is, NewReno will only use packet loss for congestion signaling. Each experiment has been run ten times with four dif-

ferent delays. Figure 4.21 shows the CWND value over time with both flows present.

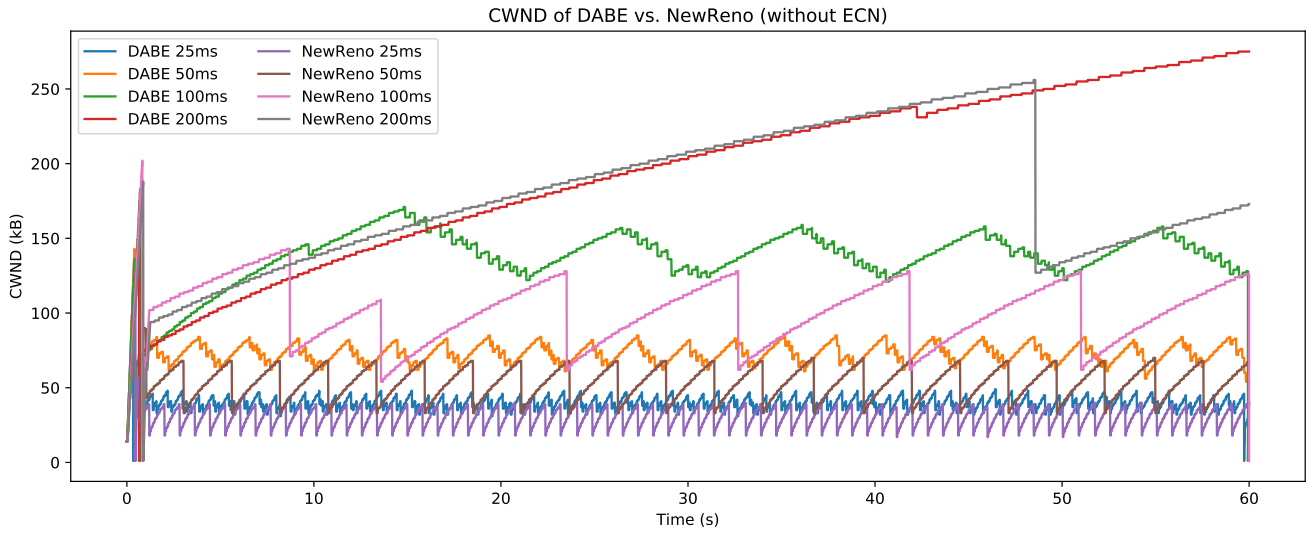


Figure 4.21: The CWND value over time for DABE and NewReno when coexisting under four different delays.

Figure 4.22 shows the RTT value over time with both flows present.

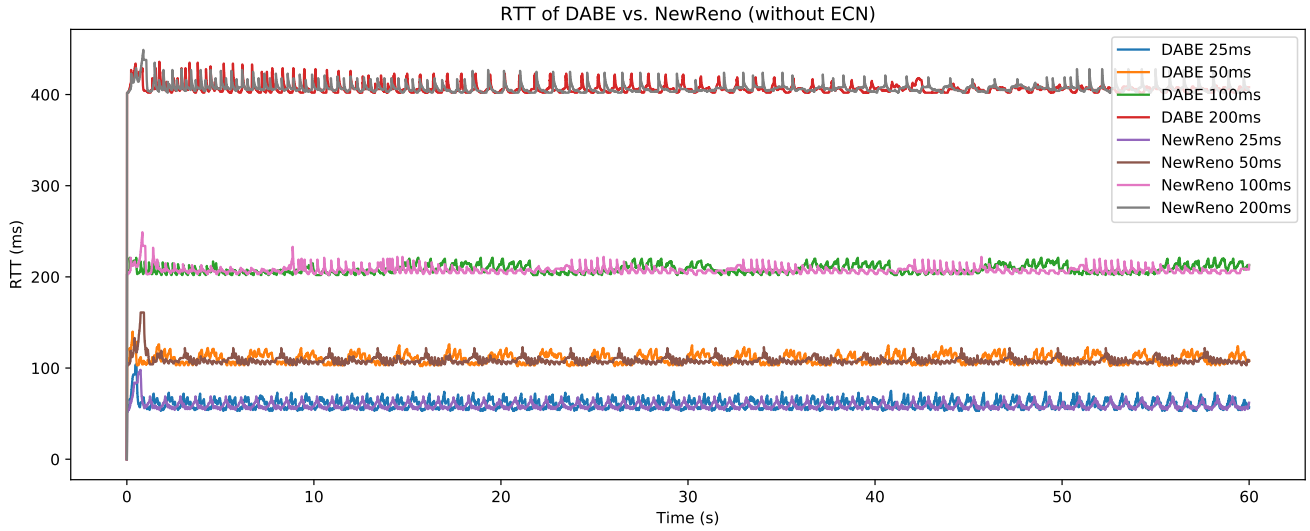


Figure 4.22: The RTT value over time for DABE and NewReno when coexisting under four different delays.

In order to assess the overall behavior when DABE and NewReno coexists, the following Figure 4.23 presents the average CWND and RTT values from the ten runs that has been conducted.

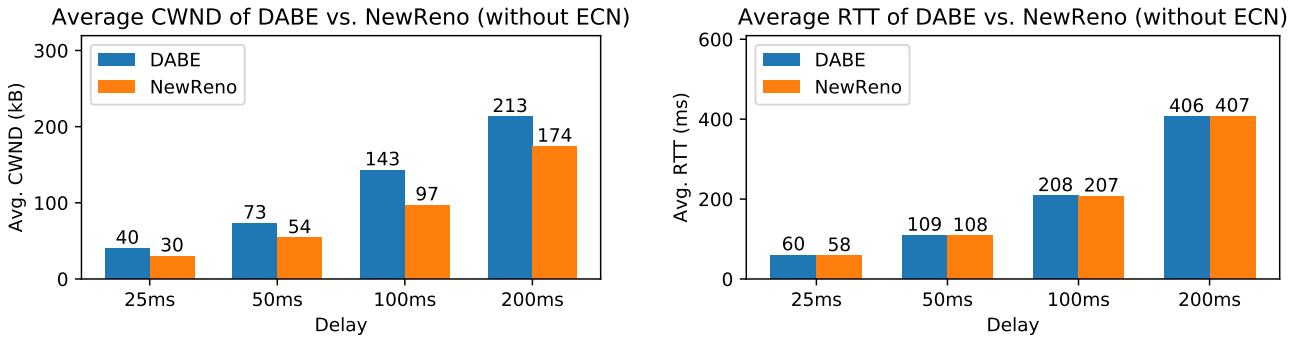


Figure 4.23: The average CWND and RTT values for DABE and NewReno when mixed. The average is taken from running the experiment ten times.

Figure 4.24 shows the throughput in four different delays when DABE and NewReno coexists.

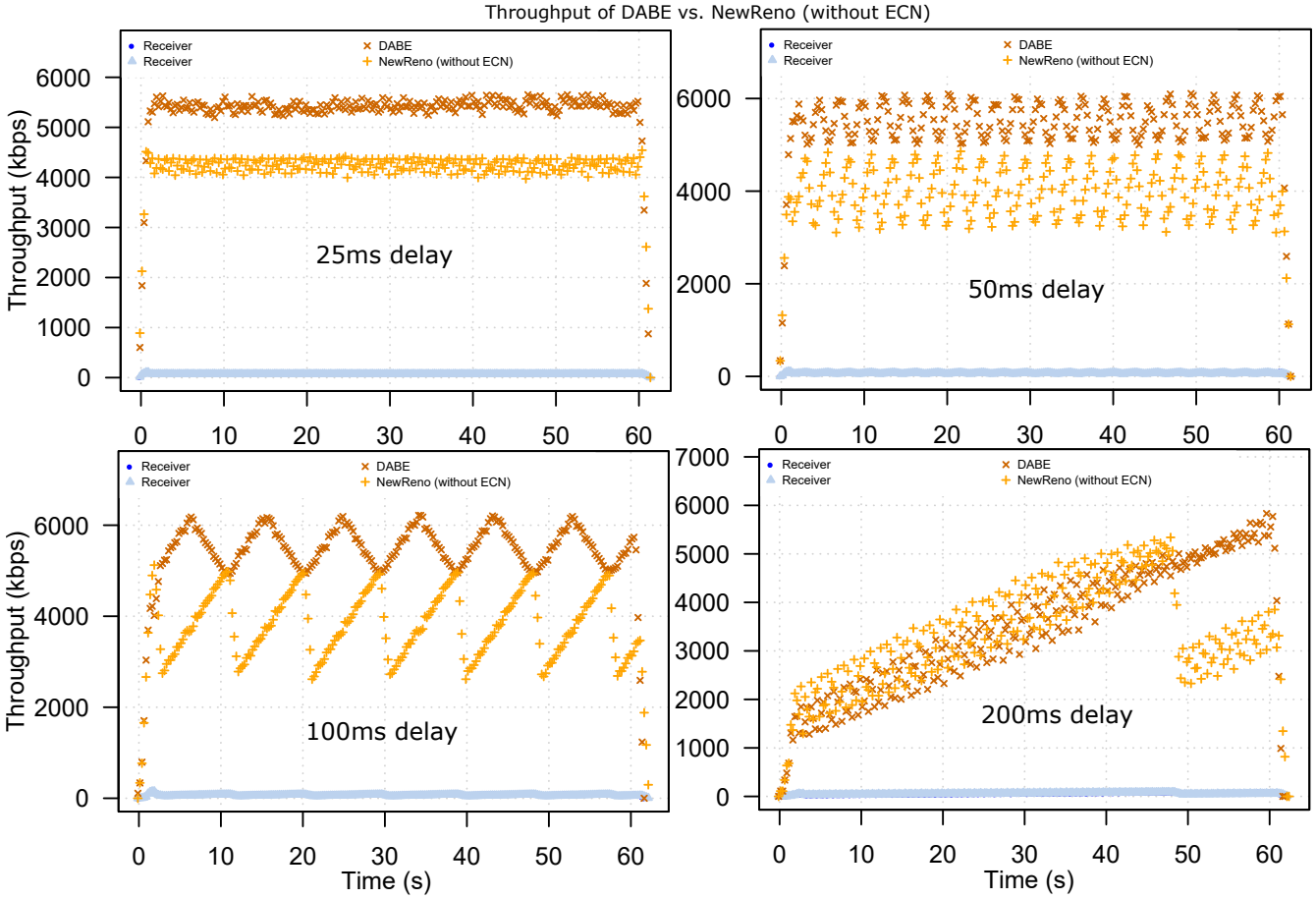


Figure 4.24: The throughput of DABE and NewReno under four different delays when coexisting.

4.4.4 DABE, ABE and CUBIC

In this experiment we will look at the behavior of mixing three flows together, namely DABE, ABE and CUBIC. Each experiment has been run ten times with four different delays. Figure 4.25 shows the CWND

value over time with both flows present.

