Rapid Acceleration in TCP Prague

Joakim Skjelbred Misund

Thesis submitted for the degree of
Master in Informatics: Programming and Networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2018

Rapid Acceleration in TCP Prague

Joakim Skjelbred Misund

Rapid Acceleration in TCP Prague

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

Printed: Reprosentralen, University of Oslo

Abstract

TCP slow start has a dilemma between accelerating fast and overshooting capacity. Accelerating fast causes more queuing delay for the new flow and existing flows. Reducing the overshoot requires a more gentle acceleration which increases the time it takes the flow to reach full utilization. We have designed, implemented and evaluated a new delay-based flow start algorithm called Paced Chirping. It is implemented in the Linux kernel with minimal changes to the existing pacing code, and evaluated on testbed. Paced Chirping was initially targeted at TCP Prague and DCTCP but, since it is purely delay-based, it should be applicable to all transport protocols using congestion control. Many attempts at solving the flow start problem have been made since its introduction in 1988, but in widespread usage the original algorithm remains unchanged. Paced Chirping seems to be a promising breakthrough that could resolve the enduring flow start dilemma of TCP. Even though we have identified limitations these have more to do with implementation, and we think they are surmountable.

Preface

Some part of me wish I had never opened the black box that is TCP. Once you realize the difficulty of the issues you face you might enter a state of despair. I have entered many times over the past year. Looking back, it was all worth it. You have to really understand something to be able to fix it. Once you really understand something you might realize that it is not possible to fix, only patch.

I would like to thank my supervisor Dr. Bob Briscoe for his invaluable knowledge and spending his time trying to teach me. I have still a lot to learn. I would also like to thank Dr. Andreas Petlund, Dr. David Hayes and the others at Simula.

Lastly I want to thank my family, especially my girlfriend Marthe, for giving moral support.

Contents

List of Figures

List of Tables

List of Algorithms

Glossary

TCP Transmission Control Protocol. Read the background.

RTT Round-trip time. The time it takes a packet to reach the receiver and the corresponding ack to return to the sender.

BDP Bandwidth-delay product. Amount of data a network path can have in transit. Can be specified in bytes and packets. Calculation: BDP $\leftarrow$ capacity $*$ RTT.

ECN Explicit Congestion Notification. A protocol for signaling congestion without having to drop packets. Cooperation between IP and TCP layers.

MSS Maximum Segment Size. The maximum number of bytes that can be encapsulated in a IP-packet over a specific path.

MTU Maximum Transmission Unit. The maximum size of a IP-layer packet over a certain link-layer technology. The MTU of Ethernet is 1500 Bytes.

AQM Active Queue Manager. Algorithm for how a network queue handles packets.

RED Random Early Detection. An AQM that can control the size of a queue by using a hard limit and linear drop/mark probability.

ack Acknowledgement. A packet that acknowledges the reception of data. A pure ACK is an acknowledgment packet without any data.

IW Initial Window. The number of packets sent in the first round-trip time of slow start. Sent as a burst.

cwnd Congestion Window. The maximal amount of data a flow can have in the network. Can be specified in bytes or packets. Linux uses packets, and so will we.

3WHS Three way hand shake. The three way hand shake is the three packets used to initiate a TCP-connection. The packets are referred to (in sequence) as syn, syn/ack and ack.

bps Bits per second. Commonly used for rates in computer networks.

FCT Flow completion time. The time it takes a flow to send all its data and get it acknowledged.

sd Standard deviation. Commonly used measurement of variation of a set of values.

Chapter 1

Introduction

Transmission Control Protocol (TCP) congestion control consists of two algorithms: slow start and congestion avoidance. Slow start is used when a new connection starts to accelerate to the network capacity, and when the capacity is reached congestion avoidance takes over. This thesis is about slow start.

Slow start makes a decision about which of rapid acceleration and capacity overshoot is most important. Choosing rapid acceleration comes at the cost of more severe overshoot which incurs delay and packet loss. Choosing to minimize the capacity overshoot leads to a slower acceleration to the capacity.

Originally we targeted slow start in the context of DCTCP, but we found the solution general enough to be applicable to other transport protocols using congestion control. Data Center TCP (DCTCP) is a congestion control that can achieve full utilization with very little queueing by using a signal called Explicit Congestion Notification (ECN). It is realized by marking packet with ECN at very low queue length threshold. This adds a new issue to slow start that we will refer to as undershoot. The threshold is usually exceeded long before slow start has reached the capacity which leaves the network unutilized when it enters congestion avoidance.

TCP Prague is a variant of DCTCP for use in the Internet. It is part of the L4S architecture [60] that aims to replace the current Internet's architecture for detecting available capacity. It aims to provide all applications with low latency, low loss, scalable throughput.

1.1 Motivation

Many applications use TCP to communicate reliably over the Internet and local networks. Every day an enormous number of TCP connections are initiated. Each of these runs the traditional slow start algorithm to get up to speed. Some have performance enhancing additions such as hybrid slow start.

Improving slow start so that it causes less congestion and accelerates faster would have a huge positive impact. We would have less packet loss due to buffer overruns and shorter transfer times, saving capacity and

improving utilization. This would also bring great economic benefits.

1.2 Contributions

We have developed a flow start algorithm called Paced Chirping to replace the current slow start algorithm. It is applicable to to all congestion controls, not just DCTCP. It has several issues, but we think that the general approach is a good starting point for a future flow start algorithm.

In regular slow start there is a coupling between measuring capacity and the offered load. To measure higher capacities the amount of data has to be increased proportionately. Paced Chirping can measure the capacity without having to increase the number of packets, to a degree. It decouples the filling of the network and identifying the capacity.

Incidental contributions

We have identified possible weaknesses in the pacing implementation in Linux related to power management in modern CPUs. The pacing internal to the kernel limits the sending rate to a certain rate, it does not try to enforce the rate.

Linux has pacing rate calculation turned on by default for most TCP CC algorithms. If FQ Qdisc (queuing discipline) is attached to the outgoing interface, pacing is applied. We show that the improvement in slow start throughput that this gives is at the cost of higher latency.

1.3 Scope

We started of by considering the problem in the context of a data center environment, but we shifted towards the Internet during the work. All of the experiments are done with network characteristics much closer to those in the Internet than in data centers. The solution does not assume support for ECN in the network. However, we have used a marking configuration that can be assumed to exist only in a data center environment.

This work is research to prove whether the idea of paced chirping can be made to improve flow start performance. If proven useful this will motivate protocol engineering to fix some of the limitations that make it hard to deploy. For example, the solution relies on quickacks on all receivers which is only feasible in a data center. We think of this as a protocol engineering problem, which is out of scope.

1.4 Roadmap

Chapters 2 to 5 make up the required background knowledge. Chapter 6 discusses the problem we are trying to solve. Chapter 7 presents and discusses some naive attempts made. Chapter 8 presents the main solution in this thesis. Chapter 9 discusses how the solution is implemented.

Chapter 10 discusses the testbed environment, methodology and the tools used. In chapter 11 the solution is evaluated with a series of experiments. Chapter 12 concludes and presents future work.

Part I

Background

Chapter 2

Transmission Control Protocol

In the following chapters I will describe and critique 3 relevant papers in depth while mentioning many others in passing. The critique is independent of the insights from the subsequent chapters on Paced Chirping.

In chapter 2 I will present traditional TCP. Chapter 3 builds on chapter chapter 2 and describes the current state of TCP Congestion Control. It presents the original paper by Jacobson that introduced TCP congestion control. Chapter 4 introduces Data Center TCP, the protocol which TCP Prague is based on. Chapter 5 introduces Hybrid slow start, a performance enhancing addition to traditional slow start. Hybrid slow start is on by default in the Linux kernel and is thus a widely used enhancement.

2.1 The seven functions of TCP

The Internet Protocol (IP) provides best effort delivery of packets. Packets can be lost, reordered, duplicated and corrupted. These events are unacceptable to applications requiring reliable delivery of data.

Transmission Control Protocol (TCP) provides in-order and reliable transfer over the IP protocol. It makes sure that what is sent is what the receiver gets. TCP also offers flow control, connection management, application multiplexing and congestion control. In the following sections we will describe the seven functions of TCP. TCP is defined in a series of standards and a roadmap can be found in [48].

A TCP connection consists of two conceptual half-connections. I will use sender and receiver to denote the end-points of a TCP half-connection. A half-connection is an asymmetric connection, meaning that one endpoint produces data and the other consumes the data. The consumer, or receiver, does not send any data over the half-connection, but it does send control information such as acknowledgements.

Reliable Transfer

In the context of TCP reliable transfer means that data is eventually delivered and delivered only once to the application.

Every sent byte is given a sequence number for identification. The receiver sends acknowledgements to the sender when data has been successfully received. An Acknowledgement (ACK) contains the sequence number of the byte that the receiver expects to receive next. The sender keeps track of which bytes it has sent and received acknowledgements for. Unacknowledged bytes are retransmitted when they are deemed lost. If the network and receiver stays operational data will eventually be delivered and acknowledged.

The receiver can detect and discard duplicated data by looking at the sequence numbers of received data.

Sequence numbers are represented by a 16-bit unsigned integer. It wraps around every 2^{16}th byte. When wrap around occurs order can not be determined by simply comparing sequence numbers and seeing which one is greater. Protection against wrapped sequence numbers [43], PAWS, is a mechanism that reduces the risk of accepting old duplicates using timestamps.

In-order delivery

In-order delivery is enforced at the receiver. The receiver keeps track of which bytes it has received and delivered to the application. When it receives new data it checks if the data is received in order. If it is in order the data and any stored (outstanding) data is delivered to the application. But if it is out of order the data is stored until it can be delivered. Data can only be acknowledged if all preceding data has been received because acknowledgements are cumulative. An acknowledgment says that every byte until the byte identified by the acknowledgement has been successfully received.

It is common to respond with an acknowledgement even though data is received out of order. Acknowledgements sent in response to out of order data contains the same sequence number as previous acknowledgments. These are commonly referred to as duplicate acknowledgements. We will use dup-ack as a short hand for duplicate acknowledgement.

A known issue with in order delivery is what is called head-of-line blocking. Loss of data will prevent subsequent data from being delivered. This incurs additional delay, and is a reason why certain time sensitive applications does not use TCP. Examples are live streaming and video-games.

Flow Control

Flow control is a mechanism that prevents a sender from sending more data that the receiver can handle. The receiver has a window that says how much data it is willing to receive. The window is sent in the TCP header to the sender on each acknowledgement. Originally it was limited to $2^{16} - 1$ bytes due to the header fields 16-bit size. TCP window scaling option allows the window to grow beyond $2^{16} - 1$ [43].

Application Multiplexing

An end-system normally has one point of attachment to a network. This attachment is through a network interface card (NIC). A NIC can have multiple IP-addresses, but TCP uses only one.

To be able to serve multiple application with the same IP-address TCP provides multiplexing by assigning each application with a unique port number. This port number is used to deliver data to the right application. A connection has two port numbers; one at the sender side and one at the receiver side. These do not have to be the same, and they rarely are.

Congestion Control

Networks which use store and forward for packets become congested if the sum of rates coming into the network exceed the capacity of the network. Network devices (routers, switches) have buffers to accommodate for short periods of congestion, but when it persists over time queues build up and packets are eventually dropped. If senders do not have a mechanism to react appropriately the network can experience congestion collapse [1]. There will be a huge amount of losses, increased delay, and application will perform poorly, if not fail.

TCP congestion control is a capacity-seeking mechanism, which in turn helps avoid severe congestion. It adapts the sending rate by reacting to feedback (or lack of feedback) from the network. Congestion in the field of congestion control is typically minor or transient caused by end-points seeking capacity. TCP congestion control will be discussed in detail in chapter 3.

Integrity

Corruption of data can occur anywhere between the sending and receiving of a TCP segment. To detect corruption TCP has a checksum calculated over the data. The checksum is sent in the TCP-header and can be used by the receiver to verify the integrity of the data. The checksum is weak [13], so checksums in lower layers are often required.

Connection Establishment and Maintenance

TCP is connection oriented meaning that it keeps state for each connection. A connection is uniquely identified by source and destination IP-address, port numbers and an initial sequence number (ISN) [38]. ISN protects against sequence number attacks and differentiates connections at different times.

Before any data can be sent TCP performs a handshake which sets up the connection. The initiator sends a syn-message to the destination. The destination responds with a syn/ack. The connection is established when the initiator responds with an acknowledgement. The handshake consists of three messages and is commonly referred to as the three way hand shake (3WHS).

During the lifetime of a connection TCP keeps track of the state of the connection. If one of the endpoints loose connectivity TCP is able to detect it and act accordingly. It also keeps track of if the connection is being actively used or if it is idle.

When communication is completed TCP normally performs a teardown-phase to terminate the connection. A endpoint that is finished sending data sends a FIN message to its peer. The peer responds with an acknowledgment. When both endpoints have completed this step the connection is terminated.

Chapter 3

TCP congestion control

3.1 Need for Congestion Control

In the mid 80's the Internet experienced a series of congestion collapses. These collapses caused the throughput on paths traversing the congested routers to drop significantly. The collapses were caused by filled queues in routers which led to increased latency and drop rate. At the time TCP lacked mechanisms to deal with congestion.

In 1988 Jacobson addressed this issue by proposing *congestion control* (CC) [2]. It provided algorithms which aimed at keeping the network stable by limit the amount of data each sender could have unacknowledged in the network. The amount of data a sender can have in the network is called the congestion window, or cwnd for short. It can also be thought of loosely as the rate a sender is allowed to send at. I will come back to the relation between cwnd and rate in section 3.3.2.

3.2 Congestion Detection

In this section we will go through how congestion is detected and how the network can signal congestion to end-systems.

3.2.1 Loss detection

Loss is not always caused by congestion in the network. Packets can be corrupted along path from the sender to the receiver. Network devices can be faulty or misbehaving. Nevertheless, since TCP cannot identify the cause of a loss it has to assume that the loss is caused by congestion.

Retransmission Timeout - RTO

TCP uses a timer to deem a packet as lost. This timer is called the retransmission timer, and its value is retransmission timeout (RTO). Each time data is sent a timer is started. If the sender does not receive an acknowledgement for that data before the timer expires the data is deemed

lost and retransmitted. This continues until the data is acknowledged or the network is considered non-operational.

The algorithmic description of the RTO calculation can be found in [35]. It is based on the algorithm in [2]. I will briefly go through the update of RTO. The value of RTO is initially set to 1 second, but thereafter it is based on an estimate of the RTT called smoothed RTT (SRTT) and its variance (RTTVAR). The update of RTO is as follows.

$$\text{RTO} \leftarrow \text{SRTT} + \max(G, K * \text{RTTVAR})$$

G is the clock granularity in seconds and K is set to 4. Taking variation into account makes the timer more robust against spurious timeouts. A spurious timeout occurs if the RTO is set too low causing a retransmission while the original packet or its acknowledgement is in the network. Spurious timeouts can make congestion problem worse because it creates unnecessary congestion, and it violates the packet conservation principle which will be discussed shortly.

Fast retransmit

Fast retransmit is an addition to TCP that tries to detect and repair packet loss quicker than the retransmit timer is able to. It is described in [27]. A sender deems a packet as lost if it receives three duplicate acks in a row. Following the third duplicate ack the sender resends the packet. The choice of three duplicate acks is unjustified, but it is an indication that a packet has been lost.

Fast retransmit is tightly coupled with fast recovery. Fast recovery has to do with how the congestion control reacts when fast recovery detects congestion. I will describe fast recovery in section 3.5.

Selective Acknowledgements - SACK

Selective acknowledgements (SACK) improves TCPs ability to detect multiple loss during one RTT. It is defined in [6] and extended in [12]. SACK is a TCP option that lets the receiver specify up to n blocks of data it has received. The sender can infer what data has been received by the receiver. Note that if TCP Timestamp option is used SACK can only carry three ranges [6].

Sack is computational heavy because the sender has to traverse a linked list of in-flight packets possibly multiple times on each ack. This problem is most prominent in network with high capacity as the linked list is longer and more computational expensive to traverse. [19, 22, 28]

Recent Acknowledgment - RACK

RACK is a time based proposal for detecting packet loss [55]. It deems a packet lost if a packet that was sent sufficiently later has been delivered. Each packet is timestamped and this is used with SACK to infer packet

loss. In environments with high degree of reordering RACK reduce the number of spurious retransmits.

RACK does not change the way TCP CC reacts to loss.

Delayed ACK

Delayed acknowledgements is described in [3], and it allows a receiver to send fewer acknowledgements than one per received data segment. Many TCP implementation sends one acknowledgement for every second packet received [43]. Delayed acks reduces network load and processing in the receiver.

A cost is that the sender looses accurate timing information. The timestamp of the most recent unacknowledged segment should be put in the acknowledgement, thus the timestamp of the first packet is lost. The per packet RTT measurement becomes more noisy.

In traditional TCP this does not affect ECN marks, but in DCTCP which relies on accurate ECN information delayed acks make the ECN information less accurate. This will make sense shortly as both ECN marks and DCTCP will be discussed.

3.2.2 Delay detection

Round-trip time (RTT) is a measure of how long it takes a packet to reach the receiver and for the associated acknowledgment to return the sender. This is commonly referred to as delay. It is measured by comparing the time a packet is sent and the time the acknowledgement for that packet is received.

One-way delay is a measure of how long it takes a data packet to reach the receiver. Generally it is the one-way delay we are interested in because it better reflects the load in the forwarding path and the bottleneck. The reverse path can add additional noise. Protocols for measuring one-way delay are currently hard to deploy.

Delay has two components: fixed delay and variable delay. The fixed delay is commonly referred to as the base delay, and it is insensitive to congestion. Variable delay is mainly queueing delay which relates to the load on the network. Congestion causes longer queues which results in higher delay. By measuring changes in delay one can detect congestion.

TCP Vegas uses delay measurements to indicate congestion [5]. It works very well when it competes with other flows using TCP Vegas, but it is outcompeted by loss based congestion control algorithms.

Individual RTT measurements are usually subject to noise in end-systems and in the network. Therefore, it is common to apply a function to RTT measurements. SRTT and RTTVAR both use exponential weighted moving averages (EWMA) [35].

3.2.3 AQM

To induce loss traditional TCP has to fill the bottleneck queue. This leads to increased delay, and the magnitude depends on the size of the queue. Large queues requires more packets which creates more delay. *Bufferbloat* is a term used to describe a situation in which uneccesarily large buffers are used [34].

An Active Queue Manager (AQM) is an algorithm that controls how the queue behaves when it has to queue packets. The main purpose of an AQM is to reduce queueing delay. It does so by dropping or marking packets as a queue starts to build, which fools TCP into responding as if the queue were full. There is often randomness involved in deciding which packets will be dropped early.

There are numerous types of AQMs. We will only look at an AQM called Random Early Detection (RED), because it is the AQM used in DCTCP [4].

3.2.4 ECN

Explicit Congestion Notification is a mechanism that allows routers in the network to explicitly signal congestion through an IP header field [14]. The IP header (v4 & v6) has two bits for ECN, which gives four codepoints; 00 (Not-ECT), 01 (ECT(1)), 10 (ECT(0)) and 11 (CE). ECT stands for ECN-Capable Transport and ECT(0) and ECT(1) are broadly equivalent. They both indicate that the endpoint uses ECN. CE stands for Congestion Experienced and is set by the routers experiencing congestion, but only if ECT(0) or ECT(1) is set. If ECT is not set a packet that should be marked is dropped instead.

The use of ECN is negotiated at the TCP layer during the 3WHS. The standard semantics of an ECN signal is the same as a loss, thus the sender has to react in the same way. To ensure that the sender reacts standard TCP has to deliver the ECN signal reliably. This is achieved through acknowledgements. The receiver puts ECN feedback in the acknowledgements to the sender until it receives an acknowledgement for the ECN feedback from the sender. This limits the number of signal per round-trip time to 1.

ECN improves congestion detection and overall efficiency. It is an explicit signal which means that end-systems can be confident that there is in fact congestion and react appropriatly. ECN removes the need for packet loss to signal congstion, which can reduce the number of retransmissions. Note that we use the word *reduces* because there are other causes of packet loss.

ECN-deployment has increased over the past years, and [41] reports that the majority of the top million web servers support ECN-negotiation.

3.3 Prerequisite Concepts

TCP congestion control has two principles: conservation of packets (steady state) and seeking capacity.

Jacobson introduced the principle of conservation of packets. It is that a new packet is not put into the network before another has exited the network. This behaviour leads to what we call 'self clocking' or 'ack clock'. A sender sends data at the same rate as acknowledgments arrives. If we assume that the arrival rate of the acknowledgments reflects the departure time from the bottleneck queue this property makes sure that the sender does not send at a higher rate than the bottleneck can handle. The TCP congestion control algorithm is built on this principle, and requires all TCP flows in the network to obey it. A system that keeps to the conservation of packets principle should be robust if congestion occurs. Jacobson notes that the conservation of packets principle makes Lyapunov stability applicable to the system.

The other principle is seeking capacity, which enables the sender to adapt to increased capacity.

These principles are the basis for the two core algorithms in TCP Congestion Control: Slow start and Congestion Avoidance. In slow start the sender tries to rapidly determine the capacity. It starts by sending a small amount of data, and increases the amount until it detects congestion. This is an example of an open loop control. The increase mechanism is not based on feedback from the system. We will go into more detail in section 3.4.

Congestion avoidance goal is to keep the cwnd as close to the capacity as possible. It has two functions: reacting to congestion and seeking capacity. Reacting to congestion is a closed loop mechanism. Probing for more capacity has traditionally been done by additive increase. This does not depend on feedback, and is thus a open loop mechanism. We will have a closer look at congestion avoidance in section 3.5.