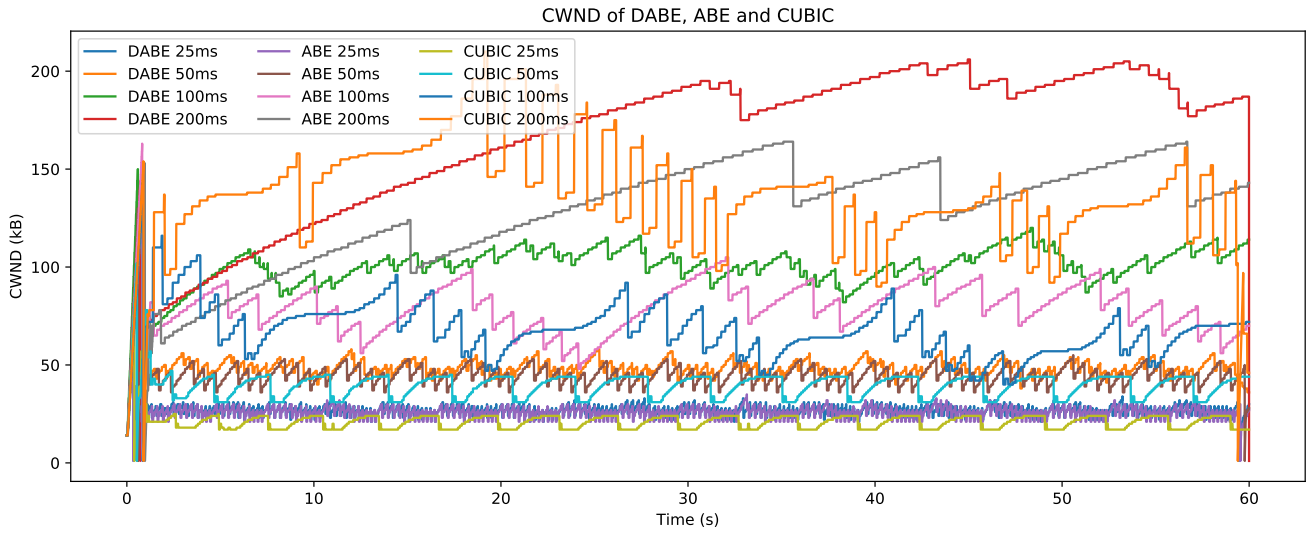


Figure 4.25: The CWND value over time for DABE, ABE and CUBIC when coexisting under four different delays.

Figure 4.26 shows the RTT value over time with both flows present.

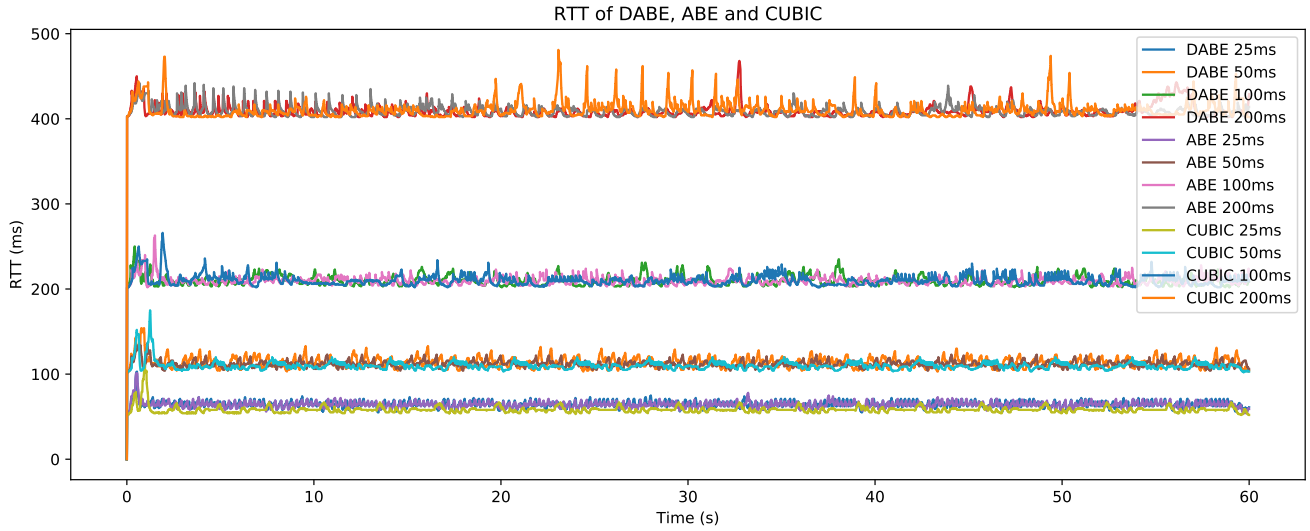


Figure 4.26: The RTT value over time for DABE, ABE and CUBIC when coexisting under four different delays.

In order to assess the overall behavior when DABE, ABE and CUBIC coexists, the following Figure 4.27 presents the average CWND and RTT values from the ten runs that has been conducted.

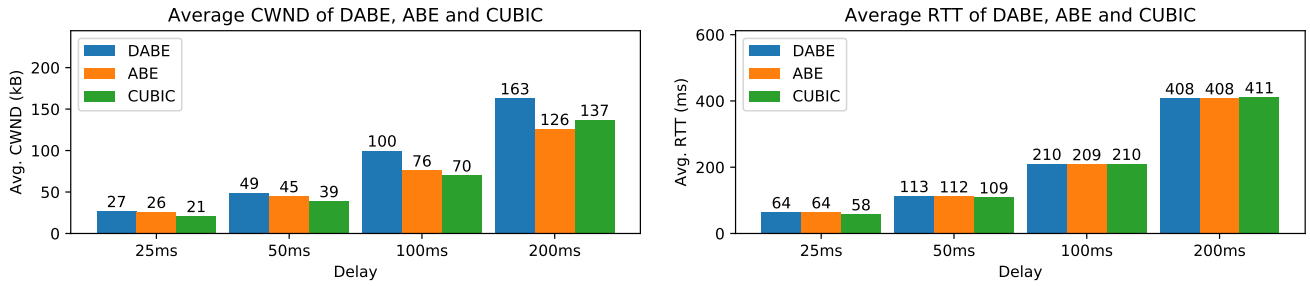


Figure 4.27: The average CWND and RTT values for DABE, ABE and CUBIC when mixed. The average is taken from running the experiment ten times.

Figure 4.28 shows the throughput in four different delays when DABE, ABE and CUBIC coexists.

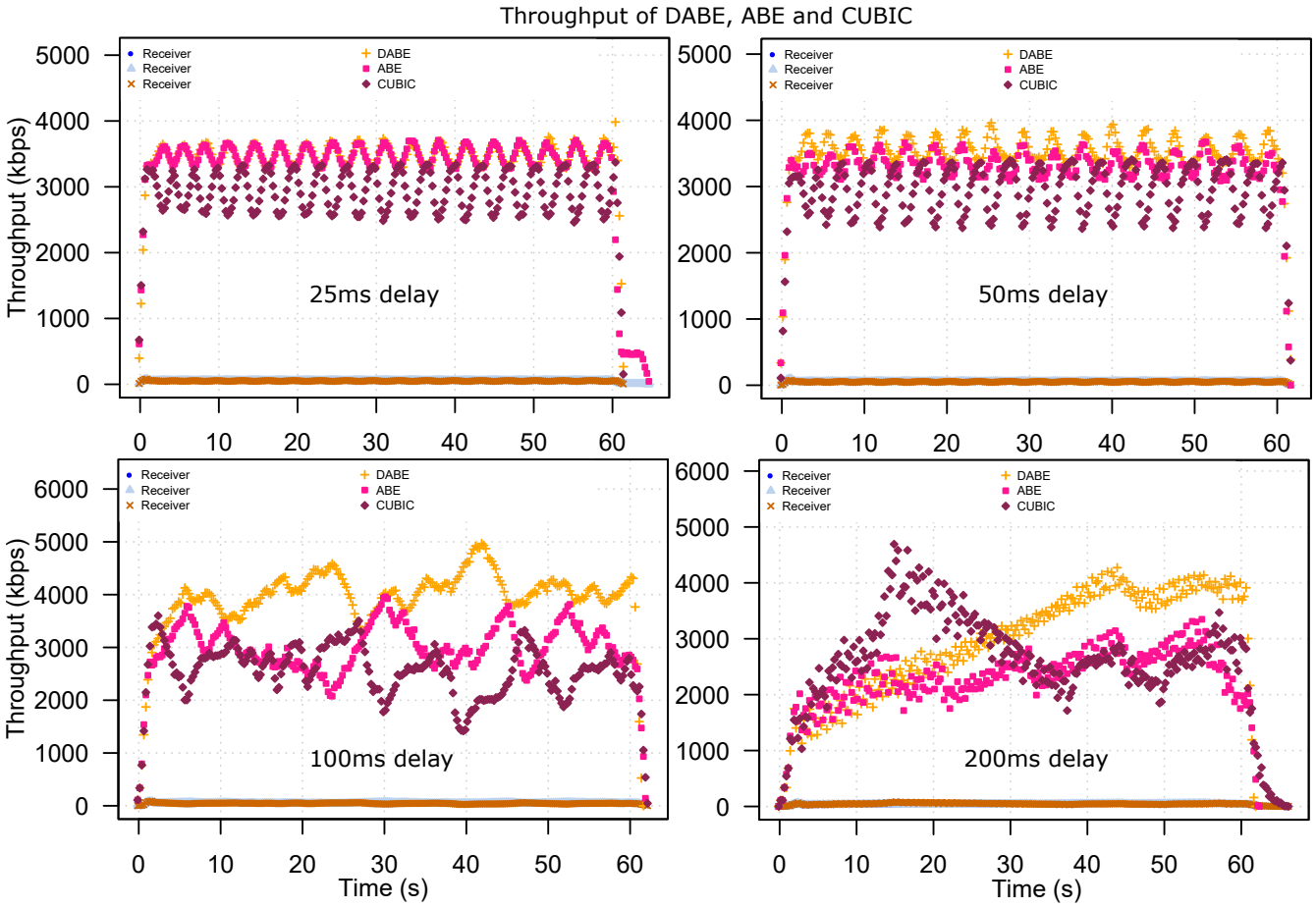


Figure 4.28: The throughput of DABE, ABE and CUBIC under four different delays when coexisting.

4.5 Evaluation

Our main goal was to improve ABE. From Figure 4.4, we can see that DABE successfully reduces its CWND by less than ABE. In addition, the sustained throughput remains consistently high and stable compared to ABE, especially when the latency increases. The exception occurs when RTT reaches 200 ms,

in which it looks like DABE and ABE starts to merge in behavior.

We see this same behavior from Figure 4.1, where the CWND of DABE consistently stays above ABE, until latency reaches 200 ms. The behavior if confirmed by looking at the average of CWND and RTT from Figure 4.3, where DABE shows a slightly higher CWND value during all the various latencies while the average RTT remains roughly the same.

We argue that the reason for this is simply because DABE reduces much more frequently with a significantly lower reduction factor compared to ABE which always reduces by 20%. This effect is especially apparent when the RTT stays stable.

In terms of fairness, we can we can see from Section 4.3 the that when DABE flows compete against each other, they eventually converge in throughput. In other words, DABE achieves intra-fairness. However, when looking at fairness in mixed flows from Section 4.4, we can see that the flows does both diverge and converge in alternate waves when it comes to throughput. We also see that DABE always yields the higher throughput, but this is because DABE reduces less which results in a higher throughput.

Chapter 5

Conclusion

This chapter concludes our thesis on improving ABE. We wrap up with a short conclusion and follow up with a reflection of further improvements. Finally, we end the thesis with a brief note on the setbacks that were experienced in this project.

The goal of the thesis was to improve ABE by making it more responsive to various network conditions. This was achieved by making the ECN congestion response depend on RTT measurements, where the ratio between the minimum and most recent RTT was used as the multiplicative decrease factor. From our limited set of experiments, this yielded a consistently higher throughput without any significant increase in latency when compared with ABE and CUBIC.

5.1 Future Work

There are many improvements that we would have liked to add to DABE. In addition, more experiments and statistical analysis would have aided in determining the usability of DABE. We will briefly discuss them here.

A major concern when depending on RTT measurements is that they are inherently noisy. Delayed ACKs and path changes in the routing of intricate networks can lead to highly inaccurate RTT estimates. In our implementation, we use the minimum RTT measured from the h_ertt module, but this value could be inaccurate. For example, if a path change in the routing occurs such that the latency increases significantly, the resulting backoff might be too much. And if the route change stays, then DABE will perform poorly for the rest of the connection. Another scenario might be if a DABE flow joins an already congested network, causing an overestimated minimum RTT that eventually reduces due to packet loss. We would have liked to investigate this further.

Another concern is the lack of experimental scenarios that have been conducted. We would have liked to run much more experiments in order to produce more statistical data. As it currently stands, there are simply too little data to make more concrete conclusions.

5.2 Setbacks

Initially, we received a cluster of Raspberry Pi 4 Model B machines, which is currently the newest model. We were tasked with dual booting the machines with FreeBSD and a Linux-based OS. After much frustration, we found out that FreeBSD does not support the newest Raspberry Pi 4[1], and so we ended up installing only Linux, namely Raspbian Buster as the OS.

This led to the biggest hurdle in our project. In order for TEACUP to work with Linux hosts, the Web10G kernel was required. Web10G is a set of TCP stack kernel instruments that enables fine grained measurements of the internal actions of the TCP stack. To install it, the existing Linux kernel installed on Raspbian

[1] https://wiki.freebsd.org/action/show/arm/Raspberry%20Pi?action=show&redirect=FreeBSD%2Farm%2FRaspberry+Pi

Buster needed to be patched with the Web10G changes and then compiled, along with installing a kernel module afterwards and some userland tools. We have never used Linux before, so this was quite the undertaking for us.

Installing Web10G was met with heavy resistance. We fought countless compile errors, modified the Linux kernel code in the hopes of fixing it, and applied many suggestions we found on the Internet. Eventually, we managed to successfully compile Web10G and boot up with the custom kernel. Unfortunately, when testing to see if Web10G was working, we got a kernel panic. We found that the issue only occured on the Raspberry Pi, as Web10G was working fine on our laptops. In other words, it was due to the ARM architecture. We tried to fix[2] the kernel panic on Raspberry Pi 4, but never managed to do so.

Thankfully, our supervisor had requested a new cluster of Raspberry Pi 3 models which we eventually received. Although Web10G also experienced kernel panics here, FreeBSD was now working and so we ditched Linux entirely on the hosts and went with only FreeBSD instead. However, new problems kept adding up due to the ARM architecture. For instance, the spp tool TEACUP uses refused to start, but we managed to get everything to sort of work after a while.

By the time everything was "working", we had used up over half of our project timeline, with two months remaining for the deadline. We could now finally start working on our actual bachelor project, which is to improve ABE. And this is where we hit another struggle. We have no experience with socket programming in C, nor any programming in C/C++ for that matter. Additionally, we have no experience with the networking stack of either the Linux of FreeBSD kernel. We quickly found that you do not simply jump into kernel development, but nonetheless, we forced the issue as time was running out.

And finally, to add some salt to our wounds after spending so much time on setting up TEACUP, most completed experiments ended up producing incomplete data, even though no errors were reported. This became increasingly frustrating when strapped for time, and especially when the experiments took several hours to run, only to find that the results could not be analysed due to "incomplete" log files. Unfortunately, this led to severe lack of experiments in the final result.

However, in the end, we managed to come up with a simple solution that we are actually proud of to have achieved.

[2] https://github.com/rapier1/web10g/issues/11

Appendix A

The PI3-Cluster Testbed

A.1 Installing an Operating System

To use the Raspberry Pi, an OS first needs to be installed. Raspbian Buster Lite is the official OS for Raspberry Pi 3, and is used on the gateway and router. FreeBSD is used on the hosts.

After downloading the OS, uncompress the file. The extracted file should be of type .img, containing a preinstalled OS that now must be written to the SD card. To write the image file to the SD card, several tools exists that can be used, with some examples following:

- Raspberry Pi Imager (official, cross-platform)
- balenaEtcher (cross-platform)
- dd (Linux)

Once the image has been successfully written, simply put the SD card into the Raspberry Pi machine and boot it. On Raspbian Buster, the default user that comes preinstalled is called pi, with the default password raspberry. On FreeBSD, two users are installed by default; root and freebsd, with the password being the same as the username.

A.2 Keyboard Layout

Raspbian Buster

To change keyboard layout with a graphical user guide, run dpkg-reconfigure keyboard-configuration. The resulting changes is permanently added.

For most full-sized keyboards, the following options are what one may want; Generic 105-key PC (intl.) -> Norwegian -> Norwegian -> The **default for** the keyboard layout -> No compose key.

FreeBSD

To change keyboard layout with a graphical user guide, run kbdmap. To make the changes persist, add the resulting configuration to the /etc/rc.conf file. For example, if Norwegian was selected, run echo 'keymap="no.kbd"' >> /etc/rc.conf.

A.3 Root Account

Raspbian Buster

Once logged in with user pi, create a root user with sudo passwd root. Enter a chosen password. A root user has now been created. To use it, log out with logout or simply reboot, and then log in as root.

To delete the pi user, issue the command deluser --remove-home pi while logged in as root.

FreeBSD

FreeBSD already comes preinstalled with two user accounts; root and freebsd. To delete the freebsd user, log in as root and run rmuser freebsd.

A.4 Updating the System

Raspbian Buster

Note: *If the current system is using a custom kernel, upgrading the system will also upgrade the kernel, thus overwriting the custom kernel. One can omit the upgrade part if that is not desired.*

To update the system, run apt update && apt upgrade. If an error about release file not valid yet appears, the system's clock needs to be fixed. This can easily be done with the date tool; date -s "15 Feb 2020 12:00".

FreeBSD

To update the system, run the following:

```
# Update FreeBSD system
freebsd-update fetch && freebsd-update install

# Update FreeBSD packages
pkg upgrade
```

A.5 Enable SSH

Raspbian Buster

To enable SSH access, simply run systemctl enable ssh. To allow SSH root login, the line PermitRootLogin yes must be added to the file /etc/ssh/sshd_config, which can conveniently be done with echo 'PermitRootLogin yes' >> /etc/ssh/sshd_config. Then start SSH service with systemctl start ssh or just reboot.

FreeBSD

SSH on FreeBSD is installed and enabled by default as seen by sshd_enable="YES" in the file /etc/rc.conf. However, root login must be explicitly allowed; echo 'PermitRootLogin yes' >> /etc/ssh/sshd_config. Reboot to apply the changes.

Note: SSH access on FreeBSD can sometimes experience a long delay when establishing a connection. If this is the case, DNS lookup can be disabled with echo 'UseDNS no' >> /etc/ssh/sshd_config.

A.6 Change Hostname

Raspbian Buster

There are two files that needs to be edited in order to change hostname. First, the /etc/hostname file, which only contains the current hostname. Simply change whatever is in it to what you want, say new_hostname . Second, the file /etc/hosts needs one line changed. Edit the line containg 127.0.1.1 so that it looks as follows:

```
127.0.1.1       new_hostname
```

A reboot will apply the changes.

FreeBSD

Only one file need to be changed. Open up the /etc/rc.conf file, and replace the hostname entry with the desired hostname. Reboot to apply the changes.

A.7 Time Synchronization

First, set up a common timezone on all machines. On Raspbian Buster, run timedatectl set-timezone CET. On FreeBSD, run tzsetup CET.

NTP Server on Raspbian Buster

Install ntp with apt install ntp. Then check if any NTP peers are connected with ntpq -p. If not, consider replacing the default pools or servers in /etc/ntp.conf with another, such as server ntp.uio.no followed up by systemctl restart ntp.

NTP Client on Raspbian Buster

To automatically query for time from the NTP server, the Raspbian Buster OS already comes with a lightweight daemon called systemd-timesyncd that allows for synchronizing the system clock across the network. However, TEACUP prefers to use ntp instead, and so ntp will be used.

Install ntp with apt install ntp. To specify which server to get the time from, prepare to edit the /etc/ntp .conf file. Comment out the default pools, and add the entry server 10.0.1.254 iburst which refers to the gateway.

Next, disable the systemd-timesyncd with timedatectl set-ntp false as we are using ntp directly to update the time. Finally, either restart the service with systemctl restart ntp or reboot. Wait a bit, and verify time synchronization with either ntpq -p or simply date.

NTP Client on FreeBSD

FreeBSD comes preinstalled with ntp. To enable it on boot, run the following:

```
# Enable NTP on FreeBSD
echo 'ntpd_enable=YES' >> /etc/rc.conf
echo 'ntpd_sync_on_start=YES' >> /etc/rc.conf
```

To specify which server to get the time from, prepare to edit the /etc/ntp.conf file. Comment out any default pool or server entries, and add the new entry server 10.0.1.254 iburst to it, which refers to the gateway. Reboot and verify time synchronization with date.

Appendix B

TEACUP

B.1 Configurations

B.1.1 Performance

The TEACUP configuration file for the completed experiments related to performance in Section 4.2 follows.