In further discussion we assume that the sending rate is limited by the congestion window, and neither the flow control window nor the application. The receiving application processes the data at a higher or equal rate than the network can supply, and the sending application produces data faster than the network can send the data.

3.3.1 Terminology

Before we jump into the algorithms we will go through some terminology. The maximum transmission unit (MTU) is the maximum size of a packet traversing a path. The maximum segment size (MSS) is the maximum size of a TCP segment that can fit in the MTU of a path. MTU depends on the type of network and configurations, but we will assume a MTU of 1500 bytes which is the MTU of Ethernet. MSS is MTU minus the size of the IP and TCP headers.

I will use cwnd and W to refer to the congestion window. The intial value of cwnd is Initial Window (IW). To be able to switch between

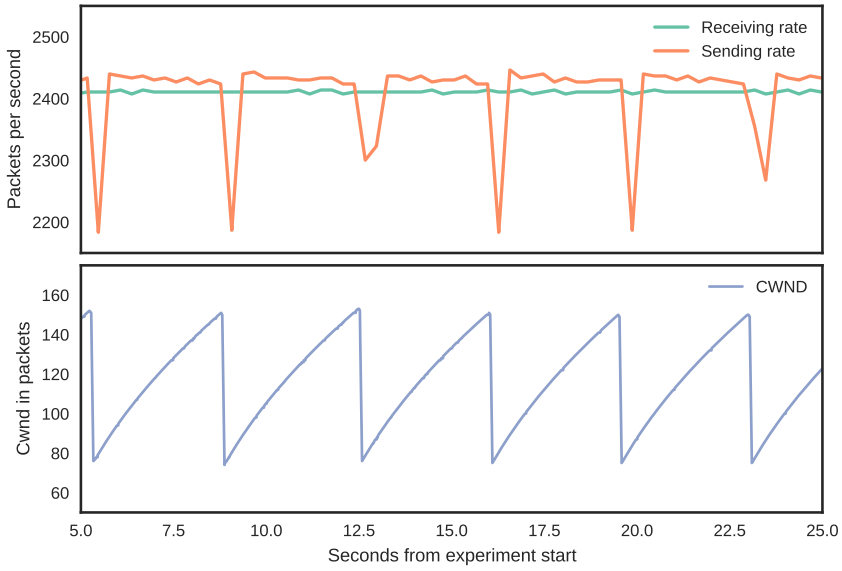


Figure 3.1: Shows sending and reception rate in packets per second in top plot, and the congestion window in the bottom plot. Flow uses TCP Reno. Queue is set to 1 BDP. Note that these values are from tcpprobe and tcpdump, so there might be a small alignment error. The bucket size is 10 RTTs.

slow start and congestion avoidance a variable called slow start threshold (ssthresh) is maintained. If the current cwnd is greater than ssthresh congestion avoidance is used. If cwnd is less than ssthresh slow start is used. If cwnd is equal to ssthresh the implementation can choose either slow start or congestion avoidance [27]. We will be specifying cwnd and ssthresh in packets, which is what Linux uses. The other option is to use bytes.

Bandwidth delay product (BDP) is the amount of data that can be in-flight on a path. It is calculated as follows: $\text{BDP} = \text{capacity} * \text{RTT}$. The BDP will also be specified in packets.

3.3.2 Window Control

A window is an amount of data. In flow control it is the amount of data the receiver currently allows the sender to send. In congestion control it is the amount of data the sender can have unacknowledged in the network. In both cases the window is decoupled from time. It does not specify at which rate the sender is allowed to send the window. If the sender gets feedback that causes the window to grow the sender is allowed to send the whole growth at once. Even a small window can result in a very large rate, because rate depends on time.

I mentioned earlier that congestion window can be thought of loosely as the rate at which the sender is allowed to send at. In fig. 3.1 we have plotted sending rate, receive rate and the congestion window of a flow

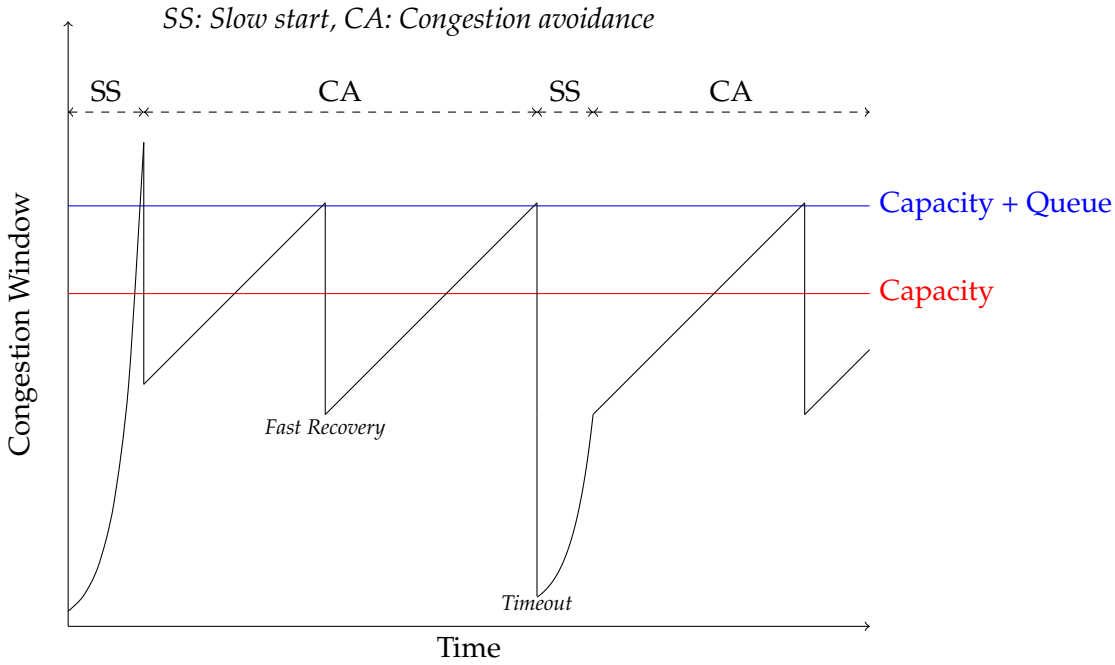


Figure 3.2: TCP congestion control plot

using TCP Reno. The bottleneck uses tail drop and holds one BDP of packets. The rates are calculated with a 10 RTTs bucket size. The sending rate decreases right after each congestion event, but the receiving rate remains fairly constant. The reason is that the queue is large enough to keep the bottleneck busy during the reduction in cwnd and sending rate.

Another approach is rate based control. The sender has a rate at which it can send. It does not specify how much data the sender can have unacknowledged in the network. It does not rely on the ack-clock and it will therefor not have bursts of packets due to noise in the ack-stream. A downside of a rate-based approach is that it continues to send data even though it gets no response from the network. If there is a major congestion incident the sender will continue sending data until it determines that there has been congestion. A window-based approach needs feedback to send more data, so its damage potential is limited.

3.4 Slow start

Slow start is a capacity seeking algorithm with the goal of quickly determining the capacity of the network. It also starts the ack clock. Normally slow start exits on the first congestion event. Exeptions are preconfigured or cached ssthresh lower than the capacity, and slow start after a timeout during congestion avoidance. Slow start has to break the packet conservation principle because there are no packets in the network.

Originally cwnd started with an IW of 1. I will come back to IW in section 3.9. For each received acknowledgments cwnd is increased by

1. This is an exponential increase, the window is doubled roughly every round-trip time. With IW set to 1 the cwnd will be 2^k after RTT_k, where $k \in [0, \text{inf})$.

TCP keeps track of the amount of data it has in the network in the variable FlightSize. Note that FlightSize is the same as cwnd if transmission is limited by the congestion control and not flow control. When a congestion event occurs the following updates are performed [27]:

$$\text{ssthresh} \leftarrow \max\left(\text{FlightSize}/2, 2\right)$$

$$\text{cwnd} \leftarrow \begin{cases} \text{ssthresh} + 3, & \text{Fast Recovery} \\ 1, & \text{Timeout} \end{cases}$$

Fast recovery is by far the most common case today. Fast recovery is a mechanism that allows the sender to continue sending data at a reasonable rate while one or multiple losses are repaired [27].

In fig. 3.2 slow start is shown from the start of the connection and in the middle of the connection. It is denoted by SS. We can see that it ramps up quite quickly and exceeds the capacity and queue which triggers the switch to congestion avoidance. After some time there is another loss, and slow start is again used, but this time it exits at the ssthresh value.

Overshoot

The exponential increase and lag between a packet being lost and the sender detecting the loss causes a situation we call queue overshoot. Queue overshoot is the situation where the sender sends much more data than the bottleneck queue can handle. This can cause a huge amount of loss, long recovery time, and increased delay.

Exponential increase doubles the cwnd every round-trip time. This leads to a doubling of the load on the network. If the cwnd in one round is exactly the amount the network can handle the amount will become twice that which the network can handle the next round.

In the lag between a packet loss occuring and that loss being detected the cwnd continutes to grow which leads to even more congestion.

The overshoot problem becomes more severe as the bottleneck queue size increases. Since the bottleneck queue size is normally set to the BDP, paths through the network with high capacity and long delay suffer more than paths with low capacity and low delay.

Impact on bottleneck queue

In this subsection we will have a look at how slow start affects the queue occupancy. Jacobson writes that the short-term queue demand on the gateway increases exponentially and opening of a window of size W packets will require W/2 packets of buffer capacity at the bottleneck. This holds if the sender's capacity is at least twice the capacity of the bottleneck. We assume that the ack-clock reflects the bottleneck rate. We are ignoring situations such as ack-compression.

If the sender does not increase cwnd at all there would be no impact on the queue because the ack-clock reflects the rate of the bottleneck. When the sender uses exponential increase and the senders capacity is at least twice the capacity of the bottleneck the sender will send at double the bottleneck rate. Thus, every additional packet results in a increase in queue length of 1.

In regular TCP slow start exits at the first loss. Let us assume that the bottleneck uses tail-drop and has a capacity of 1 BDP. The sender would have to fill the queue and send one additional packet to make it drop one. Subsequent packets will result in dup-ack from the receiver, which will eventually arrive at the sender. There is delay from the packet being lost to the sender receiving the first dup-ack, and the cwnd will continue increasing during this whole period. When the first packet that will be lost is sent from the sender the cwnd is $\text{BDP} * 2 + 1$. That means that there is $\text{BDP} * 2$ packets ahead of the first packet that will trigger the first dup-ack. Every one of these will make the cwnd grow by one. We have the following:

$$\begin{aligned}\text{cwnd at first dup-ack} &= \text{cwnd when first lost packet is sent} \\ &\quad + \text{Packets in flight} \\ &= \text{BDP} * 2 + 1 \\ &\quad + \text{BDP} * 2 \\ &= 4 * \text{BDP} + 1\end{aligned}$$

That is an overshoot of twice the capacity, and since the network can only hold $2 * \text{BDP}$ number of packets there will be a huge amount of loss and additional latency.

3.5 Congestion avoidance

The goal of Congestion avoidance is to keep the flow in equilibrium, or put another way keep cwnd as close to the available capacity as possible. Congestion avoidance has two functions; seeking available capacity and reacting to congestion. We often use the term Additive Increase Multiplicative Decrease, AIMD, to describe these functions.

Additive increase describes the way congestion avoidance seeks capacity. The cwnd is increased by 1 each RTT. Additive increase is an open loop mechanism because it operates without any feedback from the system. Since the cwnd is assumed to be close to the available capacity exponential increase as in slow start is too aggressive. The increase could have been 2 or 0.5. For some networks an increase of 1 is too aggressive while in others it is too slow. A constant additive increase is not scalable.

Multiplicative decrease describes the reaction to congestion events. Section 3.4 describes the updates that are made to ssthresh and cwnd.

3.6 Fairness

A property that is important to TCP congestion control is fair allocation of resources between competing flows. One way to look at fairness is using flow rate fairness. If there are N flows in the network each flow should get $\frac{\text{Capacity}}{N}$. This value is commonly referred to as the 'fair share'.

Fairness is a complex and much debated topic. Flow rate fariness is torn apart in [23]. Other fairness allocations are min-max and proportional fairness [8].

3.7 Applicability

Congestion control is not only mandatory in all TCP implementations, but also protocols that use UDP [57].

It is widely used because it works well, and is essential to prevent congestion collapse.

Without congestion control the Internet could not work with the amount of traffic we have today. Not all applications needs congestion control, because their bandwidth usage is far less than the capacity, e.g. VoIP.

Congestion control is usefully for seeking capacity. It is useful even if there is only one user.

3.8 Critique

In congestion avoidance the cwnd is increased by 1 each RTT to detect additional available capacity. But why is the increase 1? Jacobson has a fairly vague justification for choosing 1. He also says that it is certainly too large. TCP Cubic replaces the additive increase with a cubic function to improve detection of additional capacity [26]. However, this does not change the underlying problem which is that the additive increase value is a constant. The increase value would seem to need to be a function of the BDP. In some network the BDP is very low and 1 might be too high, while in other networks 1 might be too low.

The choice of 0.5 as decrease factor upon loss has a valid justification in slow start, but in congestion avoidance it is unjustified. The combination of additive increase and multiplicative decrease is important for convergence.Jacobson says that "being conservative at high traffic intensities is probably wise", and that the cost of large performance penalty is negligible. This is definitely not the case today. Having a decrease factor of 0.5 causes deep buffers to keep the utilization high.

3.9 Enhancements Improving TCP startup

In this section I will go through some relevant enhancements and additions to the slow start algorithm proposed by Jacobson.

Initial window

The capacity in today's Internet is much greater than it was in 1988. The majority of today's paths in the Internet can handle more than 1 packet in the first RTT.

Several proposals have been made over the years to increase the initial window. The current standard is IW of 3 [27]. However at the time of writing IW10 has become the *de facto* standard [40].

Actually, for many years app developers have bypassed the IW limitation by opening multiple connections. This effectively increases the IW. With the recent HTTP/2 this approach is replaced by one connection with several sub-streams [47]. One might wonder if this will lead to increased pressure to either increase IW further or change TCP to be more aggressive in slow start.

Slow start after idle

Not all flows send data at a constant rate, and there might be idle periods where no data is put into the network. Some, such as HTTP DASH streaming, send bursts of data. In these scenarios cwnd and ssthresh might become outdated and noisy. Using this information is dangerous. First, the network conditions can change during the idle period. Second, the packet-conservation principle is violated. Sending a burst of packets filling a large window can severely overload the network. On the other hand restarting slow start can be inefficient.

The TCP standard [27] defines a restart window, RW, and states that when TCP has not received a segment for one RTO the cwnd should be reduced to the RW. $\mathrm{RW} = \min(\mathrm{IW}, \mathrm{cwnd})$. This effectively results in a complete restart which might be inefficient.

This issue is addressed in the experimental RFC2861 [11]. It introduces a algorithm called congestion window validation, CWV. If a flow does not send any data the cwnd should be reduced by half every RTT. If a flow sends some data, but not enough to use the whole cwnd, the cwnd should be reduced to the midpoint between the cwnd and the maximum amount of cwnd used each RTT. In the latter case the flow is said to be application-limited. Ssthresh is decayed to 3/4 cwnd before cwnd reduction.

Experimental RFC7661 obsoleted RFC2861 [49]. It says that CWV is too conservative for many common rate-limited applications. The new algorithm is called New-CWV. It introduces two phases, validated and non-validate, which reflects whether or not cwnd reflects the current available capacity. In a non-validated phase the flow uses less than half the cwnd. After a Non-validated period in a non-validated phase, NVP, the cwnd and ssthresh are reduced as in CWV. New-CWV aims to treat all rate-limited traffic uniformly, and not distinguish between idle and rate-limited.

In section 5 in the New-CWV RFC it says: "A period of five minutes was chosen for this NVP. This is a compromise that was larger than the idle intervals of common applications but not sufficiently larger than the period for which the capacity of an Internet path may commonly be regarded as

stable."

Five minutes is an awfully long time. If RTT is 200ms 5 minutes is 1500 RTTs. To expect that the capacity cannot change quite drastically over the durations of 1500 RTTs is naive.

TCP Fast Open

TCP Fast Open, TFO, is an enhancement that reduces the delay from initiating the connection to sending the first data [44]. It requires the endpoints to have had a prior connection, thus the first connection will not benefit from TFO. During the 3 way handshake the client can ask for a Fast Open Cookie which can be used for subsequent connections.

The client sends a Fast Open Cookie and some data with the SYN to the server. The server can then authenticate the sender and accept the data if the cookie is valid, saving a round-trip time. The server can respond to the data before the handshake is completed.

One issue is that TFO requires application layer idempotency because of duplicated SYN messages. Otherwise duplicated or resent syn-messages can lead to a command being executed several times.

Saving of state

State between two endpoints can be saved/cached to improve the performance of subsequent connections. An example is the ssthresh which can prevent initial overshoot of the bottleneck queue. Saving of state can also lead to sub-optimal performance if the shared information has become stale, e.g. if ssthresh is too low.

Informational RFC2140 [7] and draft [58] discusses sharing of TCP state.

3.10 Related work

3.10.1 Network Assisted Congestion Control

Standard TCP congestion control treats the network as a dumb service. The network should have a better understanding of the load on the network. This begs the question; can the end-system get help from the network? There are a couple of efforts that either require the network to act a certain way or give explicit feedback. We will look at some solutions.

The issue with all these solutions is that none of them are incrementally deployable. In short, every participating entity in the network has to be aware of and use the solution. Any none-participating node can cause severe performance degradation for itself or the other nodes.

eXplicit Congestion Protocol - XCP

XCP is a protocol described in [15] that uses feedback from the network to avoid congestion and at the same time improve utilization of high speed links. Senders put their cwnd and RTT in a header in every packet.

Routers then indicate whether the cwnd should be increased or decreased by annotating the header.

Every node has to be updated with the solution logic.

RC3

RC3, described in [46] is a approach that uses priority levels and network support alongside regular TCP congestion control. It tries to fill the idle periods, where TCP does not send any packets, with low priority packets. These packets are then dropped by routers if congestion occurs, otherwise they are delivered as regular packets. This allows the sender to quickly ramp up its sending rate without incurring congestion to existing flows.

Every node has to be upgraded for this approach to work, if not the congested router might be unaware of the protocol and end up flooded.

RCP

Rate Control Protocol, RCP, is an approach where routers tell senders what rate they can send at [20]. Each packet carries a header of the current rate. If the rate is higher than the router can support a new rate is put in the header. The new rate is then acked back to the sender which adjusts its rate accordingly.

Anti-ECN

Anti-ECN is an approach that allows a TCP connection to aggressively increase its sending rate [17]. It uses a bit set by the routers to indicate to a flow that it is under-utilized.

VCP

Variable-structure congestion control protocol, VCP, is similar to XCP but it uses only ECN bits [21]. Routers uses the two ECN bits to tell the sender what level of congestion it experiences. The sender then reacts according to what level it receives.

Quick-start

Quick-start uses an IP-layer option in the TCP packets [24]. Senders put their desired rate in the header, and routers along the way can approve, reduce or not approve the rate. It can detect routers that do not understand the protocol, in which case standard congestion control is used. However, this is only at the IP layer. It can not detect hosts that does not understand the protocol.

Chapter 4

Data Center TCP

In 2010 Alizadeh et al. proposed a TCP algorithm called Data Center TCP [31]. Data Center TCP is a modification of TCP that aims to minimize queueing delay in the network without limiting the throughput of long-lived flows. It is motivated by the needs in a data-center environment where there is a mix between delay-sensitive and throughput-limited flows. In regular TCP high throughput and low delay can not coexist, because the conservative reaction to loss requires queuing to keep the throughput high. Lowering the queue size would lower the maximum delay, but at the cost of lower utilization. The core idea in DCTCP is to react to the extent of congestion, rather than react to the presence of congestion. This way it can make small adjustments to the cwnd.

DCTCP makes two changes to classical TCP, and one change to the network infrastructure. Switches and routers have to change their queue discipline to provide correct signals to the receiver. DCTCP uses ECN as the primary congestion signal. It is the same wire protocol, but changes the behaviour of the sender and receiver. The two changes to Classical TCP is how congestion signals are sent back to the sender from the receiver, and how the sender reacts to the signals.

Let us first have a look at the changes to the infrastructure before we look at the sender and receiver changes.

4.1 Network change

DCTCP uses an AQM that can be configured using hardware that supports RED with a step-threshold K, where K is specified in bytes. If the length of the queue is greater than K, packets are marked, otherwise packets are not marked. To achieve 100% utilization the marking threshold has to be equal or greater than $0.17 * \text{BDP}$ [33]. Utilization is fairly insensitive to the value of K. There is a slight trade-off between throughput and delay; a small marking threshold can be used without severely degrading the throughput of DCTCP.

4.2 End-system changes

Communicating ECN signals

In DCTCP the receiver tries to signal each CE-mark back to the sender. It does not require acknowledgements from the sender. Delayed acknowledgement complicated the sending of CE marking back to the sender. If every packet creates an acknowledgement the receiver can simply copy the CE mark from the received packet onto the ack. However, pure acknowledgements are not delivered reliably. Also delayed acknowledgement combines multiple packet into one acknowledgement. A normal configuration is to have one ack for every two packets. DCTCP uses a simple state-machine with two states to deal with delayed acks. DCTCP sends immediate acks of the most recent CE-codepoint when the CE-codepoint between subsequent delayed acks changes. If the CE-codepoint is received multiple times in a row delayed acks are sent as usual with feedback of the CE-codepoint.