CONFIGURATION FILE FOR PERFORMANCE EXPERIMENTS

```
import sys
import datetime
from fabric.api import env

#
# Fabric config
#

# User and password
env.user = 'root'
env.password = 'root'

# Set shell used to execute commands
env.shell = '/bin/sh -c'

# SSH connection timeout
env.timeout = 5

# Number of concurrent processes
env.pool_size = 2


#
# Testbed config
#

# Path to scripts
TPCONF_script_path = '/home/danny/teacup/teacup-1.1'
# DO NOT remove the following line
sys.path.append(TPCONF_script_path)

# Set debugging level (0 = no debugging info output)
TPCONF_debug_level = 0
```

```
# Host lists
TPCONF_router = ['pi3router', ]
TPCONF_hosts = [ 'pi3host2', 'pi3host7', ]

# Map external IPs to internal IPs
TPCONF_host_internal_ip = {
    'pi3router': ['172.16.10.1', '172.16.20.1'],
    'pi3host2':  ['172.16.10.2'],
    'pi3host7':  ['172.16.20.7'],
}

#
# Experiment settings
#

# Maximum allowed time difference between machines in seconds
# otherwise experiment will abort cause synchronisation problems
TPCONF_max_time_diff = 2

# Experiment name prefix used if not set on the command line
# The command line setting will overrule this config setting
now = datetime.datetime.today()
# old default test ID prefix (version < 1.0)
#TPCONF_test_id = now.strftime("%Y%m%d-%H%M%S") + '_experiment'
# new default test ID prefix
TPCONF_test_id = 'exp_' + now.strftime("%Y%m%d-%H%M%S")

# Directory to store log files on remote host
TPCONF_remote_dir = '/root/tmp/'

# Time offset measurement options
# Enable broadcast ping on external/control interfaces
TPCONF_bc_ping_enable = '1'
# Specify rate of pings in packets/second
TPCONF_bc_ping_rate = 1
# Specify multicast address to use (must be broadcast or multicast address)
# If this is not specified, by default the ping will be send to the subnet
# broadcast address.
#TPCONF_bc_ping_address = '224.0.1.199'

# List of router queues/pipes

# Each entry is a tuple. The first value is the queue number and the second value
# is a comma separated list of parameters (see routersetup.py:init_pipe()).
# Queue numbers must be unique.

# Note that variable parameters must be either constants or or variable names
# defined by the experimenter. Variables are evaluated during runtime. Variable
# names must start with a 'V_'. Parameter names can only contain numbes, letter
# (upper and lower case), underscores (_), and hypen/minus (-).

# All variables must be defined in TPCONF_variable_list (see below).
```

```
# Note parameters must be configured appropriately for the router OS, e.g. there
# is no CoDel on FreeBSD; otherwise the experiment will abort witn an error.

TPCONF_router_queues = [
    # Set same delay for every host
    ('1', " source='172.16.10.0/24', dest='172.16.20.0/24', delay=V_delay, "
        " loss=V_loss, rate=V_up_rate, queue_disc=V_aqm, queue_size=V_bsize, "
        "queue_disc_params=V_aqm_params "),
    ('2', " source='172.16.20.0/24', dest='172.16.10.0/24', delay=V_delay, "
        " loss=V_loss, rate=V_down_rate, queue_disc=V_aqm, queue_size=V_bsize, "
        "queue_disc_params=V_aqm_params "),
]

# List of traffic generators

traffic_iperf = [
    # Specifying external addresses traffic will be created using the _first_
    # internal addresses (according to TPCONF_host_internal_ip)
    ('0.0', '1', " start_iperf, client='pi3host2', server='pi3host7', port=5000, "
        " duration=V_duration "),
]

# THIS is the traffic generator setup we will use
TPCONF_traffic_gens = traffic_iperf

# Parameter ranges

# Duration in seconds
TPCONF_duration = 60

# Number of runs for each setting
TPCONF_runs = 10

# If '1' enable ecn for all hosts, if '0' disable ecn for all hosts
TPCONF_ECN = ['1']

# TCP congestion control algorithm used
# Possible algos are: default, host<N>, newreno, cubic, cdg, hd, htcp, compound, vegas
# Note that the algo support is OS specific, so must ensure the right OS is booted
# Windows: newreno (default), compound
# FreeBSD: newreno (default), cubic, hd, htcp, cdg, vegas
# Linux: newreno, cubic (default), htcp, vegas
# If you specify 'default' the default algorithm depending on the OS will be used
# If you specify 'host<N>' where <N> is an integer starting from 0 to then the
# algorithm will be the N-th algorithm specified for the host in TPCONF_host_TCP_algos
# (in case <N> is larger then the number of algorithms specified, it is set to 0
TPCONF_TCP_algos = [ 'host0', 'host1', 'host2', ]

# Specify TCP congestion control algorithms used on each host
TPCONF_host_TCP_algos = {
    'pi3host2': [ 'abe', 'newreno', 'cubic' ],
    'pi3host7': [ 'newreno', 'newreno', 'newreno' ],
}
```

```
# Specify TCP parameters for each host and each TCP congestion control algorithm
# Each parameter is of the form <sysctl name> = <value> where <value> can be a constant
# or a V_ variable
TPCONF_host_TCP_algo_params = {}

# Specify arbitray commands that are executed on a host at the end of the host
# intialisation (after general host setup, ecn and tcp setup). The commands are
# executed in the shell as written after any V_ variables have been replaced.
# LIMITATION: only one V_ variable per command
TPCONF_host_init_custom_cmds = {
    'pi3host2' : [ 'sysctl net.inet.tcp.cc.abe=1' ],
    'pi3host7' : [ 'sysctl net.inet.tcp.cc.abe=1' ],
}

# Delays in ms
TPCONF_delays = [25, 50, 100, 200]

# Loss rates in percent
TPCONF_loss_rates = [0]

# Bandwidth (downstream, upstream)
# Note: Linux syntax
TPCONF_bandwidths = [
    ('10mbit', '10mbit'),
]

# AQM
# Note this is router OS specific
# Linux: fifo (mapped to pfifo), pfifo, bfifo, fq_codel, codel, pie, red, ...
#        (see tc man page for full list)
# FreeBSD: fifo, red
TPCONF_aqms = ['fq_codel', ]

# AQM parameters
# example for PIE: 'ecn target 20ms tupdate 30ms'
# check manual for more: man tc-pie, tc-fq_codel...
TPCONF_aqms_params = ['ecn', ]

# Buffer size
# If router is Linux this is mostly in packets/slots, but it depends on AQM
# (e.g. for bfifo it's bytes)
# If router is FreeBSD this would be in slots by default, but we can specify byte sizes
# (e.g. we can specify 4Kbytes)
TPCONF_buffer_sizes = [64]

# List of all parameters that can be varied

# The key of each item is the identifier that can be used in TPCONF_vary_parameters
# (see below).
# The value of each item is a 4-tuple. First, a list of variable names.
# Second, a list of short names uses for the file names.
# For each parameter varied a string '_<short_name>_<value>' is appended to the log
```

```
# file names (appended to chosen prefix). Note, short names should only be letters
# from a-z or A-Z. Do not use underscores or hyphens!
# Third, the list of parameters values. If there is more than one variable this must
# be a list of tuples, each tuple having the same number of items as teh number of
# variables. Fourth, an optional dictionary with additional variables, where the keys
# are the variable names and the values are the variable values.

TPCONF_parameter_list = {
#   Vary name        V_ variable                file name        values      extra vars
    'ecns'          :  (['V_ecn'],          ['ecn'],        TPCONF_ECN, {}),
    'delays'        :  (['V_delay'],        ['del'],        TPCONF_delays, {}),
    'loss'          :  (['V_loss'],         ['loss'],       TPCONF_loss_rates, {}),
    'tcpalgos'      :  (['V_tcp_cc_algo'],  ['tcp'],        TPCONF_TCP_algos, {}),
    'aqms'          :  (['V_aqm'],          ['aqm'],        TPCONF_aqms, {}),
    'aqms_params'   :  (['V_aqm_params'],   ['aqm_params'], TPCONF_aqms_params, {}),
    'bsizes'        :  (['V_bsize'],        ['bs'],         TPCONF_buffer_sizes, {}),
    'runs'          :  (['V_runs'],         ['run'],        range(TPCONF_runs), {}),
    'bandwidths'    :  (['V_down_rate', 'V_up_rate'], ['down', 'up'], TPCONF_bandwidths, {}),
}

# Default setting for variables (used for variables if not varied)

# The key of each item is the parameter  name. The value of each item is the default
# parameter value used if the variable is not varied.

TPCONF_variable_defaults = {
#   V_ variable                value
    'V_ecn'             :   TPCONF_ECN[0],
    'V_duration'        :   TPCONF_duration,
    'V_delay'           :   TPCONF_delays[0],
    'V_loss'            :   TPCONF_loss_rates[0],
    'V_tcp_cc_algo'     :   TPCONF_TCP_algos[0],
    'V_down_rate'       :   TPCONF_bandwidths[0][0],
    'V_up_rate'         :   TPCONF_bandwidths[0][1],
    'V_aqm'             :   TPCONF_aqms[0],
    'V_aqm_params'      :   TPCONF_aqms_params[0],
    'V_bsize'           :   TPCONF_buffer_sizes[0],
    'V_test'            :   'foobar',
}

# Specify the parameters we vary through all values, all others will be fixed
# according to TPCONF_variable_defaults
TPCONF_vary_parameters = ['tcpalgos', 'delays', 'loss', 'bandwidths',
                          'aqms', 'aqms_params', 'bsizes', 'runs', ]
```

Listing B.1: The configuration file for TEACUP in order to conduct experiments related to the performance of DABE.

B.1.2 Mixed Flows

The TEACUP configuration file for the completed experiments related to mixed flows in Section 4.4 follows.

```
import sys
import datetime
from fabric.api import env

#
# Fabric config
#

# User and password
env.user = 'root'
env.password = 'root'

# Set shell used to execute commands
env.shell = '/bin/sh -c'

# SSH connection timeout
env.timeout = 5

# Number of concurrent processes
env.pool_size = 2


#
# Testbed config
#

# Path to scripts
TPCONF_script_path = '/home/danny/teacup/teacup-1.1'
# DO NOT remove the following line
sys.path.append(TPCONF_script_path)

# Set debugging level (0 = no debugging info output)
TPCONF_debug_level = 0

# Host lists
TPCONF_router = ['pi3router', ]
TPCONF_hosts = [ 'pi3host2', 'pi3host3', 'pi3host4', 'pi3host7', ]

# Map external IPs to internal IPs
TPCONF_host_internal_ip = {
    'pi3router': ['172.16.10.1', '172.16.20.1'],
    'pi3host2':  ['172.16.10.2'],
    'pi3host3':  ['172.16.10.3'],
    'pi3host4':  ['172.16.10.4'],
    'pi3host7':  ['172.16.20.7'],
}

#
# Experiment settings
#
```

```
# Maximum allowed time difference between machines in seconds
# otherwise experiment will abort cause synchronisation problems
TPCONF_max_time_diff = 2

# Experiment name prefix used if not set on the command line
# The command line setting will overrule this config setting
now = datetime.datetime.today()
# old default test ID prefix (version < 1.0)
#TPCONF_test_id = now.strftime("%Y%m%d-%H%M%S") + '_experiment'
# new default test ID prefix
TPCONF_test_id = 'exp_' + now.strftime("%Y%m%d-%H%M%S")

# Directory to store log files on remote host
TPCONF_remote_dir = '/root/tmp/'

# Time offset measurement options
# Enable broadcast ping on external/control interfaces
TPCONF_bc_ping_enable = '1'
# Specify rate of pings in packets/second
TPCONF_bc_ping_rate = 1
# Specify multicast address to use (must be broadcast or multicast address)
# If this is not specified, by default the ping will be send to the subnet
# broadcast address.
#TPCONF_bc_ping_address = '224.0.1.199'

# List of router queues/pipes

# Each entry is a tuple. The first value is the queue number and the second value
# is a comma separated list of parameters (see routersetup.py:init_pipe()).
# Queue numbers must be unique.

# Note that variable parameters must be either constants or or variable names
# defined by the experimenter. Variables are evaluated during runtime. Variable
# names must start with a 'V_'. Parameter names can only contain numbes, letter
# (upper and lower case), underscores (_), and hypen/minus (-).

# All variables must be defined in TPCONF_variable_list (see below).

# Note parameters must be configured appropriately for the router OS, e.g. there
# is no CoDel on FreeBSD; otherwise the experiment will abort witn an error.

TPCONF_router_queues = [
    # Set same delay for every host
    ('1', " source='172.16.10.0/24', dest='172.16.20.0/24', delay=V_delay, "
        " loss=V_loss, rate=V_up_rate, queue_disc=V_aqm, queue_size=V_bsize, "
        "queue_disc_params=V_aqm_params "),
    ('2', " source='172.16.20.0/24', dest='172.16.10.0/24', delay=V_delay, "
        " loss=V_loss, rate=V_down_rate, queue_disc=V_aqm, queue_size=V_bsize, "
        "queue_disc_params=V_aqm_params "),
]

# List of traffic generators
```

```
traffic_iperf = [
    # Specifying external addresses traffic will be created using the _first_
    # internal addresses (according to TPCONF_host_internal_ip)
    ('0.0', '1', " start_iperf, client='pi3host2', server='pi3host7', port=5000, "
        " duration=V_duration "),
    ('0.0', '2', " start_iperf, client='pi3host3', server='pi3host7', port=5000, "
    " duration=V_duration "),
    ('0.0', '3', " start_iperf, client='pi3host4', server='pi3host7', port=5000, "
    " duration=V_duration "),
]

# THIS is the traffic generator setup we will use
TPCONF_traffic_gens = traffic_iperf

# Parameter ranges

# Duration in seconds
TPCONF_duration = 60

# Number of runs for each setting
TPCONF_runs = 10

# If '1' enable ecn for all hosts, if '0' disable ecn for all hosts
TPCONF_ECN = ['1']

# TCP congestion control algorithm used
# Possible algos are: default, host<N>, newreno, cubic, cdg, hd, htcp, compound, vegas
# Note that the algo support is OS specific, so must ensure the right OS is booted
# Windows: newreno (default), compound
# FreeBSD: newreno (default), cubic, hd, htcp, cdg, vegas
# Linux: newreno, cubic (default), htcp, vegas
# If you specify 'default' the default algorithm depending on the OS will be used
# If you specify 'host<N>' where <N> is an integer starting from 0 to then the
# algorithm will be the N-th algorithm specified for the host in TPCONF_host_TCP_algos
# (in case <N> is larger then the number of algorithms specified, it is set to 0
TPCONF_TCP_algos = [ 'host0', ]

# Specify TCP congestion control algorithms used on each host
TPCONF_host_TCP_algos = {
    'pi3host2': [ 'dabe', ],
    'pi3host3': [ 'newreno', ],
    'pi3host4': [ 'cubic', ],
    'pi3host7': [ 'newreno', ],
}

# Specify TCP parameters for each host and each TCP congestion control algorithm
# Each parameter is of the form <sysctl name> = <value> where <value> can be a constant
# or a V_ variable
TPCONF_host_TCP_algo_params = {}

# Specify arbitray commands that are executed on a host at the end of the host
# intialisation (after general host setup, ecn and tcp setup). The commands are
# executed in the shell as written after any V_ variables have been replaced.
```

```
# LIMITATION: only one V_ variable per command
TPCONF_host_init_custom_cmds = {
    'pi3host2' : [ 'sysctl net.inet.tcp.cc.abe=1' ],
    'pi3host3' : [ 'sysctl net.inet.tcp.cc.abe=1' ],
    'pi3host4' : [ 'sysctl net.inet.tcp.cc.abe=0' ],
    'pi3host7' : [ 'sysctl net.inet.tcp.cc.abe=0' ],
}

# Delays in ms
TPCONF_delays = [25, 50, 100, 200]

# Loss rates in percent
TPCONF_loss_rates = [0]

# Bandwidth (downstream, upstream)
# Note: Linux syntax
TPCONF_bandwidths = [
    ('10mbit', '10mbit'),
]

# AQM
# Note this is router OS specific
# Linux: fifo (mapped to pfifo), pfifo, bfifo, fq_codel, codel, pie, red, ...
#        (see tc man page for full list)
# FreeBSD: fifo, red
TPCONF_aqms = ['fq_codel', ]

# AQM parameters
# example for PIE: 'ecn target 20ms tupdate 30ms'
# check manual for more: man tc-pie, tc-fq_codel...
TPCONF_aqms_params = ['ecn', ]

# Buffer size
# If router is Linux this is mostly in packets/slots, but it depends on AQM
# (e.g. for bfifo it's bytes)
# If router is FreeBSD this would be in slots by default, but we can specify byte sizes
# (e.g. we can specify 4Kbytes)
TPCONF_buffer_sizes = [64]

# List of all parameters that can be varied

# The key of each item is the identifier that can be used in TPCONF_vary_parameters
# (see below).
# The value of each item is a 4-tuple. First, a list of variable names.
# Second, a list of short names uses for the file names.
# For each parameter varied a string '_<short_name>_<value>' is appended to the log
# file names (appended to chosen prefix). Note, short names should only be letters
# from a-z or A-Z. Do not use underscores or hyphens!
# Third, the list of parameters values. If there is more than one variable this must
# be a list of tuples, each tuple having the same number of items as teh number of
# variables. Fourth, an optional dictionary with additional variables, where the keys
# are the variable names and the values are the variable values.
```

```
TPCONF_parameter_list = {
#   Vary name        V_ variable                file name        values      extra vars
    'ecns'          :  (['V_ecn'],          ['ecn'],        TPCONF_ECN, {}),
    'delays'        :  (['V_delay'],        ['del'],        TPCONF_delays, {}),
    'loss'          :  (['V_loss'],         ['loss'],       TPCONF_loss_rates, {}),
    'tcpalgos'      :  (['V_tcp_cc_algo'],  ['tcp'],        TPCONF_TCP_algos, {}),
    'aqms'          :  (['V_aqm'],          ['aqm'],        TPCONF_aqms, {}),
    'aqms_params'   :  (['V_aqm_params'],   ['aqm_params'], TPCONF_aqms_params, {}),
    'bsizes'        :  (['V_bsize'],        ['bs'],         TPCONF_buffer_sizes, {}),
    'runs'          :  (['V_runs'],         ['run'],        range(TPCONF_runs), {}),
    'bandwidths'    :  (['V_down_rate', 'V_up_rate'], ['down', 'up'], TPCONF_bandwidths, {}),
}

# Default setting for variables (used for variables if not varied)

# The key of each item is the parameter  name. The value of each item is the default
# parameter value used if the variable is not varied.

TPCONF_variable_defaults = {
#   V_ variable                value
    'V_ecn'             :   TPCONF_ECN[0],
    'V_duration'        :   TPCONF_duration,
    'V_delay'           :   TPCONF_delays[0],
    'V_loss'            :   TPCONF_loss_rates[0],
    'V_tcp_cc_algo'     :   TPCONF_TCP_algos[0],
    'V_down_rate'       :   TPCONF_bandwidths[0][0],
    'V_up_rate'         :   TPCONF_bandwidths[0][1],
    'V_aqm'             :   TPCONF_aqms[0],
    'V_aqm_params'      :   TPCONF_aqms_params[0],
    'V_bsize'           :   TPCONF_buffer_sizes[0],
    'V_test'            :   'foobar',
}

# Specify the parameters we vary through all values, all others will be fixed
# according to TPCONF_variable_defaults
TPCONF_vary_parameters = ['tcpalgos', 'delays', 'loss', 'bandwidths',
                          'aqms', 'aqms_params', 'bsizes', 'runs', ]
```