Sender side reaction to ECN marks

The final change is to the way the sender reacts to congestion signals. Since the signals are all sent back to the sender it can compute the fraction, F, of packets marked with CE. This allows the sender to reacts to the *extent* of congestion, rather than the presence of congestion. It maintains an estimate of the fraction of marked packets, denoted by α. Each RTT the sender calculates F and updates α according to the following EWMA.

$$\alpha \leftarrow \alpha * (1 - g) + F * g,$$

where g is the weight. The default value for g is $\frac{1}{16}$. The estimate of the fraction is then used to update the cwnd if any packets were marked.

$$\text{cwnd} \leftarrow \text{cwnd} * (1 - \frac{\alpha}{2})$$

If the extent of the congestion, α, is low the cwnd is only slightly reduced. When every packet is marked the cwnd is halved as in classical TCP.

A larger value of g causes more more oscillation which can in turn lead to queue underutilization [33].

4.3 Applicability

Data Center TCP can only be used safely in environments where one coordinated administrative entity controls the whole network and all the host operating systems.

DCTCP starves other loss-based TCP congestion control algorithms because they react differently to ECN signals. This explains why DCTCP can not be deployed in the Internet right now. There are several entities with different interests, and different congestion controls.

DCTCP and similar variants are called scalable TCP, while traditional loss-based variants are called classic TCP. Scalable TCP variants get the same amount of congestion signals per RTT in steady state regardless of the capacity [51]. In traditional TCP the time between congestion events increases as the capacity increases.

Coupled DualQ is an AQM proposal that deals with the unfairness between DCTCP and loss-based TCP congestion controls [56]. DualQ has two queues, one for scalable and one for classic. The drop-probability in the classic queue is used as input to the marking threshold in the scalable queue. The coupling ensures congestion window fairness between scalable and classic flows, while the separation preserves the benefits of having a scalable variant.

4.4 Critique

I will not focus on DCTCP's congestion avoidance phase as this is not relevant to my project.

4.4.1 Slow start

Data center TCP does not change how slow start behaves; slow start exits when the first congestion signal is received. We will come back to this in section 6.2.

4.4.2 Feedback protocol

DCTCP is limited to one ECN signal per acknowledgement. When delayed acks are used, which is common today, the receiver cannot accurately reproduce the received stream of ECN marks. DCTCP uses a state machine to deal with this problem. It is considered good enough for most scenarios. Accurate ECN is a draft that proposes a solution that makes ECN feedback more accurate [54]. It uses fields and flags in the TCP header to convey ECN information, and its use is negotiated in the 3WHS. Both endpoints have to support it.

Loss of acknowledgements, and thus ECN marks, is not handled. Losing a few ECN markings should not lead to poor performance because ECN markings are frequent in most situations.

4.4.3 Connection Establishment

"DCTCP provides no mechanism for negotiating its use."[53]. DCTCP has to be deployed on all communicating nodes in the network. In a data center this is feasible, but in the Internet it is impossible. Work is being done on making ECN-based congestion controls incrementally deployable in the Internet. [51]

TCP SYN and SYN/ACK are not ECN-capable due to various concerns at the time, mostly security-related [14]. The issue is that if control packets without ECN-capabilities arrives at a filled queue the packet has to be

dropped rather than marked. Combined with long timeouts for these packets flow establishment might take very long time. In situations with very high marking rate, i.e when there are many flows, the probability of establishing a new connection might become close to zero

There is an experimental RFC for adding support for ECT in SYN and SYN/ACK packets called ECN+ [30]. There is also ongoing work called ECN++ which rebuts all the concerns about ECT on SYNs and SYN/ACKS and allows ECT on the SYN if Accurate ECN is used [52].

Chapter 5

Hybrid Slow Start

We will first look at the motivation for hybrid slow start, hystart, before we discuss how it works in section 5.3. In section 5.2 we will have a look at some theory which hystart is based on because it is relevant as background information.

5.1 Motivation

The main motivation for hystart is to prevent slow start overshoot discussed in section 3.4.

With fast retransmit/fast recovery and SACK the overshoot is normally recovered quite fast, but at a cost. First, the lost data has to be retransmitted which consumes more capacity. Second, it increases the processing overhead in networks where the BDP is large [28]. Third, increased delay as result of filling the queue for a longer than necessary period of time, delaying both the flow itself and others.

5.2 Capacity estimate

Capacity (C) is a measure of the total capacity of a link. Available capacity (A) is a measure of spare link capacity. If a link is utilized:

$$A = C * (1 - U),$$

where $U \in (0, 1)$. A path's available capacity is the smallest available capacity of all the links.

Dovrolis et al wrote a paper called "Packet-dispersion techniques and a capacity-estimation methodology" which looks at two techniques for estimating the capacity and available bandwidth of a path [18].

The first technique is called packet-pair and it tries to determine the capacity of a path. It does so by estimating the bandwidth based on the inter-arrival times of two packets. It can use this information to calculate the bandwidth, b, with a simple amount/time formula. More formally the dispersion of two packets at link i, δ_i, is the time between when the first packet has been transmitted and the time when the last packet has been

transmitted. δ_H is the dispersion at the receiver. The dispersion at the receiver is the time between the arrival of the first and second packet. The receiver can then calculate an estimate of the capacity with $b = \frac{L}{\delta_H}$, where L is the packet size. If the link is unused the estimate will be accurate as proven in the paper. If however the link is utilized traffic can interfere with the packets, causing the estimate to be wrong. Traffic from different sources is called cross-traffic. Cross-traffic can cause underestimation.

The second technique is called packet-train. It extends the packet-pair technique to use N packet, where $N > 2$. The packets in a packet train are sent back-to-back as a burst. The bandwidth can then be calculated using the following formula.

$$b(N) = \frac{L(N-1)}{\delta_H}$$

Intuitively using more packets should decrease the variance of the estimated bandwidth $b(N)$. It does, but $b(N)$ no longer estimates the capacity. When N increases so does the interference from cross-traffic. The distribution of $b(N)$ is defined as $\beta(N)$. Devrolis found that the mean of $\beta(N)$, Average Dispersion Rate (ADR), is what the packet-train estimates.

$$A < \text{ADR} < C$$

ADR is determined by all the links on the path, and not just the one with the least available bandwidth.

5.3 Hybrid slow start operation

Hybrid slow start is an enhancement of regular slow start that tries to exit slow start before it overshoots. Hystart uses two methods to determine when it should exit. The first method uses change in the minimum measured RTT in subsequent round-trip times. The second method is based on the packet-train technique and it uses the total inter-arrival time of acknowledgements to determine if it is approaching the capacity.

Increase in RTT is detected by sampling the smallest measured RTT among the first 8 packets in a packet train and comparing it to the smallest RTT of the previous packet train. If $\text{RTT}_k \geq \text{RTT}_{k-1} + \phi$, where k denotes the RTT number and ϕ is set to $\max(2, [\text{lastRTT}/16])$, hystart exits slow start. An increase in the minimum queue length signals a persistent queue, and ϕ allows for some non-persisten queuing and noise. The choice of looking at only the first 8 packets is that the rest are subject to self induced congestion.

The bursty behaviour of TCP slow start makes packet train measurement possible without introducing extra packets or controlling the sender behaviour. To avoid having to change the receiver the sender measures the dispersion of the returning acks. The assumption is that the dispersion of the acks is equal or greater than the dispersion of the packets, so the estimate will be equal to or lower than the capacity.

Hystart uses the sum of the inter-arrival times of the acks, δ_{cwnd}, to determine when the forwarding path is filled. The forwarding path delay

is half the RTT if the forwarding path and reverse path are symmetrical. Lets call this delay T_f. If $\delta_{\text{cwnd}} \geq T_f$ then the forwarding path is assumed to be full and slow start exits. To deal with noise inter-arrival times of acks greater than 2 ms are not accepted.

5.4 Critique

Hybrid slow start aims at reducing overshoot, but does not address undershoot. If the network link is highly utilized and it has little spare capacity hystart might undershoot its fair share. It acts as a scavenger, taking what is left. This might be good for the overall utilization, however the overall utility will not be optimal. A new flow should put some pressure on the existing flows to converge faster. This can only be done by introducing congestion signals.

Transient traffic can cause temporary queue that hybrid slow start might take as congestion because of increased RTT. Hystart does not address the issues caused by noise and variability.

RTT threshold

Hybrid slow start exits if the minimum measured RTT in a window is greater than lastRTT + max(2, lastRTT/16), where lastRTT is the minimum RTT of the previous window. The second part of the threshold is an arbitrary heuristic. It does not allow for detection of a standing queue less than 2ms. If the capacity is 1Gbps each packet contributes 12 microseconds to the queue. It takes more than 166 packets to increase the RTT by 2ms at 1Gbps.

5.5 Applicability

Hystart can be used everywhere because it does not make slow start more disruptive to existing flows. In some cases it makes slow start less disruptive to others. The only downside of hystart is that it can make slow start exit too early because of the unreliability of the time measurement.

It is part of TCP Cubic and is enabled by default in Linux 4.4. Hystart is frequent disabled in production settings because it has a reputation for exiting slow start early (private communication).

In a DCTCP context where the queue is very shallow slow start exits before hystart can react. Hystart requires a fair amount of buffering to work.

Part II

Design and Development

Chapter 6

Detailed problem statement

This chapter presents and discusses the issues related to slow start. We will start off by looking at slow start in general, and then move on to discussing slows start in the context of DCTCP.

6.1 Slow start in general

A new TCP flows starts without any information about the network that it is going to send its data through. It does not know the capacity of the bottleneck or the current utilization. It starts with a three way handshake from which it can get an estimate of the RTT. With the use of TFO data transmission can start before the 3WHS has completed. In which case the sender only has a last know RTT estimate which might be stale.

To get information about the network the new TCP flow has to probe the network by sending packets into it. The algorithm used to determine the number of packets to send is called slow start. It gradually increases the number of packets that can be sent into the network.

Slow start is terminated, and the new flows transition to congestion avoidance, when the TCP congestion control encounters a congestion indicator. Here is a list of commonly used indicators:

- Loss
- ECN
- Delay measurement

All of these indicators will be discussed in more detail in section 6.3.

Network conditions can change quite frequently. The bottleneck can change. New flows can start and existing flows can terminate. These event can occur during slow start as well. So information can become outdated. Acting on outdated information can lead to severe congestion or under-utilization.

Transient traffic is traffic that lasts for a small period of time. Transient traffic can cause small periods of congestion that can result in congestion

indicators. Reacting to such congestion indicators can lead to under-utilization. Slow start must react to congestion indicators because it does not know the cause. Loss has to be treated as congestion.

Slow start's most known issue is the queue overshoot previously discussed in section 3.4.

There is a coupling between the offered load to the network and seeking capacity. To find the capacity the amount of data sent has to be increased.

6.1.1 Scalability

Scalability refers to a system's ability to handle an increasing amount of work, and its potential to add resources to handle increased load. In the context of slow start it refers to slow starts ability and required buffering to reach the capacity as the capacity increases.

The number of RTTs exponential increase needs to reach a certain congestion window can be calculated using the following formula:

$$\text{Number of RTTs} = \log_2 W - \log_2 \text{IW},$$

where W is the congestion window and IW is the initial window. It takes 4 RTTs to reach a congestion window of 160 packets with a IW of 10. To reach a window of 1280 packets takes 7 RTTs, nearly twice as long compared with the previous example. The number of RTTs needed to reach a certain capacity increases by 1 for every doubling of the capacity.

In addition slow start need the amount of buffering to increase linearly with the BDP.

Larger exponential base

Changing the exponential base from 2 to for example 4 would greatly improve acceleration, but it would cause larger overshoots. There is a trade-off between scalability and causing minimal overshoot. There does not exists an exponential base value that fit all network capacities. Exponential increase with base 2 has been in use ever since first TCP congestion control and has worked well.

Initial Window

A higher IW would improve the performance of slow start. The drawback of increasing the IW is that it causes a larger initial queue if the packets are sent as a burst. This increases the chance of causing loss, which degrades performance.

The current IW in the Linux kernel is 10. This has been shown to increase loss rates [40].

6.2 Slow start in DCTCP

In this section we will discuss slow start in the context of DCTCP.

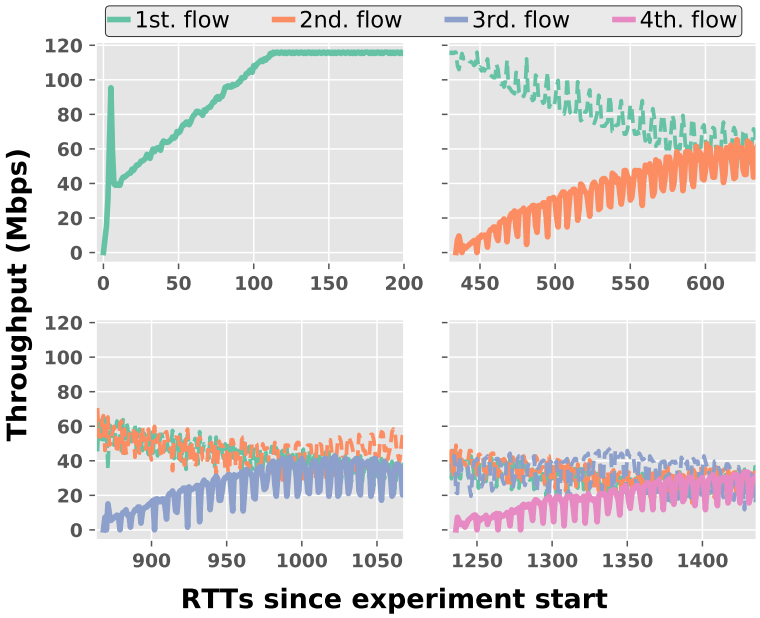


Figure 6.1: Plot showing the throughput of 4 DCTCP flows starting after one another. The marking threshold is 0.17*BDP. The RTT is 15ms and the capacity is 120Mbps. The first flow spends a long time reaching the capacity after it terminates slow start. The other flows spend a long time converging to their steady state throughput.

6.2.1 Undershoot and slow convergence

Undershoot is a situation where a joining flow ends slow start at a much lower rate than it would have had in steady state. Another way of saying this is that it terminated at a rate much lower than its fair share.

DCTCP has the undershoot issue for two reasons. First, slow start exits when it receives the first ECN signal. Second, ECN signals are frequent by design. DCTCP is designed to use a stream of ECN signals to detect congestion. It is wrong to act on only one ECN-signal in DCTCP.

Figure 6.1 shows the throughput of four DCTCP flows as they start. The first flow starts in an empty network, while the other three starts in a fully utilized network. The marking threshold is set to the recommended value 0.17 * BDP [33]. The network parameters are 120Mbit bandwidth and 15ms RTT.

The first flow spends a long time in congestion avoidance right after slow start before it reaches the capacity. This is both an overshoot and an undershoot. It is an overshoot of the marking threshold as we will see in section 6.2.2, and an undershoot of the capacity. The bursty behavior causes the threshold to be exceeded before the capacity has been reached, and initial alpha set to 1 makes the flow halve its rate as a reaction to the first ECN mark. We will discuss initial alpha shortly.

It takes more than 100 RTTs for the second flow to reach its fair share. As the number of existing flows increases the time it takes to reach the fair

share decreases. Convergence time worsens as capacity increases because of the constant additive increase in congestion avoidance.

Initial Alpha

Figure 6.1 also shows another important inefficiency of DCTCPs slow start. At the start of the first flow we see that the rate increases rapidly, however upon the first congestion signal the throughput plummets before growing at a steady rate until it reaches maximum throughput again. This is because alpha is initially set to 1 which means that cwnd is halved at the first mark. In traditional TCP the rate stays close to the maximum rate when it halves its cwnd because the queue is normally longer. A longer queue allows the cwnd to grow larger and thus end up at a higher value when the first congestion indicator appears. A longer queue also has more packets to send while the sender is backing off. However, in DCTCP the queue is only 0.17 BDP, which means that the first ECN marks are triggered early and the queue has fewer packets to send during the back off.

Critique of slow convergence justification

DCTCPs convergence rate is at most 40% slower than TCPs according to [33]. Undershoot is a known trade-off between convergence time and steady state performance of other flows made by the authors of DCTCP [31]. They posit that convergence is not a major concern in data centers. They make it clear that DCTCP is designed for data centers with specific workflow characteristics. They do not discuss the fact that the characteristics might change in the future. There might become a need for medium sized flows that need faster convergence.

The discussion uses time as a measurement, but we strongly believe that number of RTTs is a better measurement of convergence time. Time can give a false impression of the magnitude if presented without the base RTT. A convergence time of 40ms is 2 RTTs when the base delay is 20ms, but 100 RTTs if the base delay is 400 microseconds.

Responsibility

DCTCP reacts to CE-marks every RTT. The reaction is controlled by alpha. Alpha is updated with the most recent marking fraction, F, every round trip time. The update is an EWMA with smoothing factor $g = 1/16$. This means that it takes several RTTs for a DCTCP flow to react to a significant increase in the number of marks.

A new flow has to persistently cause CE-marks over several RTTs to make the existing flow(s) yield sufficient bandwidth. If DCTCP could react more quickly in congestion avoidance a new flow could gain sufficient bandwidth in fewer RTTs.

It is not straightforward to change the smoothing factor because it affects the steady state behaviour. A small value such as $g = 1/2$ would lead to oscillation and slightly poorer utilization. Combining a lower

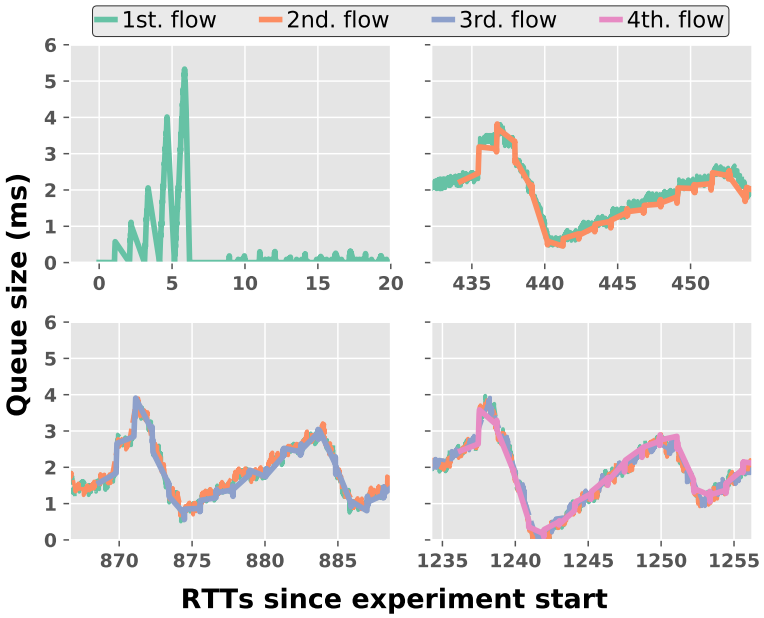


Figure 6.2: Plot showing the queue delay measured in microseconds during in the start-up phase of 4 DCTCP flows. The marking threshold is 0.17*BDP. The RTT is 15ms and the capacity is 120Mbps. The first flow causes spikes of queueing delay reaching roughly double the marking threshold. The other flows causes additional queueing delay the first two RTTs and then they exit slow start.

smoothing factor and a ramp instead of a step threshold could keep high utilization and improve reaction time. We did some initial investigation which showed instability, but did not pursue it any further. This has to be investigated.

Nevertheless, the point is that the existing flows are responsible for yielding capacity when a new flow joins the network. Even if existing flows yield more capacity to a new flow that flow has to claim it. This will require changes to slow starts response to a single ECN mark.

6.2.2 Queue impact

Figure 6.2 shows how the queue size behaves when four DCTCP flows are started after one another. fig. 6.1 shows the throughput for the same four flows. The first flow exhibits bursty behavior resulting in spikes of queue delay before it gets its first congestion signal and terminates. The marking threshold is roughly 2.5ms, and the first flow reaches an instantaneous queue length around 5ms which is roughly twice the marking threshold.

A flow joining one of more existing flows usually causes a small increase in delay because it sends a burst of 10 packets into the network. Since new flows exit at the first signal they usually do not cause spikes after the two first RTTs.

6.2.3 Lower marking threshold

In the future we might want to have a marking threshold less than the recommended 0.17 * BDP to further reduce latency. Regular slow start performs less well as one reduces the marking threshold because it will terminate sooner and at a lower rate. This is a consequence of the bursty behavior and exiting on the first CE-mark.

A lower marking threshold has implications for the steady state performance. The utilization becomes slightly poorer as the marking threshold gets lower because the queue has less data to send while the sender reacts to CE-marks.

6.3 Congestion Indicators

In this section I will discuss different signals and measurements, and how they can be used in slow start.

6.3.1 ECN marks

ECN marks are reliable signals of congestion because they are explicitly set by network devices. If a packet carries an ECN mark it can be assumed that there is congestion. In the Internet the sender does not know which AQMs are used so it cannot assume that the packet has exceeded a marking threshold. In a data center environment, however, an ECN marked packet can be assumed to have exceeded a marking threshold. Misconfigurations can happen, but we will assume that all network devices are configured correctly.

A single ECN mark conveys only a tiny piece of information; At the time the packet went through the queue the queue was filled above the threshold. ECN is a single bit feedback, often wrongly referred to as binary feedback. It is a unary feedback. To get a better understanding of the state of the network multiple consecutive ECN marks have to be used. DCTCP steady state is built around a stream of unary feedback.

The step marking in DCTCP leads to a pattern we will refer to as on-off. Packets are marked in bursts and there are periods between these bursts without any ECN marks. In a single round-trip it is likely that either all or none of the packets are marked. This is the reason alpha is updated using a smoothing factor.

The on-off pattern is even more prominent in slow start because of its bursty behavior. This makes it hard to use ECN signals. Currently, slow start exits on the first ECN marks. This goes against the fundamental design choice of using a stream of unary feedback.

Using ECN in slow start

The current reaction to the first ECN mark in slow start causes undershoot. Can we use ECN marks in a different way?

Because of the on-off behavior of ECN marks it is difficult to use them to infer information such as the number of flows and the current congestion level. If 100% of a window of packets is marked it signals that there is congestion, but not how much congestion. One can say this is severe congestion, but in the context of DCTCP it is normal. It occurs with regular intervals. To infer information from ECN signal a flow needs to collect marks over several congestion events. By the time it has the information it needs it might have already converged.

Moving the smoothing from the end-system to the network might make ECN signals usable in slow start. This could be done by using a linear marking probability, a ramp, or a proportional integral (PI) controller.It would be able to detect the extent of the congestion rather than the presence withing a few RTTs.

A ramp can also allow existing flows to react more quickly because it might remove the need for smoothing of the ECN signal. The issue with having a ramp is that it is hard to configure it correctly. Some initial investigation showed that changing the marking behaviour would require a whole thesis of new work. Therefor we have put this outside the scope of this thesis.

One idea we have been exploring is to use the known marking pattern of DCTCP in steady state to detect when a new flow has converged to its steady state cwnd. For example if we know that there should be two marks per RTT on average in steady state we could look for this pattern and exit once we come close to it. With the on-off marking behaviour and the slow smoothing such an approach would require a large amount of RTTs to work though.

An issue with using ECN in slow start in the Internet is that the bottleneck might not be ECN-capable. Even if you receive ECN marks during slow start the bottleneck might move to a different network device without support for ECN. Therefor, ECN cannot be used as the primary signal, but it could provide extra precision.

6.3.2 Loss

Regular TCP has to treat loss as severe congestion, even though there are several non-congestion related causes for loss. It follows that loss is not as reliable a congestion signal as ECN.

In a network that uses and supports ECN in all network devices a loss should be preceded by one or more ECN marks. If a loss occurs before receiving any ECN signals it is highly plausible that the loss is unrelated to congestion in the network. This can be used to justify ignoring loss that are not preceded by ECN marks.

On the other hand loss also signals that there are issues in the network or end-systems or both, unrelated to the queue and the marking threshold. Therefore, to be conservative, loss should be treated as congestion.

6.3.3 Delay Measurements

An advantage delay measurement has over ECN is that it can be used to infer how long the queue is when it is over the marking threshold.

Delay measurements are highly variable so multiple measures are needed to get useful information. Taking multiple measures consumes time which delays convergence

The precision of delay measurements affects their usefulness and reliability. Particularly in a data center environment RTT measurements are highly susceptible to noise which in turn can make a delay based congestion control overreact [31]. In recent years time measurement has become more accurate, due to new hardware and higher resolution timer support in kernels. TIMELY [50] is allegedly the first delay-based congestion control for data centers. It uses the change in RTT, its gradient, to detect congestion. The authors found that the Linux TCP stack's RTT measurements are not accurate enough to be used reliably. This claim is supported by [50]. Their solution uses RTT measurements provided by the NIC.

The necessary precision increases with capacity if the queue remains constant measured in bytes. A queue of 10 packets is 1.2 milliseconds in a network with 100Mbps, but only 12 microseconds at 10Gbps. It gets harder to detect congestion using delay measurements at higher link rates. However, if the size of the queue measured in time remains the same with increased capacity, the necessary precision does not change. In this case it is the queues length relative to the RTT that matters.

TCP timestamps [43] improve precision of RTT measurements, but it currently has only a 1ms granularity. Google has microsecond timestamps since Feb 2015 [1]. It is yet to be pushed upstream to the Linux kernel. Work on negotiation the TCP timestamp precision has previously been conducted [39, 42].

[1] https://www.ietf.org/proceedings/97/slides/slides-97-tcpm-tcp-options-for-low-latency-00.pdf

Chapter 7

Naive approach

In this chapter we will go through work done prior to working on the main solution, Paced Chirping. We start off by looking at pacing in section 7.1; How it might be used and issues. We show how pacing at a constant rate gives not information at all until the rate is greater than the capacity which causes a huge queue. Then we present the first solution attempt in section 7.2.

7.1 Pacing

Pacing is a technique that can be used to eliminate bursts of packets by limiting their sending rate. It is done by adding delay between subsequent packets. Pacing can eliminate the bursty behaviour in slow start. Intuitively this should remove the undershoot problem when the flow is alone, and possibly reduce the queueing delay.

Pacing has been on by default since Linux kernel version 4.4[1]. It requires a FQ qdisc to be attached to the outgoing interface, or for the congestion control to explicitly request internal pacing (Since version 4.13). Since this is enabled by default there should be benefits when using it, right?

7.1.1 Insights from other work

I have dedicated this subsection to discuss the findings in [9]. The paper presents an evaluation of TCP Reno with and without pacing. The pacing rate is set to $\frac{\text{cwnd}}{\text{RTT}}$, which distributes the window over the whole RTT.

We will not look at the steady state finding as this thesis is about slow start.

Scalability and Queue size

Pacing makes slow start more scalable when there is limited buffering; signals arrive after the flow has exceeded the capacity and does not depend on the queue size. When the buffer is greater than half the BDP pacing

[1] https://unix.stackexchange.com/questions/337456/is-tcp-pacing-enabled-by-default-on-linux

seems to worsen the performance. They tested with a buffer of one quarter BDP.

Synchronization effect

When multiple flows start at the same time pacing suffers from a synchronization effect. The flows does not get any signals before the network is saturated, at which point all the flows gets a signal and backs off. This might make the bottleneck under-utilized.

With regular TCP Reno some flows are likely to drop out earlier than the other flows, before the bottleneck is fully utilized (assuming a shallow buffer is used). However, when pacing is used all the flows get the signal at roughly the same time and they all react to it. Regular slow start has temporary congestion that does not affect all the flows, while pacing makes the congestion visible to all the flows.

7.1.2 DCTCP with and without pacing

We did an experiment to see how pacing affected slow start behaviour of regular DCTCP when the network is unutilized. Flows starting in a utilized network normally exit quickly due to the high marking rate, so I will not discuss pacings effect on these flows here.

As metrics we chose queue delay, and throughput. Qdisc FQ is used for pacing at the servers. The marking threshold is set to 0.17 * BDP. We ran three different RTTs (5,10,15) and three different capacities (60Mbps, 100Mbps, 150Mbps). Each run had three flows starting at 10 second intervals.

The kernel calculation of pacing rate is used. In slow start this is twice the rate if cwnd were spread over the whole RTT. More formally $\frac{\text{cwnd}}{\text{RTT}} * 2$. So when cwnd is half the BDP the pacing rate equals the bottleneck rate. The only exception is the first 10 (IW) packets which are allowed to be sent at line-rate by the FQ qdisc.

Queue impact

In fig. 7.1 the queue length during slow start has been plotted for two network scenarios with and without pacing. The topmost plots are from runs with capacity and RTT set to 60Mbps and 5ms respectivly. The bottommost plots are from runs with capacity and RTT set to 150 Mbps and 15 ms respectivly. The leftmost plots are without pacing, while the rightmost plots are with pacing.

Slow start causes more queueing delay when pacing is used than when it is turned off. The maximum queue length is roughly twice as large with pacing than without pacing. The queue also persists for a longer time.

The Pacing makes it so that the queue does not begin to grow until the cwnd exceeds half the BDP, except for the IW packets. When cwnd reaches half the BDP the pacing rate is roughly the same as the bottleneck rate. Further increase in cwnd will increase the rate and the queue delay. When

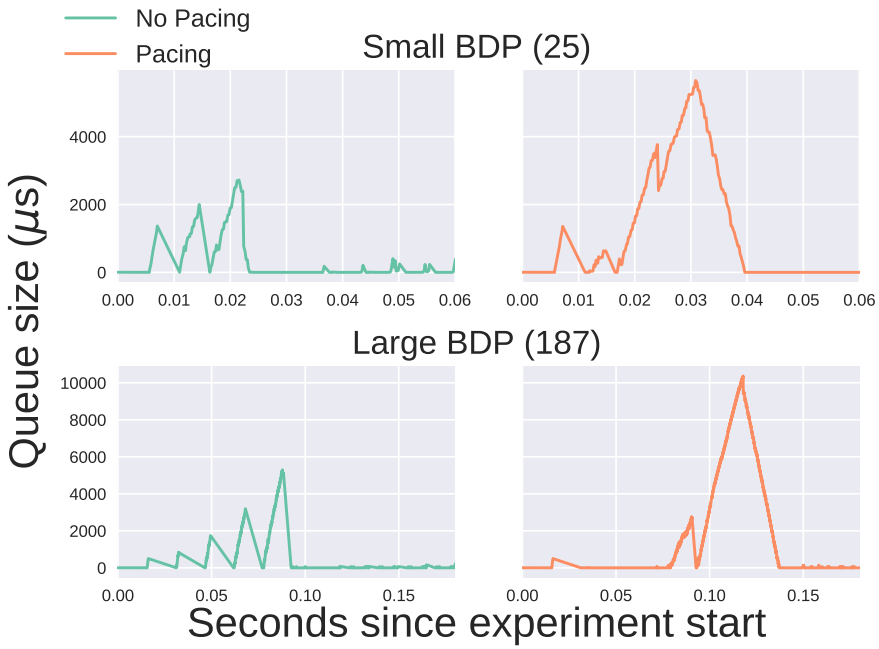


Figure 7.1: Queue delay when flow starts in an empty network when pacing is used (right, in orange) and not used (left, in green). Pacing makes the magnitude and duration of the overshoot much greater.

the marking threshold is exceeded the lag between triggering the mark and receiving that mark makes the congestion worse.

Throughput and Convergence rate

Figure 7.2 shows the throughput of the same scenarios as in fig. 7.1. When pacing is used the flow goes up to the capacity before it exceeds the marking threshold, while the flows without pacing normally exits before the capacity has been reached. It is only when the BDP is significantly greater than IW that pacing improves throughput.

We will now look at convergence time, defined in section 10.3. It is essentially the time it takes a flow to reach its steady state throughput.

Table 7.1 shows the convergence time of a flow starting in an empty network with different BDPs. The red-colored area shows that pacing can worsen the Convergance time at low BDPs.

The two cyan-colored areas shows that pacing can improve convergence time significantly.

7.1.3 Discussion

Pacing does not address undershoot when the network is utilized. We assume that the flow is alone in the following discussion.

Pacing allows slow start to reach the capacity before it starts to build a queue and eventually receive the first mark. This generally improves convergence time, but at the cost of higher queue delay.

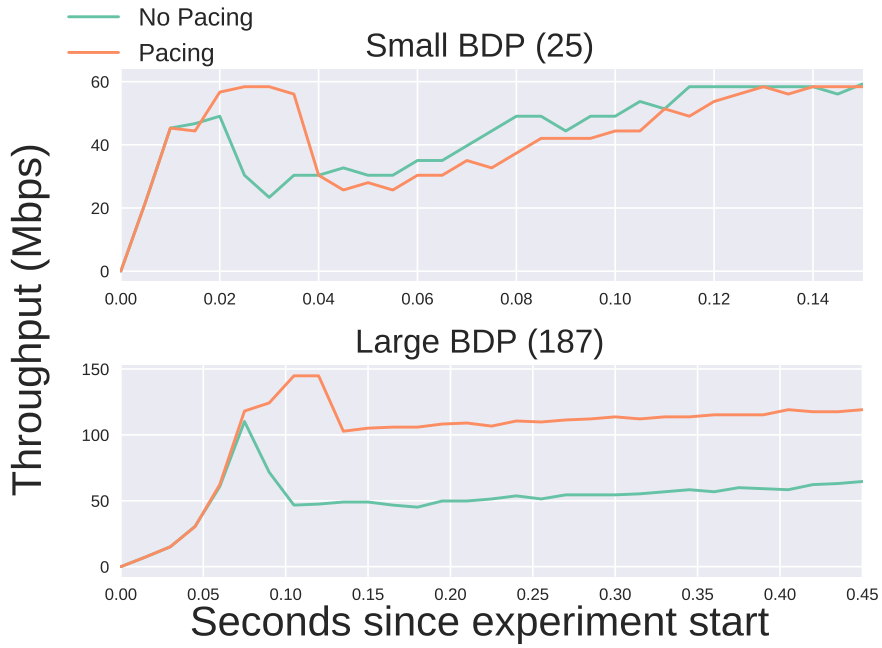


Figure 7.2: Throughput of a flows starting in an empty network with and without pacing. Pacing achieves greater throughput when the capacity increases.

BDP	Pacing	min	max	mean	sd
25	True	22	23	22	0.547723
25	False	17	21	18	1.516575
41	True	16	28	18	5.272571
41	False	31	36	33	1.923538
50	True	30	35	32	2.000000
50	False	30	40	36	5.477226
62	True	21	23	22	0.836660
62	False	41	44	43	1.414214
75	True	24	27	25	1.140175
75	False	47	49	48	0.707107
83	True	23	29	26	2.509980
83	False	52	56	54	1.581139
125	True	26	34	28	3.271085
125	False	65	67	66	0.894427
187	True	31	35	33	1.581139
187	False	108	113	110	2.073644

Table 7.1: Convergence time of DCTCP flow starting in an empty network i.e; the flow is alone. The red-colored area shows that pacing can lessen convergence time when the BDP is low. The cyan-colored areas shows that pacing improves convergence time as the BDP increases.

Pacing allows cwnd to be increased rapidly without causing bursts of packets. This can allow slow start to jump to a certain capacity without having to go through multiple RTTs with exponential increase.

Pacing reduces the effect the marking threshold has on regular slow starts performance because it is exceeded once the flow has reached the capacity. Similar insights can be found in [10].

Avoiding synchronization effect

To avoid the synchronization effect there are mainly two things that can be done. First, randomness can be introduced to the pacing rate to try to de-synchronize the flows. Second, switching between pacing and no pacing can probe the queue while using pacing.

Getting feedback before reaching capacity

The lag in the feedback causes more severe congestion. One way to deal with this could be varying the pacing rate over a period much like chirping. Chirping is presented in Chapter 8. The goal of these periods is to probe the queues ability to handle higher rates.

More reliable congestion indicator

The queue remains roughly empty until the cwnd approaches the BDP. A significant increase in the RTT should indicate congestion at the bottleneck. Pacing allows us to use RTT measurements much more reliably.

Hystart has to use the minimum RTT over a window to detect persistent queue, as discussed in chapter 5. If the flow is alone this means that part of the burst of the previous window has to be in the queue once the next burst arrives at the queue. This requires much more buffer than what is available in DCTCP.

7.2 Initial attempt

Initially we focused on finding a solution that did not require any modifications to the base-kernel.With inspiration from Hystart and packet train estimation we decided to build a solution around capacity estimation.

We moved on to Paced Chirping before we really explored this solution, mainly because we ran into an unresolved issue regarding termination of slow start. This will be discussed in section 7.2.3. Nonetheless, this chapter will present and evaluate what we did and our insights.

7.2.1 Design

A new flow sends two packet trains the first two RTTs. The inter-arrival times of the acknowledgements and the number of bytes they acknowledge are used to estimate the bottleneck capacity. The capacity estimate and the minimum RTT observed is then used to calculate the BDP of the path.

The algorithm has two different modes; Push back mode and prevent overshoot mode. The mode is chosen based on whether or not any CE marks are received during the two first RTTs. Push back mode is used when CE marks have been observed, and prevent overshoot mode is used otherwise. The assumption is that CE marks indicate existing cross-traffic.

In both modes pacing is used to prevent bursts of packet from entering the network. In push back mode it allows the new flow to increase its cwnd without causing excessive queueing due to bursts. In avoid overshoot mode pacing eliminates the bursty behaviour that would otherwise exceed the marking threshold before the capacity is reached.

Initial alpha is set to 0 instead of 1 (default) to reduce the undershoot issue discussed in 6.2.1.A value of 1 results in a halving of the rate upon receiving the first CE-mark which is too conservative. It can be argued that 0 is too aggressive. Its value should really depend on the congestion level right after slow start. A possibility is to use a higher smoothing factor in the start of a flow to quickly get an initial estimate.

Push back mode

The goal of this mode is to acquire capacity from existing flows. The new flow uses the estimated BDP and alpha to determine how much capacity it can acquire.

Each RTT the cwnd is increased proportional to the BDP and alpha, formally:

$$\Delta\text{cwnd} = \frac{\alpha}{2} * \text{BDP}$$

We found that dividing alpha by two reduced the queue impact because the other flows were allowed to react before we pushed in. The idea here is that the existing flows decrease their cwnd by roughly $\alpha * \text{BDP}$, which can be claimed without causing excessive queueing.

The number of flows is unknown and we found no good and reliable way of detecting it within reasonable time, therefore we decided to terminate once the cwnd reaches $b * \text{BDP}$. We have mainly used $b = 0.5$. If the number of existing flows is greater than one a joining flow is likely to claim more than its fair share.

Prevent overshoot mode

The goal of this mode is to prevent the new flow from overshooting the capacity. It is very simple; ssthresh is set to the estimated BDP and exponential increase continues until cwnd reaches ssthresh or a CE-mark is received. At which point slow start is exited and congestion avoidance takes over.

Estimating Capacity

The final estimate is the average of the estimates from the two packet trains.

One can argue that the the two estimates should have different weights seeing that they have different sizes and different average sending rates, and that the first is more stale. The first train is sent at line-rate. The second train can be twice as long as the first, and its average sending rate is twice that of the ack-rate.

If the first train is weighted the most the final estimate would be closer to the total capacity because the first train is the least likely to have cross-traffic between its packets of the two trains. The second train is closer to the available capacity than the first if there is cross-traffic.

Pacing

We set ssthresh to the estimated BDP. The default pacing rate calculation in the kernel sets the pacing rate to $\frac{\text{cwnd}}{\text{RTT}} * 1.2$ when cwnd $\geq \frac{\text{ssthresh}}{2}$. This allows cwnd to grow larger before it exceeds the capacity, assuming that the ssthresh estimate is close to the actual BDP. In Linux kernel 4.10 pacing is implemented in the FQ qdisc. To allow the two first packet trains to be sent as bursts the initial_quantum parameter of FQ qdisc is set to 50000 bytes, which is a little over 30 packets of 1500 bytes.

Note that pacing prevents further capacity estimates based on inter-arrival time of acks. The use of FQ qdisc for pacing makes RTT measurements unusable because the time packets wait in FQ qdisc becomes part of the estimates. This is the main reasons why RTT was not considered a good signal for this initial solution.

Issues

In this section we list the known design issues of this initial solution.

- Does not handle utilization change during slow start. If all other flows exit the new flow is stuck in push back mode because it does not get any marks. If other flows join during slow start it does not adapt. We were thinking about moving between the two modes before finding a more promising direction.
- Parallel flows. Pacing actually makes the overshoot worse if there are multiple flows doing slow start at the same time.
- Wrong estimate. If the estimate is too high the combination of exponential increase and pacing make the overshoot issue worse. To combat this issue we have been thinking of using RTT-measurements as a congestion signal. Unfortunately, we did this implementation using FQ qdisc for pacing. If the estimate is too low the new flow undershoots.
- Initial bursts can exceed the marking threshold of an empty network which makes the flow enter and stay in push back mode due to the lack of additional marks.
- Very disruptive in networks with small BDP. The two initial bursts of 10 and 20 packets can be very disruptive.

7.2.2 Implementation

Implementation was a struggle because the Linux kernel does not allow you to not react to ECN marks. It enters congestion window reduction upon seeing a CE-mark and does not allow you to send more packets until you get an acknowledgment without a CE-mark. We had to constantly (on reception of every ack) overwrite cwnd and ssthresh to try to deal with this problem.

The inter-arrival time of acknowledgements is measured using a kernel clock. The number of bytes acked is supplied by the kernel and it is necessary to handle variable sized packets. The estimated capacity is then calculated by dividing the number of bytes acked with the total inter-arrival time. Note that the first acks number of bytes are not included.

We will not go into more details because the solution was scrapped.

7.2.3 Evaluation and Discussion

In this section we will present and evaluate the initial solution. We used three bandwidths (60Mbps, 100Mbps, 120Mbps) and three RTTs (5ms, 10ms, 15ms). Each run has 4 flows started 7 seconds apart, and a total duration of 25 seconds. Each combination of bandwidth and RTT is repeated 5 times. There are 9 configurations, and each configuration is run 5 times.

We did not test the solution extensively because we moved on to paced chirping knowing that this solution had many issues.

Convergence and Queue impact

Figure 7.3 shows throughput and queue delay for plain DCTCP (left) and our modified DCTCP (right). The upper row shows throughput, and the second row shows queue delay right before and 60 RTTs after each of the flows start times. the initial solution correctly identifies the capacity when the first flow starts which results in much higher throughput after slow start and lower queue delay. Since plain DCTCP halves its rate the queue delay remains very low until it reaches the capacity.

The flows starting in a utilized network gets a fairly good estimate of the capacity. This is because the initial burst of packets are sent at line rate and the second at twice the capacity (averaged over the send period. Pairs are sent at line rate.). This leaves little room for other traffic to come in-between the packets. Since the flows increase their rate based on their estimate and their alpha value they manage to claim capacity without causing significant rise in queue delay.