Listing B.2: The configuration file for TEACUP in order to conduct experiments related to the mixed flows with DABE.

B.2 Utility Bash Scripts

Initiating TEACUP from remote x86 machine

Since the tool spp did not work on Raspberry Pi due to the ARM architecture (see TEACUP section in 3.2.1), the experiment data needs to copied over to an x86 machine in order to analyze the results. This process quickly gets tedious when conducting many experiments. The following script automates the process. One only needs to make sure that the experiment folder from which the script is run has the necessary fabfile.py included, and that the config.py file reflects the path to where teacup-1.1 is located.

INIT_TEACUP.SH

```
#!/bin/sh

################################################################
## A simple bash script for initiating a TEACUP experiment
## on a remote x86 machine in the same network.
## Make sure that fabfile.py is present in both locations.
## NOTE: Must run inside experiment folder where config.py is
################################################################

# Gateway address can be IP address or hostname
GATEWAY=192.168.10.100
USER=root

# Target is the location of the TEACUP experiment folder on the gateway
# Make sure that this location is reflected in config.py
TARGET=/home/danny/teacup/experiment
OUTPUT=result

############################################
## Start TEACUP experiment
############################################

# Delete old results
ssh $USER@$GATEWAY rm $TARGET/experiments_started.txt
ssh $USER@$GATEWAY rm $TARGET/experiments_completed.txt
ssh $USER@$GATEWAY rm -rf $TARGET/$OUTPUT

# Copy config.py to gateway
scp config.py $USER@$GATEWAY:$TARGET

# Start TEACUP experiment
ssh $USER@$GATEWAY fab -f $TARGET/fabfile.py run_experiment_multiple:test_id=$OUTPUT

# Move new results to experiment folder
ssh $USER@$GATEWAY mv ./experiments_started.txt $TARGET
ssh $USER@$GATEWAY mv ./experiments_completed.txt $TARGET
ssh $USER@$GATEWAY mv ./$OUTPUT $TARGET

############################################
## Analyse TEACUP results
############################################

# Cleanup local
rm experiments_completed.txt experiments_started.txt
rm teacup_dir_cache.txt teacup_flow_cache.txt
rm -rf $OUTPUT

# Get new TEACUP results
scp -r $USER@$GATEWAY:$TARGET/* ./

# Analyse results
```

```
fab analyse_all
```

Listing B.3: A utility bash script for initiating TEACUP from a remote host, copying the results over and then analyzing them.

Appendix C

FreeBSD CC Modules

C.1 NewReno

This section includes the source code for the NewReno congestion control module in FreeBSD 12.1. The header file cc_newreno.h C.1 and its implementation file cc_newreno.c C.2 follows.

CC_NEWRENO.H

```
/*-
* Copyright (c) 2017 Tom Jones <tj@enoti.me>
* All rights reserved.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the above copyright
*    notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in the
*    documentation and/or other materials provided with the distribution.
*
* THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*
* $FreeBSD: releng/12.1/sys/netinet/cc/cc_newreno.h 331214 2018-03-19 16:37:47Z lstewart $
*/

#ifndef _CC_NEWRENO_H
#define _CC_NEWRENO_H

#define CCALGONAME_NEWRENO "newreno"

struct cc_newreno_opts {
    int         name;
    uint32_t    val;
```

```
};

#define CC_NEWRENO_BETA         1
#define CC_NEWRENO_BETA_ECN     2

#endif /* _CC_NEWRENO_H */
```

Listing C.1: The header file for NewReno in FreeBSD.