Figure 7.4 shows the normalized queue length against convergence time for plain DCTCP and the initial solution (Modified DCTCP). It is calculated over the first 8 RTTs. We will look at the first flow first. We can see that the queueing delay is significantly less for our modified slow start than for plain DCTCP. This suggests that the initial solution can reduce

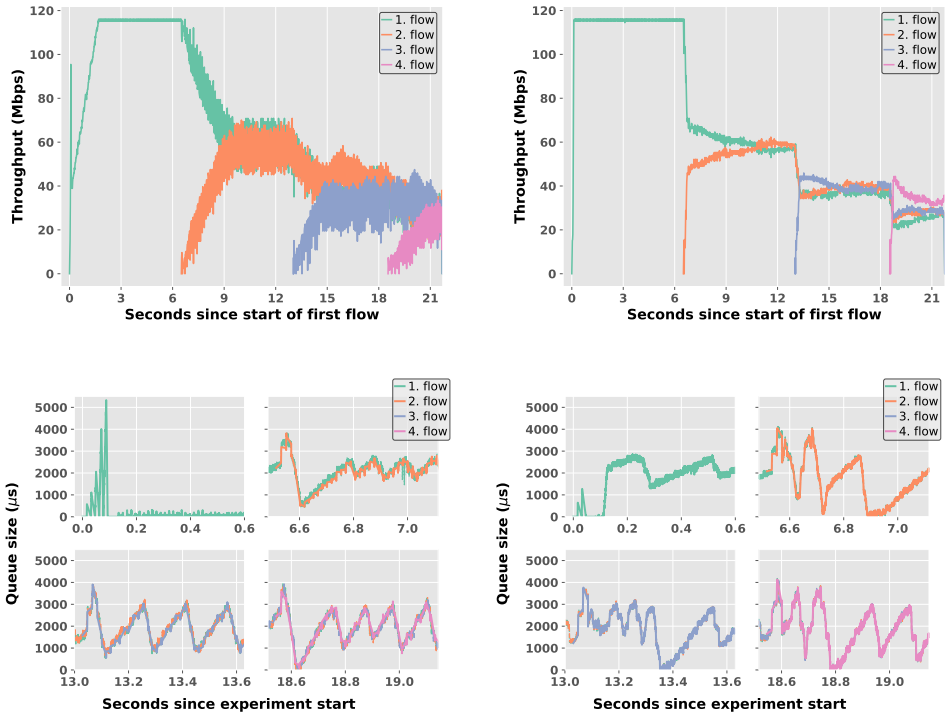
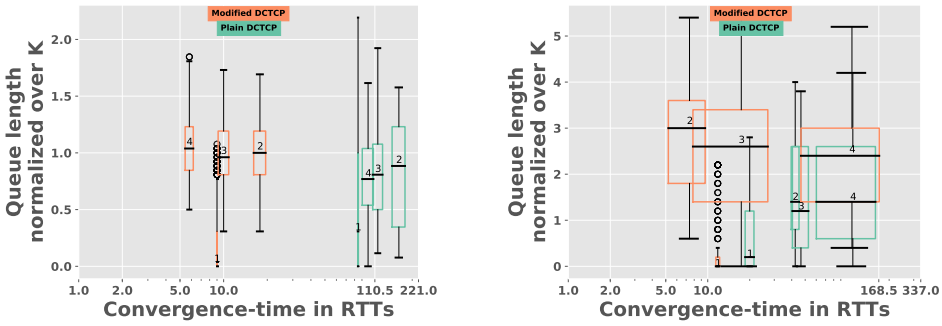


Figure 7.3: Throughput and queue delay for plain DCTCP (on the left) and the initial solution (on the right). The initial solution successfully reduce queue delay and improve convergence time for the first flow starting in an empty network. For subsequent flows the convergence time is improved at the cost of slightly higher and more variable queuing delay.



(a) 15ms, 120Mbps

(b) 5ms, 60Mbps

Figure 7.4: Both plots show queue length over convergence time. It show the trade-off between low convergence time and high delay. modified DCTCP is the initial solution. On the left we can see that the convergence time is greatly improved without significant higher queueing delay. The first flow even has less queueing delay. In the plot to the right shows the same tendency, but the queueing delay is higher.

overshoot. It also shows that since the initial solution does not back off as quickly as plain DCTCP slow start does, the convergence time is improved.

Flows starting in a utilized network (2., 3., 4., etc.) benefit from the added aggression without imposing too much queue delay. It is a trade-off between convergence time and queue delay. The higher the BDP is, the more improvement the initial solution gives.

Estimate Reliability

The solution relies on two estimates of the capacity. If either of theses estimates are wrong the flow can undershoot or overshoot. From this we learned it would have been better to continuously estimate the capacity.

Pacing and packet train estimation cannot be combined when FQ qdisc is used for pacing in Linux v4.10. One has to choose between having pacing and rely on the first two estimates, and having no pacing and getting continuous estimates.

Determining when to terminate

We found no good way to determine when push back mode should be terminated. The number of flows are unknown. The estimated capacity is not noticeably affected by the number of flows because it is based on bursts at line-rate.

Any knowledge of steady state behaviour is hard to use because the networks behaves so differently in slow start.

One possibility is to look at how fast the other flows increase their share of the capacity. We could assume that all the other flows are in congestion avoidance and increase cwnd by 1 each RTT and try to detect the total increase and infer the number of flows from that. But, it is hard to sample and make sure that the behaviour of the new flow does not affect the sampling.

There is no obvious ways for using ECN marks to determine when the new flow has reached its steady state.

7.2.4 Conclusion

Here follows what we learned from this initial naive solution attempt. It is possible to push existing flows back using alpha and BDP estimate without causing unacceptable queueing delay. Capacity estimates from initial packet trains can be very accurate. Pacing successfully removes the bursty behaviour and allows a flow to accelerate to the estimated capacity without exceeding the marking threshold. Pacing allows for a sudden jump in cwnd without creating a burst of packets.

Here follows the main reasons we abandoned the solution. Does not adapt to changing network conditions after the two first RTTs which can lead to underutilization and severe congestion. Relies solely on the initial two estimates. This makes it very suseptible to noise in the network and

the kernel. No good way to determine when to exit slow start. Hard to implement in a kernel that reacts to ECN marks.

Chapter 8

Paced Chirping

In this chapter we will describe and discuss a new flow start algorithm we have called Paced Chirping.

Section 8.1 gives a high level description of Paced Chirping and provides some definitions. Section 8.2 mentions some prior work on chirping. Section 8.3 discusses chirps in greater details. Section 8.4 discusses how the information from a completed chirp is analyzed. Section 8.5 discusses the paced chirping algorithm; how everything is put together. Section 8.6 discusses some of the choices we have made, possible improvements and future work. Section 8.7 lists known issues.

8.1 Paced chirping

A chirp is a sequence of packets sent at increasing rate with a known average rate. This is realized by controlling the inter-send gap between the packets. A chirp can probe for multiple rates without causing excessive queuing. The available capacity can be estimated for each individual chirp by comparing the queuing delay experienced by each packet. An increasing trend in queuing delay is an indication that the packets from the start of the trend has been sent faster than the available capacity.

The concept of *paced* chirping is that multiple chirps are spread out over a RTT. A single chirp can only sample the available capacity of a fraction of the RTT, so by having multiple chirps spread over a RTT the estimate is more likely to detect other traffic. Pacing the chirps also has the benefit of letting the bottleneck drain between chirps, if the estimate is wrong.

We will be using inter-send time, gap and rate to mean the same thing. We assume that each packet has the same size, 1500B. The algorithm itself can be used with different packet sizes.

$$\text{gap} = \frac{\text{Packet Size}}{\text{Rate}}$$

The algorithm maintains an estimate of the lowest average gap it can send at without causing a queue, called gap_avg. By definition this is also an estimate of the available capacity. It also has a variable M that limits the number of chirps it can send in a RTT.

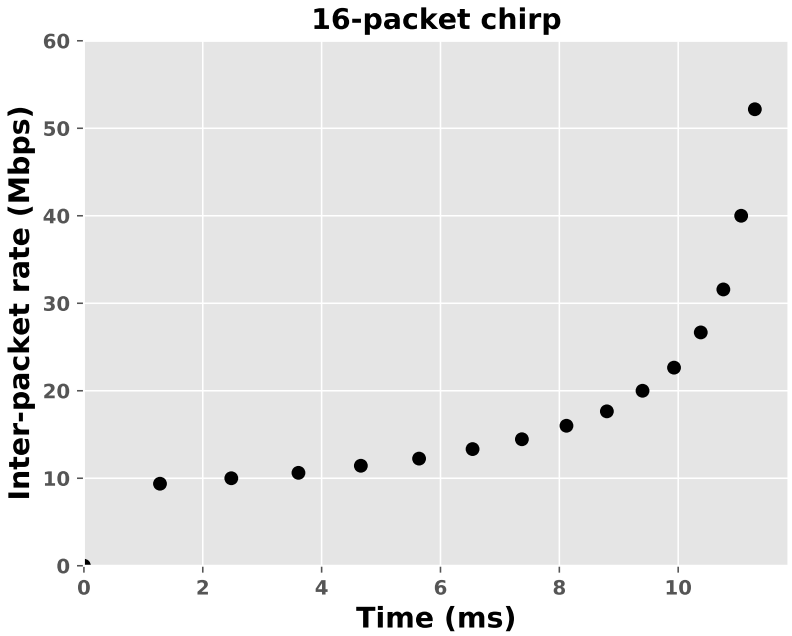


Figure 8.1: Schematic of the rates of the packets in a 16-packet chirp over time.

8.2 Prior work

PathChirp [16] is a tool that estimates available bandwidth by sending and analyzing multiple chirps at different rates. TCP RAPID [29] is a congestion control that uses chirping to adapt its rate. [32] implemented and evaluated TCP RAPID in the Linux kernel. It provides insights in what is needed to implement chirping in the Linux kernel. [59] implemented and evaluated TCP RAPID in a high bandwidth setting.

8.3 Chirp

This section will present a formal definition of a chirp and the relationship between subsequent chirps. To be able to talk about the relationship between two chirps we have to introduce a way to identify a chirp. A chirp is identified by the subscript m, where $m = 1 \ldots \infty$. We will omit the subscript when it is implied that we are talking about a single chirp.

A chirp is a sequence of N packets with an increasing rate (decreasing inter-send gaps). The gap at the sender between packet $i-1$ and i of a chirp is denoted by gap_i, where $i = 1 \ldots N-1$. Section 8.3.1 discusses the calculation of the gaps. Figure 8.1 shows a schematic of a chirp with packet-rate over time.

A chirp has a total duration D which is defined to be the sum of all the gaps, formally:

$$D = \sum_{i=1}^{N-1} \text{gap}_i$$

The time between chirp m and $m+1$ is called a guard interval, and is denote by B_m.

B_m : The time between end of chirp m and start of chirp m+1

$$B_m = \max\left(\text{gap\_avg}, \frac{\text{SRTT}}{M} - D_m\right)$$

B_m should be no smaller than gap_avg so that the average rate measured over subsequent chirps does not exceed the estimated rate.

8.3.1 Calculating gaps

We are using an arithmetic progression for the gaps within a chirp. An arithmetic progression is a sequence of numbers with the following form: $a_i = a_{i-1} + d$, where d is a constant. Our arithmetic progression is motivated by the progression in [32].

It is important that the average rate of the packets in a chirp is as close to the estimated available capacity as possible. Otherwise the chirp might cause more queueing than intended. It is not important that the OS is able to accurately reproduce the gaps, but ideally they should be monotonically decreasing. We will come back to this in section 8.4.2.

We have introduced a parameter called geometry, geometry $\in (1, 2]$, that controls the range of the gaps. By default the geometry is set to 2, which is the function used in [32]. The upper bound can be larger than 2, but then the average gap of the chirp becomes increasingly smaller compared with the target average gap.

The gap before the first packet, gap_0, is determined by the guard interval between the previous chirp and the start of the new chirp. The first gap, gap_1, is set to geometry $*$ gap_avg. The gaps of the other packets is set according to the following function:

$$\text{gap}_i = \text{gap}_{i-1} - (\text{step} * (i-1)),$$

where $i = 2 \,..\, N-1$ and

$$\text{step} = \frac{((\text{geometry} - 1) * 2) * \text{gap\_avg}}{N}$$

These equations give us a linear range from gap_1 to $\text{gap}_1 - (N-2) *$ step, and an average gap a little larger than gap_avg. The calculation of gap_i can be implemented with subtraction operations in a for loop instead of multiplications.[1].

Geometric progression of rates

[29] uses geometric progression for calculating the rates. A geometric sequence is a sequence of numbers with the following form: $a_i = a_{i-1} * k$ where $i \geq 1$ and k is a constant. A geometric progression has the advantage

[1] Code can be found in Appendix A

that it can have an increased difference between gaps, while an arithmetic sequence has a constant difference of d. The main issue with geometrical progression is that it is more computationally expensive. If an integer power of 2 is used for k it is possible to optimize the computation using shift-operations, but we wanted to have it dynamic for experimentation.

8.3.2 Measuring queue delay

The queue delay of a packet is calculated by computing the difference between the minimum RTT and the packet's instantaneous RTT measurement. More formally,

$$q_j = \text{rtt}_j - \text{RTT}_{\text{min}}$$

rtt_j and q_j are the instantaneous RTT measurement and queue delay of packet j. RTT_{min} is the minimum RTT measured over the whole duration of the flow.

8.4 Analyzing a chirp

We use the analysis technique used in PathChirp [16] with a slight change to one constant, described next. We conducted a few tests to verify we had not upset the precision of the algorithm.

The analysis identifies trends of increasing queuing delay, called excursions. An increasing trend is an excursion if there are at least L packets within the trend, where L is a whole number. PathChirp uses L = 5. An excursion is a sign that the rate the packets were sent at was faster than the available bandwidth.

A trend might not be strictly increasing because of cross-traffic and measurement inaccuracy. The analysis accepts a decrease in relative queuing delay in an increasing trend. The parameter F controls how large the decrease can be. PathChirp uses F = 1.5 which means that it accepts a decrease in relative queueing delay, within an excursion, less than $\frac{1}{3}$ of the maximum relative queueing delay. The relative queueing delay of a packet in an excursion is the difference between the packets queueing delay and the queueing delay of the first packet in the excursion.

We use L = 5 and F = 1.6. L = 5 has proved to work fine in the testbed, and is what is used in PathChirp. F is set to 1.6 to make it possible to calculate the threshold using only shift-operations. We have checked through experiments that the difference in accuracy is insignificant.

8.4.1 Effects of changing L and F

The sensitivity of the analysis can be controlled by changing L and F.

L controls how long an increasing trend has to be to be considered an excursion. A higher L makes it harder for a trend to be accepted as an excursion. A smaller L results in more trends being accepted as excursions.

Algorithm 8.1 Pseudo-code for analysis of completed chirp

```
1: N ← Chirp size
2: l ← N − 1
3: max_q ← 0
4: excursion_start ← 0
5: E ← Array[N]                                          ▷ Estimated gaps/rates
6: strikes ← 0
7: lowest_gap_i ← 1
8: for i ← 1 to N-1 do
9:     if send-gaps[j] > send-gaps[lowest_gap_i] * H  then
10:        return INVALID_CHIRP
11:    end if
12:    if send-gaps[j] > send-gaps[lowest_gap_i] then
13:        strikes ← strikes + 1
14:        if strikes >= S then
15:            return INVALID_CHIRP
16:        end if
17:    else
18:        lowest_gap_i ← i
19:    end if
20:    relative_queueing_delay ← q[i] - q[excursion_start]
21:    if excursion_length and relative_queueing_delay > max_q/F then
22:                                                     ▷ Excursion continues
23:        max_q ← max(max_q, relative_queueing_delay)
24:        excursion_length ← excursion_length + 1
25:    else
26:                                                     ▷ Excursion has ended
27:        if excursion_length ≥ L then
28:            for j ← excursion_start to excursion_start + excursion_length − 1 do
29:                if q[j] < q[j+1] then
30:                    E[j] ← send-gaps[j]
31:                end if
32:            end for
33:        end if
34:        excursion_start, max_q, excursion_length ← 0
35:    end if
36:    if !excursion_length and (i < N − 1) and q[i] < q[i+1] then      ▷ Excursion start
37:        excurison_start ← i
38:        max_q← 0
39:        excursion_length ← 1
40:    end if
41: end for
42: if excursion_length and excursion_length + excursion_start = N then
43:                                                     ▷ Unterminated excursion
44:    for j ← excursion_start to N-1 do
45:        E[j] = send-gaps[excursion_start]
46:    end for
47:    l ← excursion_start
48: end if
49: sum ← 0
50: for i ← 1 to N-1 do
51:    if E[i] = 0 then
52:        sum ← sum + E[l]
53:    else
54:        sum ← sum + E[i]
55:    end if
56: end for
      return sum/(N−1)
```

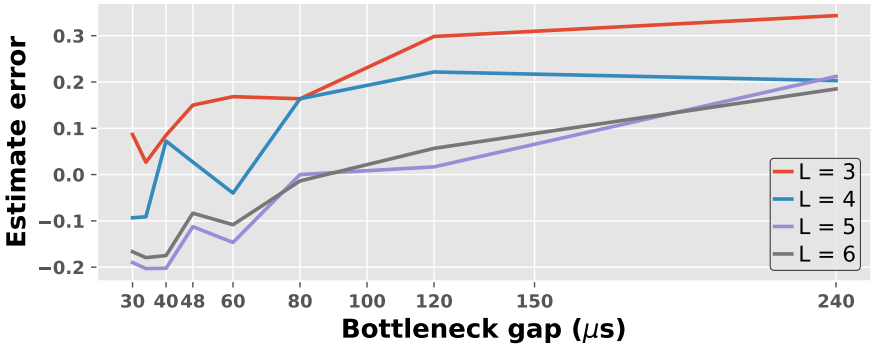


Figure 8.2: The final gap estimate error of paced chirping with varying L. The estimate error is $\frac{\text{estimate}-\text{gap}}{\text{gap}}$.

A higher F makes the analysis accept larger decreases as part of an increasing trend. A lower F makes the analysis more strict, making it harder to accept a decrease in relative queueing delay.

A higher L or lower F or both can lead to underestimation of available capacity because these makes it harder for trends to be accepted as excursions. A lower L or higher F or both can lead to overestimation because these makes it easier for trends to be accepted as excursions.

There are no two values for L and F that fit all types of networks. It depends on the noise characteristics, the capacity of the bottleneck and cross-traffic. So these values might need to be tuned to improve Paced Chirpings performance in the Internet. Figure 8.2 shows the effect of changing L on the accuracy of the final estimated gap with varying capacities. A lower L gives a more conservative estimate because it accepts more trends as excursions.

8.4.2 Discarding a chirp

If the inter-send time in a chirp is too inaccurate we see no other choice but to discard the information of that chirp. We count occurrences of increased inter-send gap as strikes. A chirp is discarded if the number of strikes is greater than or equal to S, where S is a parameter we have set to 5. However, this rule allows for arbitrary high increases in the inter-send time. To deal with this we introduce a new parameter, H, which controls how large the increase can be. H is currently set to 1.25. A chirp is discarded if a gap is greater than H times the smallest previously seen gap. This might be too strict, but we think that it is better to reject false information than insist on using it. Failing to identify excursions leads to overestimation.

We have yet to figure out how to deal with multiple discarded chirps. One option is to fall back to regular slow start. If none of the chirps in one RTT produce new estimates it is probably wise to give up on Paced Chirping.

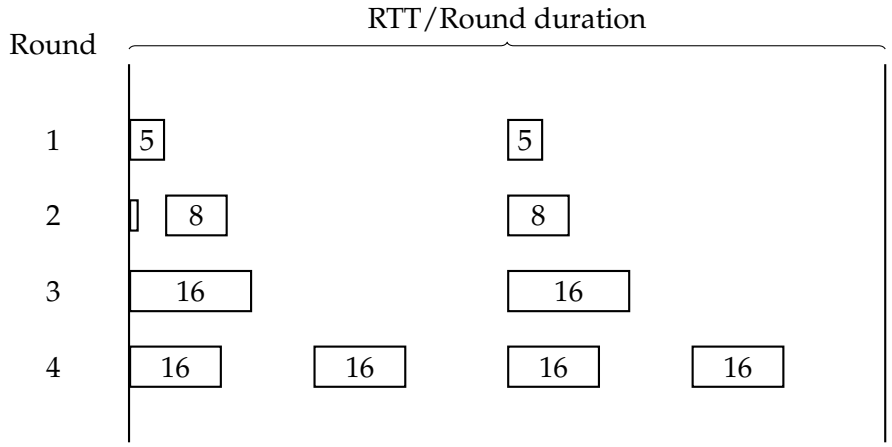


Figure 8.3: Chirping over 4 rounds/RTTs. Round 1 sends two chirps/bursts. Round 2 sends one marker packet and two chirps. The following round sends two 16-packet chirps. From the 4th round the number of chirps is increased by multiplying it by the gain. In this illustration gain is 2. The decrease in the chirp length is a result of a lower estimate.

8.5 Algorithm

Figure 8.3 shows how the algorithm operates in the in the first couple of RTTs. In round 1 we send two bursts to get an initial estimate of the capacity. See section 8.6.1 for discussion of round 1. At the start of round 2 we send a marking packet that is used to start the third round. Then two chirps of 8 packets follow with an average rate estimated from the first bursts. Round 3 starts when the acknowledgment for the marking packet arrives at the sender. At this point it sends two chirps of 16 packets each.

Round 4 shows how the algorithm might behave with a doubling of the number of chirps and a decrease in the duration of the chirps. The change in the number of chirps is controlled by a variable called gain, 2 in this case. The decrease in the duration is caused by a higher estimated rate. If the estimated rate decreases the duration increases.

Figure 8.4 shows the recorded inter-packet gaps at the sender and the receiver during flow start with gain and geometry set to 2. Each vertical sequence of dots is a chirp. The first packet of a chirp is sent at the lowest rate, thus with the highest gap. Then for each subsequent packet the rate is increased which means that the gap is decreased. The gaps at the receiver are bounded by the bottleneck capacity. Paced Chirping exits after roughly 6 and a half RTTs.

8.5.1 Updating gap estimate

Once we have an initial estimate from the first burst we continuously update the estimate with the most recent one.

When all the acknowledgment of a chirp have been received we analyze the chirp to get a new gap estimate, called gap_new. The gap estimate,

Algorithm 8.2 Actions on reception of acknowledgement

```
if In transition period
    if Chirping list empty and
            (in-flight > Estimated BDP − 2 or
            Paced for 1 RTT) then
        Exit slow start
    end if
end if
while Used pacing rates do
    Calculate inter-send time
    Update knowledge about associated chirp
    Remove entry
end while
Calculate inter-arrival time
Calculate queueing delay
Increase number of chirps if possible
Send new chirps if possible
if Current chirp finished
    if First chirp/burst
        Calculate average inter-arrival time
    else
        ANALYZE_CHIRP(Current chirp)
    end if
    Update gap estimate
end if
if Should terminate
    Set pacing rate to estimated rate
    Enter transition period
end if
```

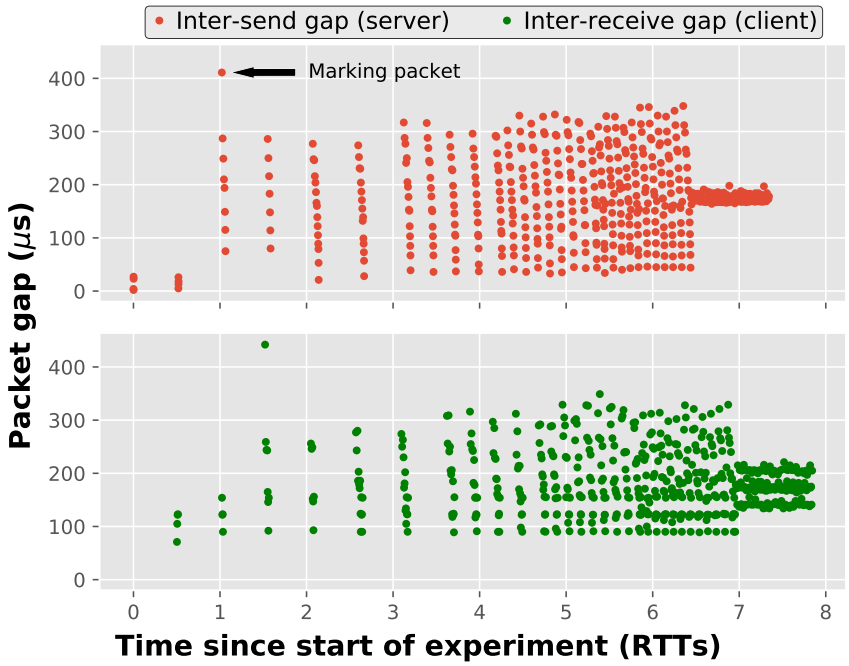


Figure 8.4: Inter-packet gap at sender and receiver when a flow starts in a unutilized network with gain and geometry set to 2.

gap_avg, is adjusted using an exponential weighted moving average with smoothing factor b, currently set to $\frac{1}{2}$.

$$\text{gap\_avg} \leftarrow \text{gap\_avg} * (1 - b) + \text{gap\_new} * b \tag{8.1}$$

The EWMA helps stabilize the estimate and make it more robust to noise in the network and measurements. Having a higher smoothing factor improves responsiveness, but makes the estimate more susceptible to noise. We have tried lower values for b, but we are content with the current value.

8.5.2 Controlling the number of chirps

The gain is a variable that can be adjusted in response to the capacity measurements, but we have set it to a constant for the initial experiments. In Section 11.6 we show a naive attempt on adjusting the gain (and geometry).

Every round the number of chirps is increased proportionally to the gain using the following update function:

$$M \leftarrow M * \text{gain},$$

where M is the number of chirps allowed in a round. When the first acknowledgment of the current round arrives at the sender a new round is started, and M is updated.

M can have a fractional part, but this part is removed when it is used. An example is scheduling new chirps.

8.5.3 New chirp

A new chirp can be scheduled if the number of chirps sent in the current round is less than M. In addition we limit the number of chirps scheduled with the same estimate by allowing only ceil (gain) chirps to be scheduled during the handling of the acks from a chirp.When the gain is 2 every chirp of the previous round can only trigger two new chirps. If the gain is 3 then every chirp can trigger three new chirps.

Note that chirps are scheduled, which means that there might be a lag from scheduling a chirp to when the first packet of that chirp is sent into the network. During this lag the chirp cannot be modified in the current implementation. The reason is that pacing entries are not associated with a chirp, so finding the pacing entries belonging to a particular chirp is difficult, if not impossible.

8.5.4 Termination

Paced chirping is terminated when the sum of the durations of the scheduled chirps and the minimum guard interval exceed the smoothed RTT. Formally,

$$\sum_{m=1}^{\text{num chirps sent}} D_m + \text{gap\_avg}_m$$

A loss event also leads to termination. We have not done experiments with loss so this will have to be investigated in the future.

When we terminate Paced Chirping we want to transition to using the ACK-clock. If there are scheduled chirps that have yet to be sent we need to wait for them to be sent before we enter congestion avoidance. If the average rate of the scheduled chirps is lower than the final estimated rate the network might be underutilized when all the chirps have been sent. We use a transition period to deal with this. We also need to find the correct cwnd for utilizing the estimated rate.

We set the pacing rate to the estimated rate, and calculate the target BDP using the estimated rate and the minimum RTT. Once all scheduled chirps have been sent we continue pacing packets at the estimated rate until either the number of in-flight packets is greater or equal to BDP $-$ 2, or BDP packets have been sent at the pacing rate. At this point we set cwnd and ssthresh to the number of in-flight packets and transition to congestion avoidance.

8.6 Discussion

8.6.1 Round 1 - Bursts or Chirps or both

It is not clear whether we should send two burst, two chirps, or one of each. These options gives different information.

A burst is likely to give an estimate that is close to the total capacity, not the available capacity. It will not be able to detect and react to the utilization of cross-traffic. Bursts do not require any prior knowledge of the network.

A chirp on the other hand is designed to find the available capacity. It can estimate the capacity used by other flows. When a new flow starts in the presence of cross-traffic it is likely that there is very little spare capacity so a chirp will provide a very conservative estimate.

We have observed that an initial chirp tends to choose the largest gap of the chirp, i.e lowest capacity, as the available capacity. This is probably because there is very little available capacity when the new flow joins the network.

When a flow starts it has no information about the capacity of the network, so it has to blindly set a gap_avg for an initial chirp. It can base it on the RTT from the 3WHS, but it does not know the relation between the RTT and the capacity of the bottleneck. Thus, it is difficult to set a range of rates.

A burst has the nice property that it causes additional queueing which might trigger ECN marks for cross-traffic to receive. The existing flows will reduce their utilization and allow the new flow to claim it. A chirp causes very little queueing by design, so it might not push back the existing flows at all.

The combination of a burst and a chirp can give a better picture of the network. A burst gives us the total capacity, while a chirp gives us information about existing flows. The burst will grow a queue and hopefully be able to trigger ECN signals to make the existing flows yield some capacity.

Until we find a good way to set the initial value of gap_avg we use two bursts in round 1.

8.6.2 ECN

Currently Paced Chirping does not use ECN signals. It is not clear how CE-marks should be used; there are many possibilities.

The presence of CE-marks can can be used to adjust the gain or geometry or both. It might also be used to make the chirp analysis more accurate, or find a better termination condition. How ECN signals can and should be used in ECN-based flow start is still an open question.

In addition to the lack of a clear use of ECN one of the goals was to create a flow start algorithm that could handle bottlenecks without ECN support; Either by design or misconfiguration. This is a requirement for a similar algorithm to be deployed in the Internet.

8.6.3 A strong core - Performance optimization is secondary

There are numerous ways the algorithm could be improved to give better performance on a specific set of experiments. The gain and geometry can be fine tuned to get optimal behaviour. ECN can be used to prevent

overshoot and quicker convergence in one scenario with the cost of slower convergence in another scenario.

We have presented Paced Chirping in its purest form. Adding small improvements at this stage is too early. We have tried small improvements, such as not increasing the number of chirps if ECN-marks have been encountered, that have given better performance in certain experiments. However, once the algorithm is presented with such improvements the core idea is kind of locked. We think it is important to understand what we can do with the core algorithm. Can we remove the need for the gain and have continuous updates of the number of chirps based on the capacity estimates? Can CE-marks be used to find out if a new flow have converged with the existing flows? How can ECN be used in the chirp-analysis?

Answering the difficult questions might yield an algorithm that does not need small optimizations to perform well in a wider set of scenarios. Let us try to perfect the core before we trim the rough edges.

8.7 Limitations and known issues

Here are the known issues.

- Does not handle reordering and duplicate acks. This should show up as noise in the analysis. It would be great to be able to match acknowledgment with packet using sequence numbers. This can become computationally expensive if reordering is high.
- Does not handle delayed acks.
- Does sort of handle loss in 3WHS. It does not accept inter-arrival times greater than half the RTT. We discovered a bug in DCTCP; if it is configured with ECN enabled per route per route and the default CC is set to a non-ECN protocol (i.e Cubic) the syn-ack does not carry the ECT(0) bit. This makes the syn-ack eligible for drop instead of marking.
- No good initial gap_avg. Constant is not a good solution. Inter-arrival time of bursts generally measures capacity, not available capacity.
- TCP Fast Open (TFO) is an extension that allows a TCP connection to send data before the 3WHS is completed using a cryptography cookie. This might complicate the implementation as there might be no recent estimate of the RTT when the first data packet is sent.
- Bursty link MACs (Medium Access Controllers) can eliminate the individual rates of the packets in a chirp. Dealing with this situation is future work.
- Termination condition. We have implemented a pragmatic solution to the question of when paced chirping should exit and switch to congestion avoidance. But a more principled solution is still an open issue.

- Have not tested high statistical multiplexed bottlenecks.
- Have yet to integrate ECN. We have suggested some ideas, but it is not clear which direction to take.

Chapter 9

Paced Chirping Implementation

In this chapter we will go through the implementation of Paced chirping. The implementation is divided into two parts. The first is a modification to the internal pacing implementation which is located in the kernel. The second is a loadable TCP CC module. Linux is structured so that code can be added during run-time by loading it as a module. This makes development simple because the kernel does not have to be recompiled every time a change is made.

Paced chirping is implemented as a loadable module in Linux using the DCTCP code as a starting point. All code related to congestion avoidance is preserved. We have not focused on optimizing the code because the algorithm is still under development.

Section 9.1 discusses our changes to the internal pacing implementation in the Linux kernel. Section 9.2 will discuss the changes we have made to the internal pacing implementation. Section 9.3 briefly discusses the TCP CC module implementation.

9.1 Pacing implementation

Linux version 4.13 introduced a TCP pacing implementation internal to the kernel[1]. Prior to this the FQ qdisc kernel module had to be used for pacing.

Internal pacing makes RTT-measurements more accurate as the time packets wait in FQ qdisc is no longer part of it. It also makes it simpler to change the pacing behavior because the code is now located in the TCP stack.

I will not go into detail on how interrupts and scheduling work in the Linux kernel. I recommend the following site: https://notes.shichao.io/lkd/ch8/.

[1] https://patchwork.ozlabs.org/patch/762784/

9.1.1 Internal pacing implementation

Pacing is implemented by inserting time between the sending of consecutive packets. Each packet is sent at the NICs capacity, but the inserted time makes it so that the average rate over multiple packets can be controlled. Internal pacing uses a high-resolution timer, hereby referred to as the pacing timer, to manage the time between packets.

When the TCP stack attempts to send a new packet it checks if the pacing timer is active. An active pacing timer indicated that it is too soon to send a new packet and doing so would violate the set sending rate. If the pacing timer is active the TCP stack aborts the attempt to send a new packet, and tries again later.

If the pacing timer is not active the packet can be sent and the pacing timer reset to a new value.

When the pacing timer expires the kernel receives a hard IRQ (hardware interrupt) and **tcp_pace_kick** is called. This function schedules transmission of the next packet of the associated socket. To be more specific the socket is added to a CPU bound tasklet work queue. It can be delayed between scheduling and transmission of the next packet, and the delay depends on how busy the CPU is and the number of sockets in queue.

The pacing functionality is mainly located in **tcp_output.c**. The pacing timer is located in **struct tcp_sock**. Note that each socket has its own tcp_sock structure and therefor its own pacing timer. Internal pacing is applied only if **sk_pacing_status** in **struct sock** is set to **SK_PACING_NEEDED**.

Figure 9.1 shows a simple schematic of how pacing is applied to outgoing packets. The red parts are changes we have done and will be discussed in section 9.2. **tcp_write_xmit** is the function that sends a new packet. It calls **tcp_pacing_check** to see if the pacing timer is active. If it is the function returns. If it is not active **tcp_transmit_skb** is called. After the packet has been transmitted **tcp_transmit_skb** calls **tcp_internal_pacing** which resets the pacing timer to a value derived from **sk_pacing_rate** in struct sock.

9.1.2 Kernel calculated pacing rate

The kernel updates sk_pacing_rate at each ack-arrival, and when the connection is set up, by a call to **tcp_update_pacing_rate**. There does not seem to be a way for a TCP CC to disable or override the calculation except for implementing the **cong_control** callback for rate-based CCs used and introduced by BBR. The issue with implementing cong_control is that it omits part of the cwnd-oriented TCP stack code.

9.1.3 Turning internal pacing on and off

Internal pacing can be turned on and off by setting sk_pacing_status to SK_PACING_NEEDED. Pacing is also disabled if sk_pacing_rate is set to 0U or ~0U (maximum value).

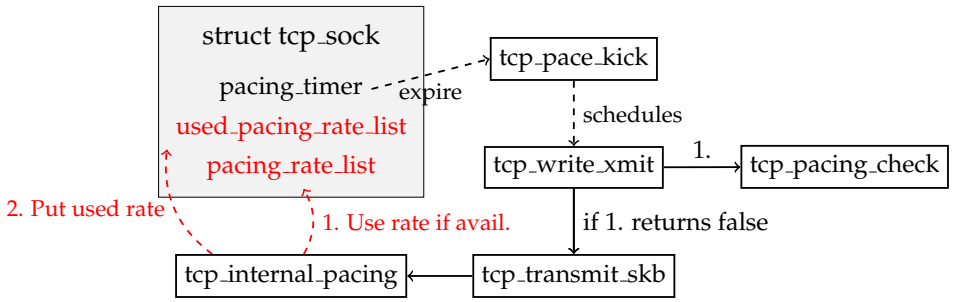


Figure 9.1: Internal pacing implementation with changes in red. When the pacing timer is updated in tcp_internal_pacing the code checks if there are any per-packet rates available.

9.1.4 Issues

The pacing implementation is designed to prevent packets being sent at too high a rate. It is not designed to make sure that packets are sent at a certain rate. The tasklet is scheduled with TASKLET_SOFTIRQ softirq. Another option is the HI_SOFTIRQ softirq that has higher priority. This could improve the latency between scheduling and sending packets.

The pacing implementation does not take delay introduced between the TCP stack and the NIC into consideration. Having the NIC insert the gap between packet might improve the performance, but on the other hand if the bottleneck is in the stack or between the stack and the NIC it might pose problems.

If FQ qdisc is used at the outgoing interface it will disable internal pacing and do the pacing itself. FQ qdisc has a check that sets sk_pacing_status to SK_PACING_FQ if it is set to anything else. So, Paced Chirping can not be used with FQ qdisc at the outgoing interface.

9.2 Kernel modifications

In this section I will present, justify, and discuss the modifications we have done in the Linux kernel version 4.13. A patch for the kernel modifications can be found online, see Appendix A.

9.2.1 Disable existing pacing rate calculation

During the transition period we need to have full control over the pacing rate. With the current rate calculation in the kernel this is not possible. Therefor we decided to add a variable to struct tcp_sock called **disable_kernel_pacing_calculation**. If it is set to 0 the kernel calculates the pacing rate as usual, but If it is set to 1 the kernel does not calculate and update the pacing rate. This gives all TCP CC modules the option to calculate the pacing rate themselves or disable pacing.

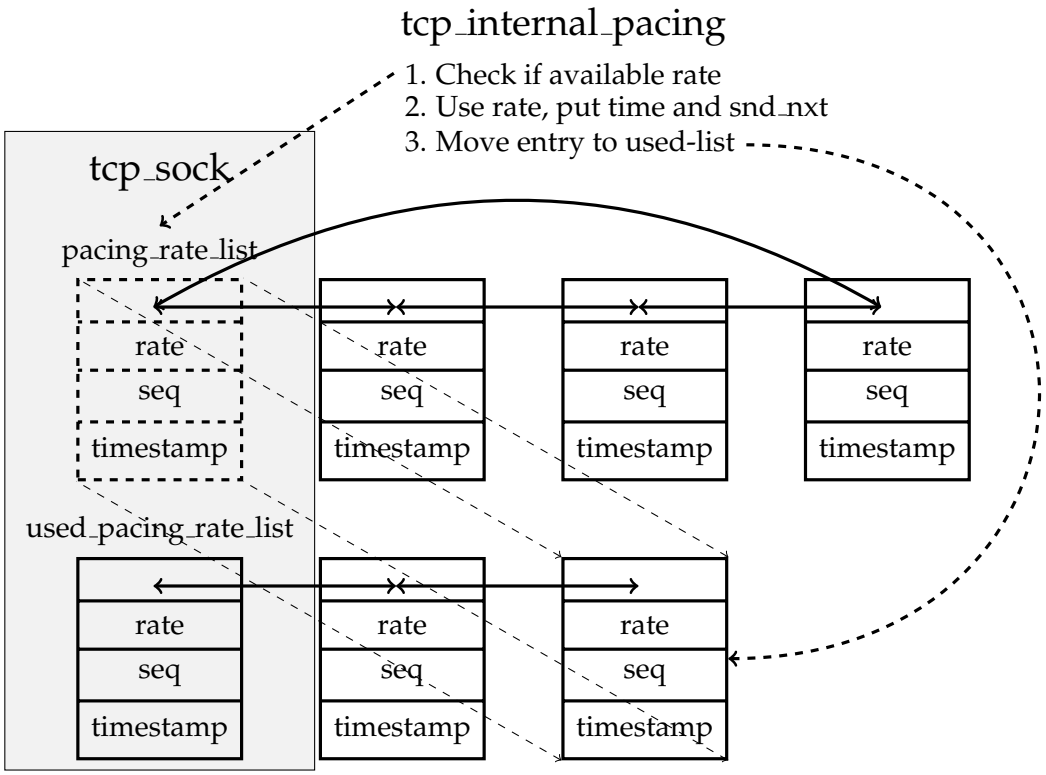


Figure 9.2: Shows how tcp_internal_pacing interacts with the two pacing rate lists. Both lists are double-linked lists. If pacing_rate_list contains an entry the first entry is updated and its rate used to set the new pacing timer. The entry is then moved to used_pacing_rate_list for the CC modules use.

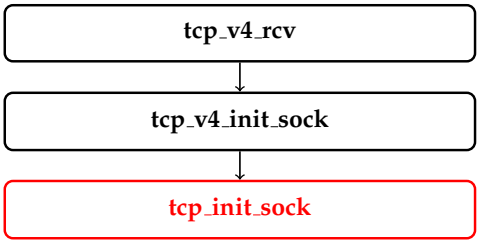


Figure 9.3: The client/initiator side creates the socket before it sends the syn-packet.

9.2.2 Per packet pacing rate

To realize chirping we needed a way to control the sending rate of individual packets. The internal pacing framework only allows us to set and update a single rate, sk_pacing_rate.

We have added two lists to the tcp_sock struct. Both consists of struct **tcp_pacing_list** entries which contains a rate, a sequence number and a timestamp. The first list, **pacing_rate_list**, contains entries with rates to be applied. A CC module can append entries to this list to control the rates of the next packets to be sent. The internal pacing framework checks if there is an entry in this list every time it sets the pacing timer. If there is an entry in the list the rate of the entry is used to calculate the new value of the pacing timer. The next sequence number (snd_nxt) and a timestamp is added to the entry, and it is moved to the second list which will be discussed shortly.

The second list, **used_pacing_rate_list**, contains entries with rates that have been applied. A CC module can examine these entries to see the timestamp and the next sequence number (snd_nxt) at the time the rate was applied.

Figure 9.1 shows the changes in red, and fig. 9.2 shows how the kernel code interacts with the two lists in more detail.

Discussion

We decided to go with a list mainly because it is very flexible. Having one entry per chirp with the necessary parameters might be more efficient in terms of memory usage (both size and cache), but it lacks flexibility.

To improve accuracy we considered having a callback to the CC module every time a new rate was needed. We decided not to do so because we thought the overhead would be too great. This might be false and we have added investigation as future work in section 12.2.

Paced Chirping uses time and not rate, but since the existing kernel code uses rate we decided to convert time to rate and let the kernel convert it back to time.