CC_NEWRENO.C

```
/*-
* SPDX-License-Identifier: BSD-2-Clause-FreeBSD
*
* Copyright (c) 1982, 1986, 1988, 1990, 1993, 1994, 1995
*    The Regents of the University of California.
* Copyright (c) 2007-2008,2010,2014
*    Swinburne University of Technology, Melbourne, Australia.
* Copyright (c) 2009-2010 Lawrence Stewart <lstewart@freebsd.org>
* Copyright (c) 2010 The FreeBSD Foundation
* All rights reserved.
*
* This software was developed at the Centre for Advanced Internet
* Architectures, Swinburne University of Technology, by Lawrence Stewart, James
* Healy and David Hayes, made possible in part by a grant from the Cisco
* University Research Program Fund at Community Foundation Silicon Valley.
*
* Portions of this software were developed at the Centre for Advanced
* Internet Architectures, Swinburne University of Technology, Melbourne,
* Australia by David Hayes under sponsorship from the FreeBSD Foundation.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the above copyright
*    notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in the
*    documentation and/or other materials provided with the distribution.
*
* THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
```

```
*/

/*
* This software was first released in 2007 by James Healy and Lawrence Stewart
* whilst working on the NewTCP research project at Swinburne University of
* Technology's Centre for Advanced Internet Architectures, Melbourne,
* Australia, which was made possible in part by a grant from the Cisco
* University Research Program Fund at Community Foundation Silicon Valley.
* More details are available at:
*   http://caia.swin.edu.au/urp/newtcp/
*
* Dec 2014 garmitage@swin.edu.au
* Borrowed code fragments from cc_cdg.c to add modifiable beta
* via sysctls.
*
*/

#include <sys/cdefs.h>
__FBSDID("$FreeBSD: releng/12.1/sys/netinet/cc/cc_newreno.c"
                        " 347901 2019-05-17 08:21:27Z tuexen $");

#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/malloc.h>
#include <sys/module.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/sysctl.h>
#include <sys/systm.h>

#include <net/vnet.h>

#include <netinet/tcp.h>
#include <netinet/tcp_seq.h>
#include <netinet/tcp_var.h>
#include <netinet/cc/cc.h>
#include <netinet/cc/cc_module.h>
#include <netinet/cc/cc_newreno.h>

static MALLOC_DEFINE(M_NEWRENO, "newreno data",
    "newreno beta values");

static void newreno_cb_destroy(struct cc_var *ccv);
static void newreno_ack_received(struct cc_var *ccv, uint16_t type);
static void newreno_after_idle(struct cc_var *ccv);
static void newreno_cong_signal(struct cc_var *ccv, uint32_t type);
static void newreno_post_recovery(struct cc_var *ccv);
static int newreno_ctl_output(struct cc_var *ccv, struct sockopt *sopt, void *buf);

VNET_DEFINE_STATIC(uint32_t, newreno_beta) = 50;
VNET_DEFINE_STATIC(uint32_t, newreno_beta_ecn) = 80;
#define V_newreno_beta VNET(newreno_beta)
#define V_newreno_beta_ecn VNET(newreno_beta_ecn)
```

```
struct cc_algo newreno_cc_algo = {
    .name = "newreno",
    .cb_destroy = newreno_cb_destroy,
    .ack_received = newreno_ack_received,
    .after_idle = newreno_after_idle,
    .cong_signal = newreno_cong_signal,
    .post_recovery = newreno_post_recovery,
    .ctl_output = newreno_ctl_output,
};

struct newreno {
    uint32_t beta;
    uint32_t beta_ecn;
};

static inline struct newreno *
newreno_malloc(struct cc_var *ccv)
{
    struct newreno *nreno;

    nreno = malloc(sizeof(struct newreno), M_NEWRENO, M_NOWAIT);
    if (nreno != NULL) {
        /* NB: nreno is not zeroed, so initialise all fields. */
        nreno->beta = V_newreno_beta;
        nreno->beta_ecn = V_newreno_beta_ecn;
        ccv->cc_data = nreno;
    }

    return (nreno);
}

static void
newreno_cb_destroy(struct cc_var *ccv)
{
    free(ccv->cc_data, M_NEWRENO);
}

static void
newreno_ack_received(struct cc_var *ccv, uint16_t type)
{
    if (type == CC_ACK && !IN_RECOVERY(CCV(ccv, t_flags)) &&
        (ccv->flags & CCF_CWND_LIMITED)) {
        u_int cw = CCV(ccv, snd_cwnd);
        u_int incr = CCV(ccv, t_maxseg);

        /*
        * Regular in-order ACK, open the congestion window.
        * Method depends on which congestion control state we're
        * in (slow start or cong avoid) and if ABC (RFC 3465) is
        * enabled.
        *
        * slow start: cwnd <= ssthresh
```

```
        * cong avoid: cwnd > ssthresh
        *
        * slow start and ABC (RFC 3465):
        *   Grow cwnd exponentially by the amount of data
        *   ACKed capping the max increment per ACK to
        *   (abc_l_var * maxseg) bytes.
        *
        * slow start without ABC (RFC 5681):
        *   Grow cwnd exponentially by maxseg per ACK.
        *
        * cong avoid and ABC (RFC 3465):
        *   Grow cwnd linearly by maxseg per RTT for each
        *   cwnd worth of ACKed data.
        *
        * cong avoid without ABC (RFC 5681):
        *   Grow cwnd linearly by approximately maxseg per RTT using
        *   maxseg^2 / cwnd per ACK as the increment.
        *   If cwnd > maxseg^2, fix the cwnd increment at 1 byte to
        *   avoid capping cwnd.
        */
        if (cw > CCV(ccv, snd_ssthresh)) {
            if (V_tcp_do_rfc3465) {
                if (ccv->flags & CCF_ABC_SENTAWND)
                    ccv->flags &= ~CCF_ABC_SENTAWND;
                else
                    incr = 0;
            } else
                incr = max((incr * incr / cw), 1);
        } else if (V_tcp_do_rfc3465) {
            /*
            * In slow-start with ABC enabled and no RTO in sight?
            * (Must not use abc_l_var > 1 if slow starting after
            * an RTO. On RTO, snd_nxt = snd_una, so the
            * snd_nxt == snd_max check is sufficient to
            * handle this).
            *
            * XXXLAS: Find a way to signal SS after RTO that
            * doesn't rely on tcpcb vars.
            */
            if (CCV(ccv, snd_nxt) == CCV(ccv, snd_max))
                incr = min(ccv->bytes_this_ack,
                    ccv->nsegs * V_tcp_abc_l_var *
                    CCV(ccv, t_maxseg));
            else
                incr = min(ccv->bytes_this_ack, CCV(ccv, t_maxseg));
        }
        /* ABC is on by default, so incr equals 0 frequently. */
        if (incr > 0)
            CCV(ccv, snd_cwnd) = min(cw + incr,
                TCP_MAXWIN << CCV(ccv, snd_scale));
    }
}
```

```
static void
newreno_after_idle(struct cc_var *ccv)
{
    int rw;

    /*
    * If we've been idle for more than one retransmit timeout the old
    * congestion window is no longer current and we have to reduce it to
    * the restart window before we can transmit again.
    *
    * The restart window is the initial window or the last CWND, whichever
    * is smaller.
    *
    * This is done to prevent us from flooding the path with a full CWND at
    * wirespeed, overloading router and switch buffers along the way.
    *
    * See RFC5681 Section 4.1. "Restarting Idle Connections".
    */
    if (V_tcp_do_rfc3390)
        rw = min(4 * CCV(ccv, t_maxseg),
            max(2 * CCV(ccv, t_maxseg), 4380));
    else
        rw = CCV(ccv, t_maxseg) * 2;

    CCV(ccv, snd_cwnd) = min(rw, CCV(ccv, snd_cwnd));
}

/*
* Perform any necessary tasks before we enter congestion recovery.
*/
static void
newreno_cong_signal(struct cc_var *ccv, uint32_t type)
{
    struct newreno *nreno;
    uint32_t beta, beta_ecn, cwin, factor;
    u_int mss;

    cwin = CCV(ccv, snd_cwnd);
    mss = CCV(ccv, t_maxseg);
    nreno = ccv->cc_data;
    beta = (nreno == NULL) ? V_newreno_beta : nreno->beta;
    beta_ecn = (nreno == NULL) ? V_newreno_beta_ecn : nreno->beta_ecn;
    if (V_cc_do_abe && type == CC_ECN)
        factor = beta_ecn;
    else
        factor = beta;

    /* Catch algos which mistakenly leak private signal types. */
    KASSERT((type & CC_SIGPRIVMASK) == 0,
        ("%s: congestion signal type 0x%08x is private\n", __func__, type));

    cwin = max(((uint64_t)cwin * (uint64_t)factor) / (100ULL * (uint64_t)mss),
        2) * mss;
```

```
    switch (type) {
    case CC_NDUPACK:
        if (!IN_FASTRECOVERY(CCV(ccv, t_flags))) {
            if (IN_CONGRECOVERY(CCV(ccv, t_flags) &&
                V_cc_do_abe && V_cc_abe_frlossreduce)) {
                CCV(ccv, snd_ssthresh) =
                    ((uint64_t)CCV(ccv, snd_ssthresh) *
                    (uint64_t)beta) /
                    (100ULL * (uint64_t)beta_ecn);
            }
            if (!IN_CONGRECOVERY(CCV(ccv, t_flags)))
                CCV(ccv, snd_ssthresh) = cwin;
            ENTER_RECOVERY(CCV(ccv, t_flags));
        }
        break;
    case CC_ECN:
        if (!IN_CONGRECOVERY(CCV(ccv, t_flags))) {
            CCV(ccv, snd_ssthresh) = cwin;
            CCV(ccv, snd_cwnd) = cwin;
            ENTER_CONGRECOVERY(CCV(ccv, t_flags));
        }
        break;
    }
}

/*
* Perform any necessary tasks before we exit congestion recovery.
*/
static void
newreno_post_recovery(struct cc_var *ccv)
{
    int pipe;

    if (IN_FASTRECOVERY(CCV(ccv, t_flags))) {
        /*
        * Fast recovery will conclude after returning from this
        * function.  Window inflation should have left us with
        * approximately snd_ssthresh outstanding data.  But in case we
        * would be inclined to send a burst, better to do it via the
        * slow start mechanism.
        *
        * XXXLAS: Find a way to do this without needing curack
        */
        if (V_tcp_do_rfc6675_pipe)
            pipe = tcp_compute_pipe(ccv->ccvc.tcp);
        else
            pipe = CCV(ccv, snd_max) - ccv->curack;

        if (pipe < CCV(ccv, snd_ssthresh))
            /*
            * Ensure that cwnd does not collapse to 1 MSS under
            * adverse conditons. Implements RFC6582
```

```
            */
            CCV(ccv, snd_cwnd) = max(pipe, CCV(ccv, t_maxseg)) +
                CCV(ccv, t_maxseg);
        else
            CCV(ccv, snd_cwnd) = CCV(ccv, snd_ssthresh);
    }
}

static int
newreno_ctl_output(struct cc_var *ccv, struct sockopt *sopt, void *buf)
{
    struct newreno *nreno;
    struct cc_newreno_opts *opt;

    if (sopt->sopt_valsize != sizeof(struct cc_newreno_opts))
        return (EMSGSIZE);

    nreno = ccv->cc_data;
    opt = buf;

    switch (sopt->sopt_dir) {
    case SOPT_SET:
        /* We cannot set without cc_data memory. */
        if (nreno == NULL) {
            nreno = newreno_malloc(ccv);
            if (nreno == NULL)
                return (ENOMEM);
        }
        switch (opt->name) {
        case CC_NEWRENO_BETA:
            nreno->beta = opt->val;
            break;
        case CC_NEWRENO_BETA_ECN:
            if (!V_cc_do_abe)
                return (EACCES);
            nreno->beta_ecn = opt->val;
            break;
        default:
            return (ENOPROTOOPT);
        }
        break;
    case SOPT_GET:
        switch (opt->name) {
        case CC_NEWRENO_BETA:
            opt->val = (nreno == NULL) ?
                V_newreno_beta : nreno->beta;
            break;
        case CC_NEWRENO_BETA_ECN:
            opt->val = (nreno == NULL) ?
                V_newreno_beta_ecn : nreno->beta_ecn;
            break;
        default:
            return (ENOPROTOOPT);
```

```
        }
        break;
    default:
        return (EINVAL);
    }

    return (0);
}

static int
newreno_beta_handler(SYSCTL_HANDLER_ARGS)
{
    int error;
    uint32_t new;

    new = *(uint32_t *)arg1;
    error = sysctl_handle_int(oidp, &new, 0, req);
    if (error == 0 && req->newptr != NULL ) {
        if (arg1 == &VNET_NAME(newreno_beta_ecn) && !V_cc_do_abe)
            error = EACCES;
        else if (new == 0 || new > 100)
            error = EINVAL;
        else
            *(uint32_t *)arg1 = new;
    }

    return (error);
}

SYSCTL_DECL(_net_inet_tcp_cc_newreno);
SYSCTL_NODE(_net_inet_tcp_cc, OID_AUTO, newreno, CTLFLAG_RW, NULL,
    "New Reno related settings");

SYSCTL_PROC(_net_inet_tcp_cc_newreno, OID_AUTO, beta,
    CTLFLAG_VNET | CTLTYPE_UINT | CTLFLAG_RW,
    &VNET_NAME(newreno_beta), 3, &newreno_beta_handler, "IU",
    "New Reno beta, specified as number between 1 and 100");

SYSCTL_PROC(_net_inet_tcp_cc_newreno, OID_AUTO, beta_ecn,
    CTLFLAG_VNET | CTLTYPE_UINT | CTLFLAG_RW,
    &VNET_NAME(newreno_beta_ecn), 3, &newreno_beta_handler, "IU",
    "New Reno beta ecn, specified as number between 1 and 100");

DECLARE_CC_MODULE(newreno, &newreno_cc_algo);
```

Listing C.2: The implementation file for NewReno in FreeBSD.

C.2 Dynamic ABE

This section contains the source code for the implementation of our work on DABE, along with instructions for how to use it. The code is based on the original NewReno module above in Appendix C.1, with

the necessary modifications added. Our solution consists of only the implementation file cc_dabe.c from C.3, and the Makefile from C.4 has been used to compile our kernel module.

To compile the module, the kernel source is needed as explained in Section 3.4. With this in place, and both the cc_dabe.c and Makefile in the same directory, the following commands are used.

```
# Compile DABE
make
# Load DABE kernel module after compiled
make load
# Unload DABE
make unload
```

The implemention file and its Makefile follows.