List initialization

The pacing lists can be initialized by the TCP CC module when the congestion control is initialized, but since the list belongs to the tcp_sock

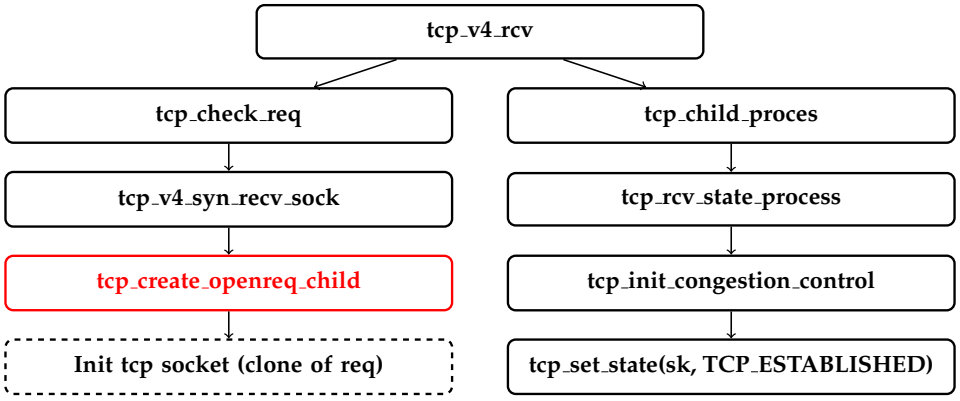


Figure 9.4: The server/listener side creates the socket when it received the ack of the 3WHS

struct we decided that it was better to initialize it where the rest of the tcp_sock is initialized.

It turned out to be more difficult than expected because a new TCP socket is initialized in two different placed depending on whether or not it initiates the connection.

The initiator side initializes its socket in the function highlighted in red in fig. 9.3, tcp_init_sock. The listening side initializes its socket when it receives the acknowledgment of the 3WHS. Figure 9.4 shows the significant parts of the sequence of calls when the acknowledgment is received. The function highlighted in red is the place where the TCP socket's variables are initialized, and where we have put the list initialization.

The two lists are initialized in both of the highlighted functions.

9.2.3 Make delayed acks toggleable

Chirping relies on per-packet queueing information, and therefore we had to add the option to turn delayed acks off. Responding to every data packet with an ack is commonly called quickack. It is the receiver side that has to have delayed acks turned off, and there is currently no protocol for the sender to negotiate the ack policy. The Linux kernel has a heuristic for the receiver to detect when a sender is in slow start. The heuristic does not understand chirps and how they behave, and thus turns on delayed acks too quickly. A sender can detect use of delayed acks by looking at the number of bytes acknowledged in each acknowledgment (Ignoring proxies that do ack-compression).

We have added a sysctl variable called tcp_delayed_acks that controls the ack-behavior of a host. It can have values 0 and 1. If it is set to 1 delayed acks will be sent if it is what the TCP stack wants. It if it set to 0 the kernels decision to send a delayed ack is overridden, and a regular ack is sent.

We envision that in the future a protocol will be added for signaling the preferred ack-policy to the receiver if Paced Chirping is proven useful.

9.2.4 Prevent CWR upon receiving CE-mark(s)

When the Linux kernel receives an acknowledgment with the CE-bit set it enters congestion window reduction, cwr. The cwnd is reduced and no new packets are sent until an acknowledgment without the CE bit is received. In the early stages of this project we tried to override this reduction from the CC module, and this seemed to work. Once we started looking at chirping this was no longer the case. We saw no other option than to add an option to not enter cwr upon reception of CE.

We added the variable disable_cwr_upon_ece in the tcp_sock struct. If it is set to any value but 0 the TCP stack does not enter cwr when a CE marked acknowledgment has been received. The variables name contains ece because the kernel uses FLAG_ECE to check if the acknowledgment was marked.

Sysctl is not an option for this change as we need cwr after slow start has terminated.

9.3 TCP CC module Implementation

It would be too extensive to discuss all the details of the code, but we will try to mentioned the most important parts. The code can be found by reading Appendix A.

Module callbacks

The TCP stack provides a number of callbacks at different events with different information. These are defined in **struct tcp_congestion_ops**. Here follows the functions, the information they provide, and what the algorithm does:

- **pkts_acked(rtt):** Record ack arrival time. Measure inter-arrival time. Record queue delay. Start new chirps. Analyze completed chirps. Check if paced chirping should terminate.
- **in_ack_event(ECN flag, acked bytes):** Used to record ECN-marks. They are currently unused, but might be used in the future. Cleaning and recording used pacing rate entries. Acked bytes is used to make inter-arrival time measurement handle less than MSS sized segments.
- **cong_avoid(packets acked):** Remove traditional exponential increase

Record inter-send time

The actual inter-send time of packets in a chirp is susceptible to noise in the kernel, so the requested gaps cannot be used in the analysis of a chirp. Therefore, we record the inter-send times by looking at the used pacing rate entries which have a timestamp for when each entry was used.

A chirp has $N - 1$ pacing rate entries and one entry for the guard interval. We will refer to these with E_i, where $i = 1 \; .. \; N$. Each entry

has a timestamp for when it was used. The recorded inter-send time of packet i and $i-1$ will be denoted by S_i, where $i = 1 \,..\, N-1$. We have the following:

$$S_i = E_{i+1}.\text{timestamp} - E_i.\text{timestamp},$$

where $i = 1 \,..\, N-1$.

Discussion

The timestamps are set when the packet is sent from the TCP-stack to the IP-stack. A packet goes through multiple steps from being passed off to the IP-stack to actually being transmitted by the NIC. Packets go through the Q-Disc(s) of the interface before they end up in the NIC driver queue, ready to be transmitted. These steps can introduce additional delay variation which we cannot see.

Ideally we would like to know exactly when the NIC transmitted each packet. This would lead to a much better measurement of the inter-send gaps and thus more accurate analysis and possibly better performance.

So how simple is it to do something like this? If the NIC supports TCP Timestamping the inter-send time could be calculated using these timestamps instead of those from the TCP stack. Support for microsecond or adjustable granularity is needed; see section 6.3.3 for discussion on TCP timestamp granularity.

Recording and Measuring time

The current time is recorded by calling to **ktime_get** and the returned value is converted to nanosecond precision using **ktime_to_ns**. The inter-arrival time of two acks is measured by calculating the difference between the recorded arrival times of those acks.

Scheduling a chirp

The code allocates memory, using **kzalloc**, for chirp meta-data and pacing rate entries in a single call to reduce overhead. We have observed that a call to **kmalloc** and **kzalloc** can take as much as a couple of microseconds. The memory is free when a chirp has been analyzed or when the CC module is instructed to terminate through the **release** callback.

An arithmetic progression of gaps allows us to use addition when calculating subsequent gaps. The gaps are converted to rates using multiplication and division. It might be beneficial to change the pacing code to accept gaps instead of rates, but as already explained the focus was feasibility so optimizations has not been prioritized.

Handling used pacing rates

Each time a new acknowledgement arrives the used pacing rate list is checked for used entries. If there are any used entries they are examined

and removed. First the metadata structure that the entry belongs to is located. Then the entrys timestamp and a previously seen timestamp is used to calculate the recorded inter-send time of the packet acked. Lastly the entry is removed from the list.

Module parameters

The following module parameters have been added.

- **dctcp_ss_gain** The initial gain of the algorithm.
- **dctcp_chirp_geometry** The geometry of the chirps.
- **dctcp_chirp_L** L in the analysis of chirps.

Note that parameters F, S, H and N are hard-coded in the implementation. Other parameters such as enable and disable Paced Chirping might be added in the future.

We have changed the default value of **dctcp_alpha_on_init** to 1 (effectively 0) from 1024.

Part III

Evaluation

Chapter 10

Testbed, Tools and Methodology

This chapter will present and discuss the physical testbed and its configuration, the tools used and methodology. We will go through the tools used for running experiments and analyzing the data. We will clearify how certain metrics are obtained and computed.

In addition to the physical testbed we have a docker-based testbed that has the same topology as the physical testbed. It has been used primarily for development and preliminary experimentation. We will not discuss it here.

10.1 Physical testbed

The testbed consists of 5 machines and a gigabit switch connected in a dumbbell topology. Figure 10.1 shows how the testbed is connected. There are two servers and two clients. The data flows from the servers to the clients through the AQM machine which is configured to be a bottleneck. The clients are connected to a switch to make it so that they share the bottleneck. The servers are connected to the AQM directly. Each machine

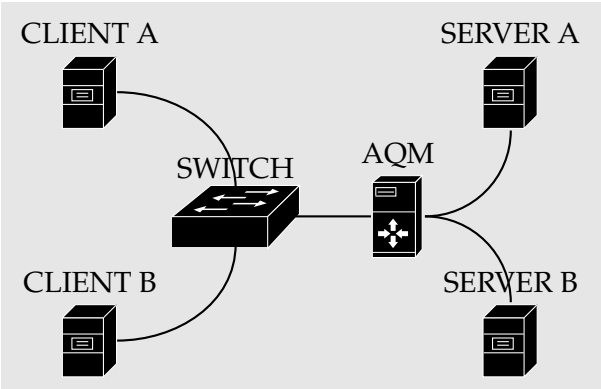


Figure 10.1: Testbed topology

Variable	value	Comment
tcp_ecn	1	Send ECN-req in SYN
tcp_delayed_acks	0	Toggle delayed acks
tcp_no_metrics_save	1	Toggle reuse/caching
tcp_low_latency	1	Turn off Nagle's
tcp_autocorking	0	Similar to Nagle's
tcp_fastopen	0	Can omit 3WHS
rmem_max	8388608	
wmem_max	8388608	
rmem_default	8388608	
wmem_default	8388608	
tcp_rmem	8388608 8388608 8388608	
tcp_wmem	8388608 8388608 8388608	
tcp_mem	8388608 8388608 8388608	

Table 10.1: Sysctl variables used on all machines in testbed

has an interface to a local network that is used for control-traffic to avoid tainting the experiments.

All the machines have Ubuntu 16.04 installed. The clients and servers are running a modified linux kernel version 4.13.16. The AQM runs 4.10.0-42-lowlatency. Information about each machine can be found in the table below. All the network interface cards support 1Gbit/s. The switch is a Netgear GS108Tv2 Gigabit smart switch.

Machine	NIC	Driver v.	Firmware v.	Kernel
CLIENTS/ SERVERS	Intel 82574L Gigabit	3.2.6-k	1.8-0	Linux 4.13.16
AQM	Intel I350 Gigabit	5.4.0-k	0.93	Linux 4.10.0-42-lowlatency

10.1.1 Machine configuration

In this section we will go through and discuss the configuration of the machines.

Sysctl variables

Sysctl variables is a way for the user to alter kernel behaviour by changing the value of exposed variables. On the AQM machine there are close to 60 tcp-related sysctl variables. Table 10.1 shows the variables that we are actively controlling and their value. The rest of the variables are set to their default. The class to which each sysctl belongs (net.ipv4, net.core) has been omitted for brevity. The sysctls starting with tcp belong to net.ipv4, while the other four belongs to net.core. The memory-related variables, starting at rmem, are set so that memory should not become the bottleneck of the system.

Network Interface Card configuration

The following list is the commands that we run on every interface to disable or alter NIC-functionality. They will be discussed in this section.

```
ethtool -K INTERFACE gso off                    //Turn off gso
ethtool -K INTERFACE tso off                    //Turn off tso
ethtool -K INTERFACE gro off                    //Turn off gro
ethtool -K INTERFACE tx-gso-partial off         //Turn off gso
ethtool -K INTERFACE sg off                     //Turn off
    scatter-gather
ethtool -A INTERFACE autoneg off tx off rx off //Turn off
    ethernet flow-control
ethtool -C INTERFACE rx-usecs 0 tx-usecs 0      //Turn off
    Coalescing
```

Interrupt Coalescing is a feature where the NIC can wait for a certain amount of time after receiving a packet before interrupting the machine in the hopes that it can process several packets in one interrupt. Interrupt coalescing is necessary to utilize 10Gbps capacity according to [36].
TCP/Generic Segmentation Offload (TSO/GSO) lets the kernel send segments larger than the MTU to the NIC. The NIC will then segment the large segment into packets fitting the MTU. This both reduces the CPU load and reduces the frequency of communication between the kernel and the NIC.
Generic Receive Offload (GRO)[1] allows the NIC driver to combine several packets into a single packet that is delivered to the kernel.

All of the mentioned features are turned off on all the machines. As implemented, Paced Chirping does not work with these features because it relies on one ack per packet and accurate time measurements.If the solution proves fruitful the functionality can be implemented in the NIC while using these optimizations.

Driver Queue The driver queue is unchanged. TCP small queues is a mechanism implemented in linux that deals with excessive queueing between the TCP stack and the NIC by delaying sending of packets when it determines an increase in queueing[37].

Ethernet flow control Ethernet flow control is a mechanism that allows a congested network device to pause transmission of ethernet frames by sending a PAUSE-frame to its peers. This mechanism can introduce delay and can push the bottleneck to the NIC. Ethernet flow control is turned off (the default) on all the interfaces to ensure that this cannot be the bottleneck.

Power management

We had some unexpected issues with the accuracy of pacing in the physical testbed which we were able to resolve.

[1] https://lwn.net/Articles/358910/

Processors have power saving modes that are used when the processor is lightly loaded or idle to reduce energy consumption. There are several modes, commonly referred to as C-states, which stop or reduce different functionalities or configurations. Entering these modes comes at the price of reduced performance. The kernel chooses which mode should be used based on the current system load.

In mode C0 the processors functionality is fully turned on so there is no energy saved and it provides maximum performance. In mode C3 the internal clock on the CPU is turned off. In mode C6 the total voltage of the CPU can go as low as 0V which means that it is practically turned off. [45]

We discovered that the power saving modes have a major negative impact on the accuracy of internal pacing. During experiments the servers were not loaded enough for the processors to exit the upper C-states, which resulted in an inaccurate pacing timer. To remove this issue we forced the CPU to go no lower than C0 C-state by adding setting GRUB_CMDLINE_LINUX_DEFAULT in /etc/default/grub to "processor.max_cstate=0 intel_idle.max_cstate=0". In addition we set each CPU's scaling governor to performance mode instead of powersave mode. A scaling governor controls the frequency of a CPU, and can adjust it automatically based on system load [61].

These two changes made the pacing much more accurate. On a busy server or desktop these setting should not be required as long as the load is high enough for the CPU to avoid the upper power saving C-states. The issue was not observed in our docker-setup simply because the CPU was busy enough to stay in the lowest C-states.

Discussion The findings suggests that paced chirping cannot be used in devices that are lightly loaded and configured with power saving capability enabled. Cellphones and laptops belongs in this category. More experiments have to be conducted to confirm this.

The pre-existing pacing-implementation does not aim to send packets at a certain rate; rather it prevents them from being sent faster than a certain rate. The rate is primarily decided by the ack-rate, not the pacing rate. The pacing-implementation might have to be changed to be able to send at a given rate, not just limit to a given rate.

10.1.2 Bottleneck and simulated delay

The bottleneck is created at the egress interface from the AQM to the switch (clients), using a Hierarchical Token Bucket (HTB) qdisc. A HTB is a policer that can limit the rate at which packet pass through it.

Netem is used for simulating delay. Netem is a network emulator for linux that provides traffic control capabilities such as adding packet loss, jitter, duplication and more. Netem is placed on the egress interfaces of the AQM towards the servers.

Configurations of netem with high BDP affect the precision of the bottleneck's rate and the simulated delay because the number of pending interrupts grows with the BDP.

Modified RED Qdisc

We have made two changes to the standard RED qdisc. The first change allows the receiver to record the queue delay (solely for experimental purposes). The time spent in the queue is computed for each packet, and the result is put in the id-field of the IPv4-header. The receiver can read the queue delay for each packet. The second change adds support for a time-based marking threshold. When a packet is dequeued the time that packet has been in the queue is used to determine if that packet should be marked. We had to make changes to the tc-command of the ip-route package to support configuration of the time-based RED qdisc. See Appendix A for the code for the modified RED qdisc.

Delay-based marking threshold

We have introduced three new parameters qth_min_us, qth_max_us and use_time. The first two set the min and max thresholds of the ramp in units of microseconds (as opposed to bytes). The toggle, use_time, switches between delay-based and byte-based marking threshold.

If use_time is toggled marking is moved from enqueue to dequeue and the standard byte-based calculations are omitted. When dequeue is called we calculate the delay of the next packet in the queue, called qdelay, and compare it to the delay thresholds. If the delay is less than or equal to the minimum threshold we simply return the packet. If the dalay is greater than or equal to the maximum threshold we mark the packet with CE. If the delay is in between the minimum and maximum thresholds we mark it with a linear probability. We set internal variable delta to $\frac{2^{32}-1}{\text{qth\_max\_us}-\text{qth\_min\_us}}$. The packet is marked if the following is true: $(\text{qdelay} - \text{qth\_min\_us}) * \text{delta} > \text{prandom\_u32}()$, where prandom_32 returns a random variable in the range $0 - \left(2^{32} - 1\right)$. If we expand delta we have the following

$$(\text{qdelay} - \text{qth\_min\_us}) * \frac{2^{32}-1}{\text{qth\_max\_us} - \text{qth\_min\_us}} > \text{prandom\_u32}()$$

$$\left(\frac{\text{qdelay} - \text{qth\_min\_us}}{\text{qth\_max\_us} - \text{qth\_min\_us}}\right) * (2^{32} - 1) > \text{prandom\_u32}()$$

So we can see that it has a linear probability of marking the packet.

To add configuration support we added a new parameter, time, to the tc (traffic control) command. When time is provided the third (previously unused) bit of the flags variable that is given to the RED Qdisc is set. This signifies that parameters max and min are specified in microseconds instead of bytes.

10.2 Tools

10.2.1 Traffic generation

There are many traffic generators available. We have had issues with iperf's CPU-usage before so we decided to go with netcat[2]. Data is fetched from /dev/zero which is a special file that provides NULL-bytes.
D-ITG[3], Distributed Internet Traffic Generator, is used to generate background UDP traffic.

10.2.2 Data collection

Tcpdump[4] is used for capturing all packets going through the servers and clients. Only the first 96 bytes of each packet is saved. This is to conserve space.
Ftrace[5] is used to get information such as the dctcp-alpha value from the TCP CC module. It has mainly been used for debugging and assuring that the solution works as expected.
Tcpprobe[6] is a module that adds a hook before tcp_recv to log information about a TCP flow's state. We have used it to extract the congestion window and slow start threshold.

10.2.3 Analysis Tools

I have used python3 for analysis, aggregation of statistics, and visualization. Matplotlib has been used for plotting. Below are the tools I have used to analyze raw data files from the experiments

Mergecap Mergecap is used to combine tcpdump output files. This is useful for comparing to tcpdump-files from different machines. Section 10.3.1 discussed some of the issues with combining tcpdump data from different machines.

C-based Pcap parsers We use self-written C-based programs to compute throughput and retrieve queueing delay. There are many tools for computing throughput, but we used our own given the simplicity of creating a new one and the fact that we could tailor it to our needs (arguments, output file, filters) made the choice easy.

10.3 Metrics

In this subsection we will discuss how we collected and computed the used metrics.

[2] https://packages.ubuntu.com/xenial/netcat
[3] http://traffic.comics.unina.it/software/ITG/
[4] https://www.tcpdump.org/
[5] https://github.com/torvalds/linux/blob/master/Documentation/trace/ftrace.txt
[6] https://wiki.linuxfoundation.org/networking/tcpprobe

Throughput Throughput is computed with a 1-round-trip time interval which means that it might contain spikes caused by bursts and silent periods. Such a small time interval is necessary to be able to see difference in throughput in slow start which can last for only a couple of round-trip times. It might have been appropriate to go as low as half a round-trip time interval.

The throughput of a time interval is computed by dividing the sum of bytes seen by the duration of the time interval. A packet consists of headers and data. To get the number of data-bytes of an IP-packet we subtract the size of the IP-header and the TCP-header from from total length of the packet. The calculation takes IP and TCP options into account.

Convergence Time Convergence time is measured in number of round-trip times. A new flow is defined as converged once its throughput is within [80%-120%] of its fair share, and the average throughput of all the running flows stays within [80%-120%] of the fair share for the next 10 round-trip times. Fair share is the capacity divided by the number of flows. 10 is an arbitrary number that seems to reflect what we have observed in data and preliminary plots. [25] uses a similar definition of convergence time; a new flow is considered converged when its short-term throughput is within 80% of its long-term average value. It is unclear whether or not they consider the situation where the new flow exceed its long-term average.

It is necessary to look at all the flows' throughputs a short period after the fair share has been reached because it can detect a burst of increased throughput and also starvation of the exisitng flows. Looking only at the moment the new flow reaches 80% of its fair share is insufficient. For example, when a flow overshoots the capacity in regular DCTCP slow start the throughput might spike to 80% of the capacity, but it will quickly reduce its throughput thereafter. Note that capacity and fair share is the same when there is only one flow, which is assumed in the previous example.

This metric fails to tell whether the flow came from above or below the fair share when it converged. Incorporating this into the metric is future work.

Flow completion time Flow completion time is calculated by subtracting the timestamp of the first syn-packet from the timestamp of the last acknowledgement. This could have been measured by the application, but the difference is minimal. Also, doing it in the application would obscure the results with time taken by other OS-related processing before and after transmission.

Queue length Queue length is calculated in the modified RED qdisc and put in the IP-header ID field for each packet, purely to collect data during experiments[7]. The queue length is calculated by subtracting enqueue and

[7]This approach is from ongoing PhD thesis work by Olga Bondarenko

dequeue time for each individual packet. The granularity is 1 microsecond, and since the ID field is 16 bits the highest supported queue length is 65.532 milliseconds. To speed up calculation a shift of 10 is used instead of a divison of 1000. The source code can be found in Appendix A. The queue length of each packet is read offline, after the experiment, from tcpdump-files.

The calculation and writing of the queue length into the IP-header of each packet is only used for experimentation, it is not intended for use.

10.3.1 Clock skew issue

We have analyzed data from four different physical machines. Each machine has its own clock, and these can drift out of sync. Clock skew is a problem that cannot be avoided, however we have done our best to identify experiment runs with this issue and rerun them. We identified these runs by visually comparing the observed queueing delay over time from two different machines. The queueing delay should be aligned, and a skew in the clocks makes the observed queueing delay misaligned.