CC_DABE.C

```
/*-
* SPDX-License-Identifier: BSD-2-Clause-FreeBSD
*
* Copyright (c) 1982, 1986, 1988, 1990, 1993, 1994, 1995
*    The Regents of the University of California.
* Copyright (c) 2007-2008,2010,2014
*    Swinburne University of Technology, Melbourne, Australia.
* Copyright (c) 2009-2010 Lawrence Stewart <lstewart@freebsd.org>
* Copyright (c) 2010 The FreeBSD Foundation
* All rights reserved.
*
* This software was developed at the Centre for Advanced Internet
* Architectures, Swinburne University of Technology, by Lawrence Stewart, James
* Healy and David Hayes, made possible in part by a grant from the Cisco
* University Research Program Fund at Community Foundation Silicon Valley.
*
* Portions of this software were developed at the Centre for Advanced
* Internet Architectures, Swinburne University of Technology, Melbourne,
* Australia by David Hayes under sponsorship from the FreeBSD Foundation.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the above copyright
*    notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in the
*    documentation and/or other materials provided with the distribution.
*
* THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
```

```
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */

/*
 * This software was first released in 2007 by James Healy and Lawrence Stewart
 * whilst working on the NewTCP research project at Swinburne University of
 * Technology's Centre for Advanced Internet Architectures, Melbourne,
 * Australia, which was made possible in part by a grant from the Cisco
 * University Research Program Fund at Community Foundation Silicon Valley.
 * More details are available at:
 *   http://caia.swin.edu.au/urp/newtcp/
 *
 * Dec 2014 garmitage@swin.edu.au
 * Borrowed code fragments from cc_cdg.c to add modifiable beta
 * via sysctls.
 *
 */

#include <sys/cdefs.h>

#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/malloc.h>
#include <sys/module.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/sysctl.h>
#include <sys/systm.h>

#include <net/vnet.h>

#include <netinet/tcp.h>
#include <netinet/tcp_seq.h>
#include <netinet/tcp_var.h>
#include <netinet/cc/cc.h>
#include <netinet/cc/cc_module.h>

#include <sys/khelp.h>
#include <netinet/khelp/h_ertt.h>

static MALLOC_DEFINE(M_NEWRENO, "newreno_dabe data",
    "newreno_dabe beta values");

static void newreno_dabe_cb_destroy(struct cc_var *ccv);
static void newreno_dabe_ack_received(struct cc_var *ccv, uint16_t type);
static void newreno_dabe_after_idle(struct cc_var *ccv);
static void newreno_dabe_cong_signal(struct cc_var *ccv, uint32_t type);
static void newreno_dabe_post_recovery(struct cc_var *ccv);
```

```
static int newreno_dabe_ctl_output(struct cc_var *ccv, struct sockopt *sopt, void *buf);
static int newreno_dabe_init(void);

VNET_DEFINE_STATIC(uint32_t, newreno_dabe_beta) = 50;
VNET_DEFINE_STATIC(uint32_t, newreno_dabe_beta_ecn) = 80;
#define V_newreno_dabe_beta VNET(newreno_dabe_beta)
#define V_newreno_dabe_beta_ecn VNET(newreno_dabe_beta_ecn)

VNET_DEFINE(int, cc_abe_frlossreduce) = 0;
#define V_cc_abe_frlossreduce VNET(cc_abe_frlossreduce)
VNET_DEFINE(int, cc_do_abe) = 1;
#define V_cc_do_abe VNET(cc_do_abe)
VNET_DEFINE(int, tcp_abc_l_var) = 2;
#define V_tcp_abc_l_var VNET(tcp_abc_l_var)
VNET_DEFINE(int, tcp_do_rfc3465) = 1;
#define V_tcp_do_rfc3465 VNET(tcp_do_rfc3465)
VNET_DEFINE(int, tcp_do_rfc3390) = 1;
#define V_tcp_do_rfc3390 VNET(tcp_do_rfc3390)
VNET_DEFINE(int, tcp_do_rfc6675_pipe) = 0;
#define V_tcp_do_rfc6675_pipe VNET(tcp_do_rfc6675_pipe)

struct cc_newreno_dabe_opts {
    int         name;
    uint32_t    val;
};

#define CC_NEWRENO_BETA         1
#define CC_NEWRENO_BETA_ECN     2

struct cc_algo newreno_dabe_cc_algo = {
    .name = "dabe",
    .cb_destroy = newreno_dabe_cb_destroy,
    .ack_received = newreno_dabe_ack_received,
    .after_idle = newreno_dabe_after_idle,
    .cong_signal = newreno_dabe_cong_signal,
    .post_recovery = newreno_dabe_post_recovery,
    .ctl_output = newreno_dabe_ctl_output,
    .mod_init = newreno_dabe_init,
};

static int32_t ertt_id;

struct newreno_dabe {
    uint32_t beta;
    uint32_t beta_ecn;
};

static int
newreno_dabe_init(void)
{
    ertt_id = khelp_get_id("ertt");
    if (ertt_id <= 0) {
        printf("%s: h_ertt module not found\n", __func__);
```

```
        return (ENOENT);
    }
    return (0);
}

static inline struct newreno_dabe *
newreno_dabe_malloc(struct cc_var *ccv)
{
    struct newreno_dabe *nreno;

    nreno = malloc(sizeof(struct newreno_dabe), M_NEWRENO, M_NOWAIT);
    if (nreno != NULL) {
        /* NB: nreno is not zeroed, so initialise all fields. */
        nreno->beta = V_newreno_dabe_beta;
        nreno->beta_ecn = V_newreno_dabe_beta_ecn;
        ccv->cc_data = nreno;
    }

    return (nreno);
}

static void
newreno_dabe_cb_destroy(struct cc_var *ccv)
{
    free(ccv->cc_data, M_NEWRENO);
}

static void
newreno_dabe_ack_received(struct cc_var *ccv, uint16_t type)
{
    if (type == CC_ACK && !IN_RECOVERY(CCV(ccv, t_flags)) &&
        (ccv->flags & CCF_CWND_LIMITED)) {
        u_int cw = CCV(ccv, snd_cwnd);
        u_int incr = CCV(ccv, t_maxseg);

        /*
        * Regular in-order ACK, open the congestion window.
        * Method depends on which congestion control state we're
        * in (slow start or cong avoid) and if ABC (RFC 3465) is
        * enabled.
        *
        * slow start: cwnd <= ssthresh
        * cong avoid: cwnd > ssthresh
        *
        * slow start and ABC (RFC 3465):
        *   Grow cwnd exponentially by the amount of data
        *   ACKed capping the max increment per ACK to
        *   (abc_l_var * maxseg) bytes.
        *
        * slow start without ABC (RFC 5681):
        *   Grow cwnd exponentially by maxseg per ACK.
        *
        * cong avoid and ABC (RFC 3465):
```

```
         *   Grow cwnd linearly by maxseg per RTT for each
         *   cwnd worth of ACKed data.
         *
         * cong avoid without ABC (RFC 5681):
         *   Grow cwnd linearly by approximately maxseg per RTT using
         *   maxseg^2 / cwnd per ACK as the increment.
         *   If cwnd > maxseg^2, fix the cwnd increment at 1 byte to
         *   avoid capping cwnd.
         */
        if (cw > CCV(ccv, snd_ssthresh)) {
            if (V_tcp_do_rfc3465) {
                if (ccv->flags & CCF_ABC_SENTAWND)
                    ccv->flags &= ~CCF_ABC_SENTAWND;
                else
                    incr = 0;
            } else
                incr = max((incr * incr / cw), 1);
        } else if (V_tcp_do_rfc3465) {
            /*
            * In slow-start with ABC enabled and no RTO in sight?
            * (Must not use abc_l_var > 1 if slow starting after
            * an RTO. On RTO, snd_nxt = snd_una, so the
            * snd_nxt == snd_max check is sufficient to
            * handle this).
            *
            * XXXLAS: Find a way to signal SS after RTO that
            * doesn't rely on tcpcb vars.
            */
            if (CCV(ccv, snd_nxt) == CCV(ccv, snd_max))
                incr = min(ccv->bytes_this_ack,
                    ccv->nsegs * V_tcp_abc_l_var *
                    CCV(ccv, t_maxseg));
            else
                incr = min(ccv->bytes_this_ack, CCV(ccv, t_maxseg));
        }
        /* ABC is on by default, so incr equals 0 frequently. */
        if (incr > 0)
            CCV(ccv, snd_cwnd) = min(cw + incr,
                TCP_MAXWIN << CCV(ccv, snd_scale));
    }
}

static void
newreno_dabe_after_idle(struct cc_var *ccv)
{
    int rw;

    /*
    * If we've been idle for more than one retransmit timeout the old
    * congestion window is no longer current and we have to reduce it to
    * the restart window before we can transmit again.
    *
    * The restart window is the initial window or the last CWND, whichever
```

```
     * is smaller.
     *
     * This is done to prevent us from flooding the path with a full CWND at
     * wirespeed, overloading router and switch buffers along the way.
     *
     * See RFC5681 Section 4.1. "Restarting Idle Connections".
     */
    if (V_tcp_do_rfc3390)
        rw = min(4 * CCV(ccv, t_maxseg),
            max(2 * CCV(ccv, t_maxseg), 4380));
    else
        rw = CCV(ccv, t_maxseg) * 2;

    CCV(ccv, snd_cwnd) = min(rw, CCV(ccv, snd_cwnd));
}

/*
* Perform any necessary tasks before we enter congestion recovery.
*/
static void
newreno_dabe_cong_signal(struct cc_var *ccv, uint32_t type)
{
    struct newreno_dabe *nreno;
    uint32_t beta, beta_ecn, cwin, factor;
    u_int mss;

    struct ertt *e_t = khelp_get_osd(CCV(ccv, osd), ertt_id);

    cwin = CCV(ccv, snd_cwnd);
    mss = CCV(ccv, t_maxseg);
    nreno = ccv->cc_data;
    beta = (nreno == NULL) ? V_newreno_dabe_beta : nreno->beta;
    beta_ecn = (nreno == NULL) ? V_newreno_dabe_beta_ecn : nreno->beta_ecn;
    if (V_cc_do_abe && type == CC_ECN)
        factor = beta_ecn;
    else
        factor = beta;

    /* Catch algos which mistakenly leak private signal types. */
    KASSERT((type & CC_SIGPRIVMASK) == 0,
        ("%s: congestion signal type 0x%08x is private\n", __func__, type));

    cwin = max(((uint64_t)cwin * (uint64_t)factor) / (100ULL * (uint64_t)mss),
        2) * mss;

    switch (type) {
    case CC_NDUPACK:
        if (!IN_FASTRECOVERY(CCV(ccv, t_flags))) {
            if (IN_CONGRECOVERY(CCV(ccv, t_flags) &&
                V_cc_do_abe && V_cc_abe_frlossreduce)) {
                CCV(ccv, snd_ssthresh) =
                    ((uint64_t)CCV(ccv, snd_ssthresh) *
                    (uint64_t)beta) /
```

```
                    (100ULL * (uint64_t)beta_ecn);
            }
            if (!IN_CONGRECOVERY(CCV(ccv, t_flags)))
                CCV(ccv, snd_ssthresh) = cwin;
            ENTER_RECOVERY(CCV(ccv, t_flags));
        }
        break;
    case CC_ECN:
        if (!IN_CONGRECOVERY(CCV(ccv, t_flags))) {
            CCV(ccv, snd_ssthresh) = CCV(ccv, snd_cwnd) * e_t->minrtt / e_t->rtt;
            CCV(ccv, snd_cwnd) = CCV(ccv, snd_ssthresh);
            ENTER_CONGRECOVERY(CCV(ccv, t_flags));
        }
        break;
    }
}

/*
* Perform any necessary tasks before we exit congestion recovery.
*/
static void
newreno_dabe_post_recovery(struct cc_var *ccv)
{
    int pipe;

    if (IN_FASTRECOVERY(CCV(ccv, t_flags))) {
        /*
        * Fast recovery will conclude after returning from this
        * function. Window inflation should have left us with
        * approximately snd_ssthresh outstanding data. But in case we
        * would be inclined to send a burst, better to do it via the
        * slow start mechanism.
        *
        * XXXLAS: Find a way to do this without needing curack
        */
        if (V_tcp_do_rfc6675_pipe)
            pipe = tcp_compute_pipe(ccv->ccvc.tcp);
        else
            pipe = CCV(ccv, snd_max) - ccv->curack;

        if (pipe < CCV(ccv, snd_ssthresh))
            /*
            * Ensure that cwnd does not collapse to 1 MSS under
            * adverse conditons. Implements RFC6582
            */
            CCV(ccv, snd_cwnd) = max(pipe, CCV(ccv, t_maxseg)) +
                CCV(ccv, t_maxseg);
        else
            CCV(ccv, snd_cwnd) = CCV(ccv, snd_ssthresh);
    }
}

static int
```

```
newreno_dabe_ctl_output(struct cc_var *ccv, struct sockopt *sopt, void *buf)
{
    struct newreno_dabe *nreno;
    struct cc_newreno_dabe_opts *opt;

    if (sopt->sopt_valsize != sizeof(struct cc_newreno_dabe_opts))
        return (EMSGSIZE);

    nreno = ccv->cc_data;
    opt = buf;

    switch (sopt->sopt_dir) {
    case SOPT_SET:
        /* We cannot set without cc_data memory. */
        if (nreno == NULL) {
            nreno = newreno_dabe_malloc(ccv);
            if (nreno == NULL)
                return (ENOMEM);
        }
        switch (opt->name) {
        case CC_NEWRENO_BETA:
            nreno->beta = opt->val;
            break;
        case CC_NEWRENO_BETA_ECN:
            if (!V_cc_do_abe)
                return (EACCES);
            nreno->beta_ecn = opt->val;
            break;
        default:
            return (ENOPROTOOPT);
        }
        break;
    case SOPT_GET:
        switch (opt->name) {
        case CC_NEWRENO_BETA:
            opt->val = (nreno == NULL) ?
                V_newreno_dabe_beta : nreno->beta;
            break;
        case CC_NEWRENO_BETA_ECN:
            opt->val = (nreno == NULL) ?
                V_newreno_dabe_beta_ecn : nreno->beta_ecn;
            break;
        default:
            return (ENOPROTOOPT);
        }
        break;
    default:
        return (EINVAL);
    }

    return (0);
}
```

```
static int
newreno_dabe_beta_handler(SYSCTL_HANDLER_ARGS)
{
    int error;
    uint32_t new;

    new = *(uint32_t *)arg1;
    error = sysctl_handle_int(oidp, &new, 0, req);
    if (error == 0 && req->newptr != NULL ) {
        if (arg1 == &VNET_NAME(newreno_dabe_beta_ecn) && !V_cc_do_abe)
            error = EACCES;
        else if (new == 0 || new > 100)
            error = EINVAL;
        else
            *(uint32_t *)arg1 = new;
    }

    return (error);
}

SYSCTL_DECL(_net_inet_tcp_cc_abe);
SYSCTL_NODE(_net_inet_tcp_cc, OID_AUTO, newreno_dabe, CTLFLAG_RW, NULL,
    "New Reno related settings");

SYSCTL_PROC(_net_inet_tcp_cc_abe, OID_AUTO, beta,
    CTLFLAG_VNET | CTLTYPE_UINT | CTLFLAG_RW,
    &VNET_NAME(newreno_dabe_beta), 3, &newreno_dabe_beta_handler, "IU",
    "New Reno beta, specified as number between 1 and 100");

SYSCTL_PROC(_net_inet_tcp_cc_abe, OID_AUTO, beta_ecn,
    CTLFLAG_VNET | CTLTYPE_UINT | CTLFLAG_RW,
    &VNET_NAME(newreno_dabe_beta_ecn), 3, &newreno_dabe_beta_handler, "IU",
    "New Reno beta ecn, specified as number between 1 and 100");

DECLARE_CC_MODULE(newreno_dabe, &newreno_dabe_cc_algo);
MODULE_DEPEND(newreno_dabe, ertt, 1, 1, 1);
```

Listing C.3: The implementation file for DABE.

MAKEFILE

```
KMOD = cc_dabe
SRCS = cc_dabe.c
.include <bsd.kmod.mk>
```

Listing C.4: The Makefile for cc_dabe.c.

Terms

Acknowledgement (ACK) A signal that is passed between communicating processes, computers, or devices to signify acknowledgement, or receipt of message, as part of a communications protocol.

Active Queue Management (AQM) The policy in routers and switches for what to do with packets inside a buffer before it becomes full.

Additive-increase/Multiplicative-decrease (AIMD) A feedback control algorithm best known for its use in TCP congestion control. AIMD combines linear growth of the congestion window with an exponential reduction when congestion is detected.

Alternative Backoff with ECN (ABE) A simple sender-side only modification that, upon a receipt of en ECN mark, reduces the CWND by less than the usual response for loss.

Bandwidth Delay Product (BDP) The product of a data link's capacity and RTT. Widely used in determing the buffer size.

Centre for Advanced Internet Architectures (CAIA) Research center in part of the Faculty of Science, Engineering and Technology at Swinburne University of Technology. Currently, as part of a broader organisational restructure, CAIA is being moved to the Internet For Things (I4T) Research Lab.

Congestion Control (CC) The process of managing the sender's packet rate to not overwhelm the network.

Congestion Encountered (CE) A bit set in the IP header by a router with AQM support to indicate congestion in an ECN-capable network.

Congestion Window (CWND) A TCP state variable that limits the amount of data the sender can send into the network before receiving an ACK.

Congestion Window Reduced (CWR) A flag in the TCP header so that a sender can inform the receiver that the CWND has been reduced. Only applicable in an ECN-capable network.

Controlled Delay (CoDel) A more recent AQM that seeks to control bufferbloat-generated excess delay in modern networking environments.

Dynamic Alternative Backoff with ECN (DABE) Our work building upon ABE. Instead of reducing the CWND by a constant factor in the event of a ECN congestion signal, the reduction is now based on the shortest RTT measured divided by the most recent RTT.

Dynamic Host Configuration Protocol (DHCP) A client/server protocol that automatically provides an Internet Protocol (IP) host with its IP address and other related configuration information such as the subnet mask and default gateway.

ECN-Capable Transport (ECT) A bit set in the IP header to indicate if a transport protocol connection is ECN-capable.

ECN-Echo (ECE) A flag in the TCP header so that a receiver can echo back to the sender when a CE packet has been received. Only applicable in an ECN-capable network.

Explicit Congestion Notification (ECN) An extension to IP and TCP that allows end-to-end notification of network congestion without dropping packets.

Internet Engineering Task Force (IETF) An open standards organization which develops and promotes voluntary Internet standards.

Internet Protocol (IP) The principal communications protocol for relaying datagrams across network boundaries. Its routing function enables internetworking, and essentially establishes the Internet.

Maximum Segment Size (MSS) The largest specified amount of data (in bytes) that a communications device can receive in a single TCP segment.

Network Address Translation (NAT) A method of remapping one IP address space into another. Often used such that one Internet-routable IP address of a NAT gateway can be used for an entire private network. This is used in conjunction with IP masquerading, which is a technique that hides an entire IP address space, usually consisting of private IP addresses, behind a single IP address in another, usually public address space.

Network Time Protocol (NTP) A networking protocol for clock synchronization between computer systems.

Operating System (OS) System software that manages computer hardware, software resources, and provides common services for computer programs.

Power over Ethernet (PoE) A standard or ad hoc systems which pass electric power along with data on twisted pair Ethernet cabling. This allows a single cable to provide both data connection and electric power to devices such as wireless access points, IP cameras, VoIP phones and later Raspberry Pi machines.

Proportional Integral Controller Enhanced (PIE) A more recent and lightweight AQM that seeks to control the average queuing latency to a target value.

Random Early Detection (RED) One of the first and most widely supported AQM algorithms. It monitors the average queue size and drops packets based on statistical probabilities.

Receiver Window (RWND) A TCP state variable that advertises the amount of data that the receiver can receive.

Request for Comments (RFC) A formal document drafted by engineers and computer scientists describing the specifications for a particular technology.

Round Trip Time (RTT) The time it takes for a signal to be sent plus the time for an ACK of that signal to be received.

Slow start threshold (ssthresh) A TCP state variable used to determine whether the slow start or congestion avoidance algorithm is used to control data transmission.

TCP Experiment Automation Controlled Using Python (TEACUP) A tool for automating repeatable TCP experiments using Python.

Transmission Control Protocol (TCP) One of the main communication protocols of the Internet that defines how to establish and maintain a network conversation through which applications can exchange data.

Virtual LAN (VLAN) Any broadcast domain that is partitioned and isolated in a computer network at the data link layer (OSI layer 2).

World Wide Web (WWW) Commonly known as the Web; an information system in the form of web-pages that web browsers can read and interact with.

References

[1] J. Gettys, "Bufferbloat: Dark buffers in the internet," *IEEE Internet Computing*, vol. 15, no. 3, pp. 96–96, 2011.

[2] M. Welzl, D. Papadimitriou, B. Briscoe, M. Scharf, and M. Welzl, *Open Research Issues in Internet Congestion Control*, RFC 6077, Feb. 2011. DOI: 10.17487/RFC6077. [Online]. Available: https://rfc-editor.org/rfc/rfc6077.txt.

[3] B. Trammell, M. Kühlewind, D. Boppart, I. Learmonth, G. Fairhurst, and R. Scheffenegger, "Enabling internet-wide deployment of explicit congestion notification," vol. 8995, Mar. 2015, pp. 193–205. DOI: 10.1007/978-3-319-15509-8_15.

[4] N. Khademi, G. Armitage, M. Welzl, S. Zander, G. Fairhurst, and D. Ros, "Alternative backoff: Achieving low latency and high throughput with ecn and aqm," Jun. 2017, pp. 1–9. DOI: 10.23919/IFIPNetworking.2017.8264863.

[5] E. Blanton, D. V. Paxson, and M. Allman, *TCP Congestion Control*, RFC 5681, Sep. 2009. DOI: 10.17487/RFC5681. [Online]. Available: https://rfc-editor.org/rfc/rfc5681.txt.

[6] G. Appenzeller, I. Keslassy, and N. McKeown, "Sizing router buffers," in *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '04, Portland, Oregon, USA: Association for Computing Machinery, 2004, pp. 281–292, ISBN: 1581138628. DOI: 10.1145/1015467.1015499. [Online]. Available: https://doi.org/10.1145/1015467.1015499.

[7] S. Floyd and V. Jacobson, "On traffic phase effects in packet-switched gateways," *ACM SIGCOMM Computer Communication Review*, Feb. 1991.

[8] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, Aug. 1993, ISSN: 1558-2566. DOI: 10.1109/90.251892.

[9] V. Jacobson, "A rant on queues. a talk presented at mit lincoln labs, lexington, ma," 2006. [Online]. Available: http://www.pollere.net/Pdfdocs/QrantJul06.pdf.

[10] K. Nichols and V. Jacobson, "Controlling queue delay," *Queue*, vol. 10, no. 5, pp. 20–34, May 2012, ISSN: 1542-7730. DOI: 10.1145/2208917.2209336. [Online]. Available: https://doi.org/10.1145/2208917.2209336.

[11] K. Nichols, V. Jacobson, A. McGregor, and J. Iyengar, *Controlled Delay Active Queue Management*, RFC 8289, Jan. 2018. DOI: 10.17487/RFC8289. [Online]. Available: https://rfc-editor.org/rfc/rfc8289.html.

[12] R. Pan, P. Natarajan, F. Baker, and G. White, *Proportional Integral Controller Enhanced (PIE): A Lightweight Control Scheme to Address the Bufferbloat Problem*, RFC 8033, Feb. 2017. DOI: 10.17487/RFC8033. [Online]. Available: https://rfc-editor.org/rfc/rfc8033.html.

[13] S. Floyd, D. K. K. Ramakrishnan, and D. L. Black, *The Addition of Explicit Congestion Notification (ECN) to IP*, RFC 3168, Sep. 2001. DOI: 10.17487/RFC3168. [Online]. Available: https://rfc-editor.org/rfc/rfc3168.txt.

[14] J. H. Salim and U. Ahmed, *Performance Evaluation of Explicit Congestion Notification (ECN) in IP Networks*, RFC 2884, Jul. 2000. DOI: 10.17487/RFC2884. [Online]. Available: https://rfc-editor.org/rfc/rfc2884.txt.

[15] M. Menth, B. Briscoe, and T. Tsou, "Pre-congestion notification – new qos support for differentiated services ip networks," *IEEE Communications Magazine - IEEE Commun. Mag.*, vol. 50, pp. 94–103, Mar. 2012. DOI: 10.1109/MCOM.2012.6163587.

[16] D. L. Black, *Relaxing Restrictions on Explicit Congestion Notification (ECN) Experimentation*, RFC 8311, Jan. 2018. DOI: 10.17487/RFC8311. [Online]. Available: https://rfc-editor.org/rfc/rfc8311.txt.

[17] N. Khademi, M. Welzl, D. G. Armitage, and G. Fairhurst, *TCP Alternative Backoff with ECN (ABE)*, RFC 8511, Dec. 2018. DOI: 10.17487/RFC8511. [Online]. Available: https://rfc-editor.org/rfc/rfc8511.txt.

[18] S. Zande and G. Armitage, "Teacup v1.0 — a system for automated tcp testbed experiments," Tech. Rep., May 2015. [Online]. Available: http://caia.swin.edu.au/tools/teacup.