This issue can affect convergence time because it relies on comparing throughput from different machines.

Chapter 11

Experimental Evaluation

We will use Paced Chirping as a term for DCTCP with Paced Chirping instead of regular slow start. If not specified the sender sets the gain and geometry to 2. If not otherwise specified, a byte-based RED AQM configured as a step-threshold with no queue averaging is used and the marking threshold is set to $0.17 * \text{BDP}$.

The boxplots shown in many figures have the following properties [1]. The boxes extend from $Q1$ to $Q3$, where Q stands for quartile. The whiskers extend to $Q3 + 1.5 * (Q3 - Q1)$ and $Q1 - 1.5 * (Q3 - Q1)$. Values outside the whiskers are considered outliers, shown as circles.

I will be using pX to denote percentiles; e.g. p95 is the 95th percentile.

11.1 Varying chirp geometry

The chirp geometry controls how wide a range of rates each chirp probes for.

We wanted to investigate how paced chirping behaves with different geometries. We ran an experiment with base round-trip time 30 ms and capacity 100Mbps, and we varied the chirp geometry with the following values: 1.2, 1.4, 1.6, 1.8, 2.0. The experiment has four flows that are started after one another with sufficient time to converge before the next flow is started. We will refer to this as a *staggered start*. The experiment is run 10 times for each geometry.

Convergence time

Figure 11.1 shows the convergence time with varying geometry for the four flows. Convergence time is measured in round-trip times.

The first flow starts in an empty network. The convergence time improves as the geometry decreases because the final estimate becomes more accurate with decreasing geometry. A smaller geometry homes in on an estimate resulting in a more accurate estimate. This can be seen in table 11.1. The actual gap is 120 µs.

[1] https://matplotlib.org/api/_as_gen/matplotlib.pyplot.boxplot.html

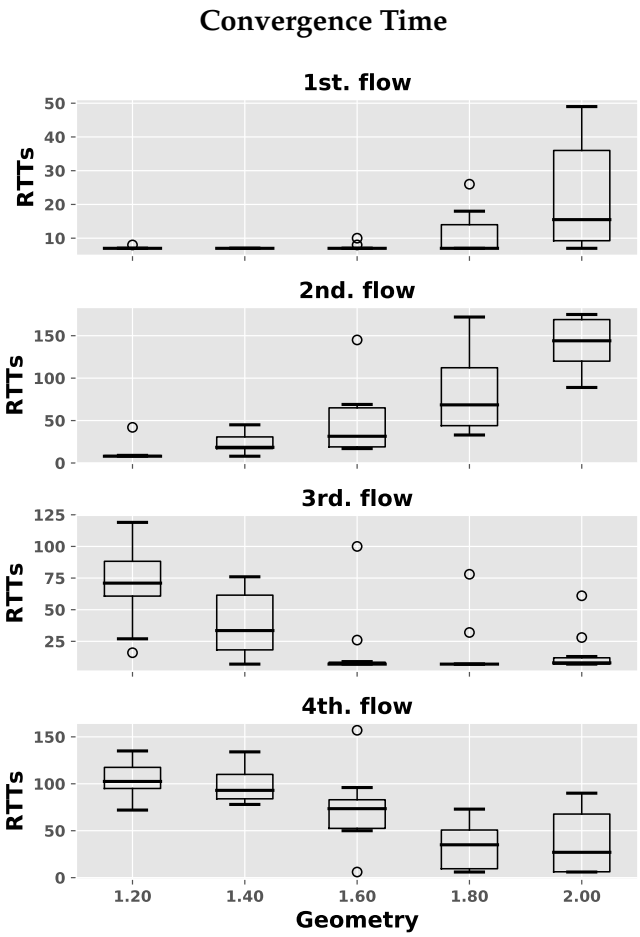


Figure 11.1: Convergence time measured in round-trip times of four flows started staggered. The round-trip time is 30 ms, and the capacity 100Mbps. The first flow starts in an empty network, and we can see that the convergence time gets better as the geometry decreases. The 3rd and 4th flows show that the responsiveness of the new flow decreases with decreasing geometry.

For the other flows, starting in a fully utilized network, the geometry affects the algorithms ability to adapt to the other flows. The geometry controls the range of rates each chirp probes for. A high geometry allows the algorithm to make larger adjustment compared with a small geometry.

The second flows convergence time improves as the geometry decreases. The reason is that the flow is more aggressive with a small geometry than with a large geometry. The flow starts with an estimate close to the total capacity from the initial packet trains. This estimate is then adapted to the information from the following chirps until the network is filled. If the chirps have a small range the estimate will normally change less than when the range is large.

For the third and fourth flow the trend is reversed; higher geometry gives better convergence time. This is because lower geometry cause the flow to exceed the fair share.

A flow starting in an empty network benefits from having a small

Final Gap estimate (µs)

Chirp geometry	mean	sd
1.2	116.9	5.6
1.4	115.6	6.2
1.6	114.9	10.1
1.8	125.6	9.7
2.0	137.1	21.2

Table 11.1: Final gap estimate of a single flow entering an empty link over 10 runs with varying chirp geometry. The expected value is 120 µs. Increasing the chirp geometry increases estimate variance. It also seems to lead to a more conservative estimate.

geometry because it helps it focus its chirps on the estimate which gives more accurate information. When there are existing flows the convergence time depends on the relationship between geometry, number of flows, initial packet train estimates and BDP. There is no clear relationship between geometry and convergence time when the network is fully utilized.

Queueing delay

Figure 11.2 shows the distribution of the maximum queuing delay in the first 15 round-trip times over 10 runs with varying geometry.

We will consider the flow starting in an empty network first. In 6 of a total of 50 experiments the capacity is overestimated resulting in an overshoot. Only two of them are above 2 ms, and the marking threshold is roughly 5 ms.

There is a clear increasing trend in queueing delay when the geometry reduces for the second, third and fourth flow. This supports the claim that the geometry affects the aggressiveness and the algorithms ability to make large adjustments.

Discussion

We have now seen how the geometry affects queueing delay, convergence time, responsiveness and estimate accuracy. The insights can be used to improve the algorithm.

A constant value for the geometry is undesirable because the optimal value depends on the utilization of the network. The geometry should be low if the network is unutilized to improve accuracy, but a low geometry causes large queueing delay when the network is fully utilized.

Ideally the value of the geometry should be variable and its value depend on the detected characteristics of the traffic. If the flow seems to be alone the geometry should be small, and if it competes with other flows the geometry should be high.

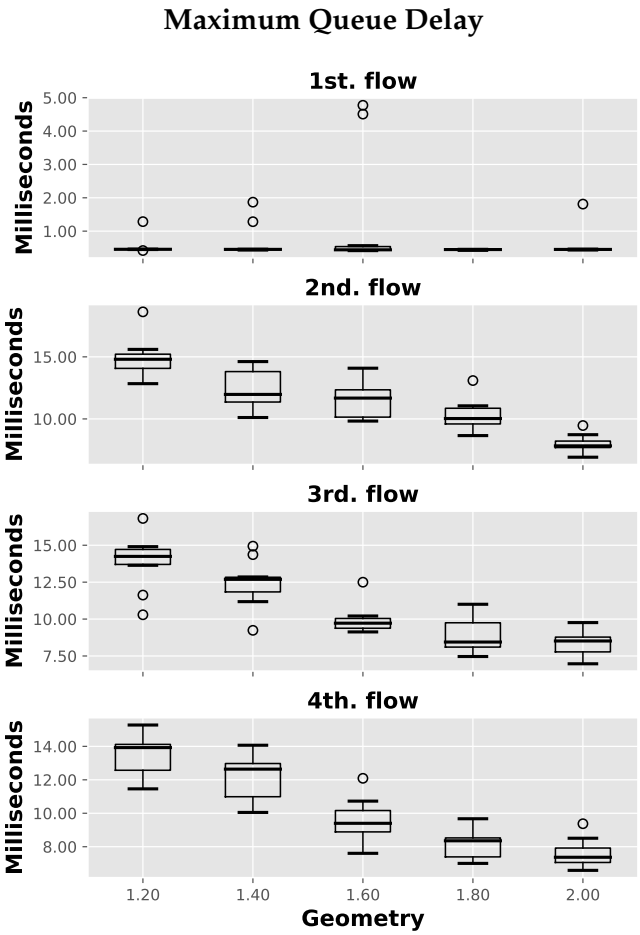


Figure 11.2: Maximum queueing delay measured over the first 15 round-trip times after flow start. The round-trip time is 30ms, and the capacity 100Mbps. Note that the scales on the vertical axes are different. For the first flow there is no significant relationship between chirp geometry and queue delay. Queue delay increases when the geometry decreases for the other flows that starts in an utilized network.

In section 11.6 we present and evaluate a modification to Paced Chirping that adjusts the geometry (and gain) based on the variation in the rate estimates.

11.2 Varying chirp gain

The chirp gain controls how fast the number of chirps increases. In this experiment we investigate how the gain affects the queueing delay and convergence time of Paced Chirping. The capacity is set to 150Mbps and the round-trip time to 30 ms. The experiment has four flows starting staggered and is repeated 15 times.

Figure 11.3 shows the convergence time of the four flows. Figure 11.4 shows the maximum queue delay during the first 15 round-trip times of each flow.

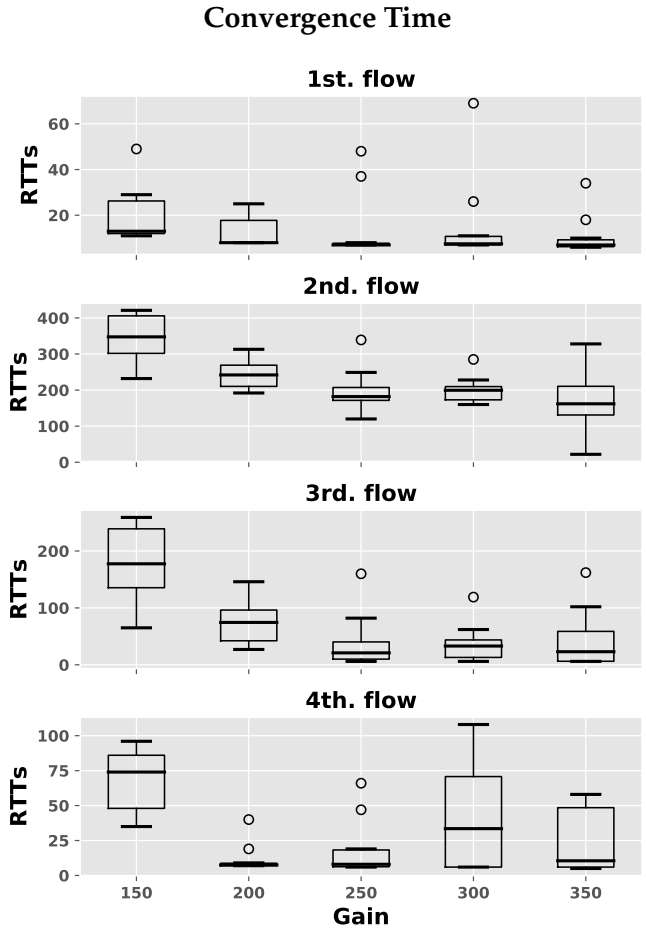


Figure 11.3: Convergence time measured in round-trip times of four flows started staggered. The round-trip time is 30ms, and the capacity 150Mbps. The gain values are scaled by a 100. For the first flow a higher gain seems to improve convergence time as it allows for faster acceleration to the estimated capacity. For the other flows the gain improves convergence time until the number of flows makes it so that the new flow significantly exceeds its fair share.

11.2.1 Empty network - First flow

A consequence of having a higher gain is that the final estimate might be based on fewer chirps (depends on how much the gains differ). The number of values an estimate is based on usually affects the variability and accuracy of the estimate. The variability decreases as the number of samples increases. A higher gain can lead to a more variable estimate and consequently more variable convergence time and queueing delay.

Queueing Delay

Looking at Figure 11.4 a higher gain typically causes higher queueing delay. The acceleration is too fast for the algorithm to get an accurate estimate.

Note that if the rate is persistently overestimated the gain should have no effect on the queue delay.

Convergence time

Figure 11.3 shows that the convergence time improves with higher gain, but there are more outliers. This backs up the claim that higher gain gives more variability.

Higher gain can improve convergence time by accelerating faster, but at the cost of more variable performance.

11.2.2 Utilized network

Now we will consider the second, third and fourth flow.

Queueing Delay

The number of existing flows has minimal effect, if any, on the queue delay as a new flow enters the network. This can be seen by comparing the three last subplots in fig. 11.4. This is because the flows have a multiplicative reaction. For instance two flows halving their combined rate reduces the total utilization as much as if one flow with the combined rate halves its rate.

The maximum queue delay grows with higher gain because the new flow is more aggressive.

Convergence time

The convergence time seems to improve with increased gain until the gain becomes too great for the number of existing flows. The fourth flows has poorer convergence time with gain 3 than gain 2. This is because the fourth flow significantly exceeds its fair share.

11.2.3 Discussion

We have seen how the gain affect the variability of the final estimate, queueing delay and convergence time.

A higher gain marginally improves the acceleration of the first flow in the experiment. In a network with a much higher BDP the acceleration might be more noticeable. The reason why we did not try higher BDP is that the testbed would become overloaded which could lead to false results. It might be possible to compensate the variability that comes with a higher gain by having a smaller geometry.

The gain has a clear effect on the queueing delay. When the flow is alone this is due to the variability of the final estimate. When the network is utilized it is because the new flow claims capacity faster than the existing flows yield capacity. The network becomes overutilized which causes queueing delay. Once the new flow transitions to congestion avoidance the queuing delay reduces.

As with geometry there is no one value that is optimal for when the network is utilized. Ideally it should be adapted too.

Section 11.6 presents a modification that adapts the gain (and geometry) to the variation of the estimates.

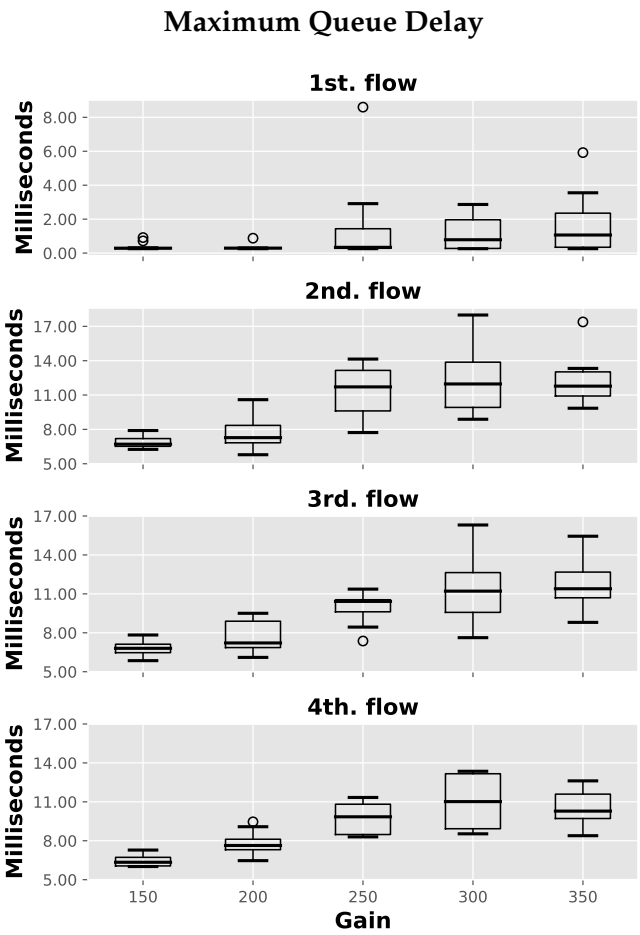


Figure 11.4: Maximum queue delay measured over the first 15 round-trip times after flow start. The round-trip time is 30ms, and the capacity 150Mbps. The gain values are scaled by a 100. For the first flow a higher gain increases the chance of overestimating the capacity which leads to higher queueing delay. For the other flows a higher gain increases the queuing delay because they push in more aggressively. Notice that the number of flows affects the queuing delay very little. This shows that the flows are reacting as if they are one flow because of the multiplicative back-off.

11.3 Flow completion time

Since paced chirping is much more aggressive than regular dctcp slow start it should improve flow completion time in an utilized network. Flow completion time (FCT) is how long it takes a flow to complete its transmission.

The goal of this experiment is to compare paced chirping to plain dctcp slow start in terms of flow completion time in a utilized network. Each run has a single greedy background flow, and 100 smaller flows started staggered over a period of 5 minutes, 3 seconds apart. We have the following factors: flow/file size in KB [75, 150, 300, 600, 1000, 2000]. The following metrics are considered: FCT, queue delay, throughput/utilization. The network has a round-trip time of 15 ms. and a capacity of 150Mb/s. Therefore, the BDP is roughly 188 packet of 1500 bytes.

The file sizes have been selected based on what is categorized as delay sensitive short messages in the original DCTCP paper [31]. Delay sensitive short messages are in the range of 100KB to 1MB, so our range is slightly wider. The network condition is semi-arbitrary. We wanted to have a BDP large enough for slow start to have significant impact on the result, and we knew from experience that the testbed handles the set capacity and latency.

Flow completion time

Figure 11.5 shows a plot of the FCT for DCTCP and Paced Chirping with varying files size. The x-axis is logarithmic.

Paced chirping performs better than DCTCP in terms of FCT for all the flow sizes in this experiment. The difference in FCT for the 1MB flow is roughly 30 RTTs.

FCT Variance

The FCT of DCTCP has more variance compared with that of paced chirping. Paced chirping has noticeable variance from 600KB, and DCTCP has noticeable variance for all the flow sizes.

DCTCPs slow start performance is determined by when the first ECN-mark is received. This depends on the occupancy of the queue. If the queue is empty a new flow gets its first ECN mark later than if the queue is full. The queue occupancy is variable; it fluctuates around the marking threshold. It follows that DCTCPs slow start performance depends on the state of the queue at the time it joins the network. This explains the variance of dctcp.

Paced chirpings performance is mainly determined by its gain and rate estimate. The gain is constant. The capacity estimate depends on time measurements that has variable accuracy. Therefor we expect to see variance in the FCT for paced chirping, but as we can see in fig. 11.5 it is very stable. Paced chirping sends 10 packets the first round, then another 16 packet the second round, and 32 packets the third round regardless of

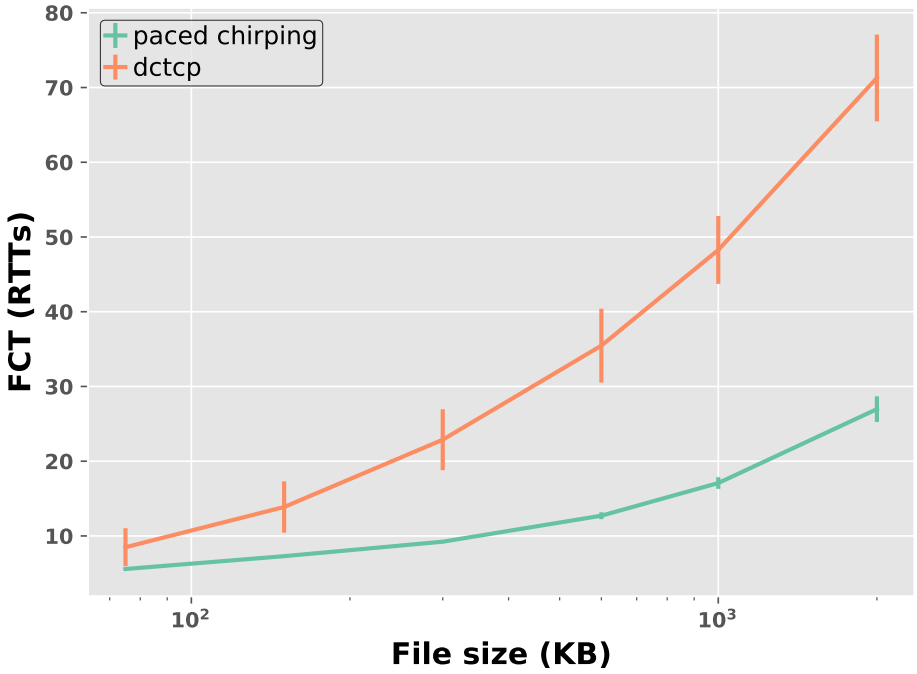


Figure 11.5: Flow completion time of DCTCP and Paced Chirping with varying file size. The whiskers represent variance. The flow completion time is greatly improved as the file size increases. For the smallest file size (75KB) the difference is minimal.

Flow size (KB)	Paced Chirping mean	median	sd	p95	DCTCP mean	median	sd	p95
75	3044	3183	588	3765	2224	2408	938	3498
150	3352	3392	732	4462	2003	1941	898	3498
300	4169	4026	1195	6236	1950	1933	811	3358
600	3281	3510	2024	6314	1938	1952	663	2862
1000	2063	944	2195	5988	1966	2000	602	2730
2000	1402	626	1734	5286	1979	2014	527	2688

Table 11.2: Queue delay in microseconds over the duration of each flow. For small files Paced Chirping causes more queuing delay. There is a trade-off between throughput and latency. The highlighted areas shows that the mean and median queueing is better for Paced Chirping than DCTCP when the flow size increases. This is related to reduction in total utilization.

the estimate. This adds up to a total of 58 packet and with a packet size of 1500bytes this is 87KB. So Paced chirping sends 87KB as long as the first estimates are great enough. Consequently there is no noticeable variance for the lower flow sizes.

Queue delay

Table 11.2 shows a summary of the queue delay during the lifetime of the smaller flows. The values are computed with the queueing delay for every packet from every small flow.

We will consider DCTCP first. The mean and median queueing delay is very stable and the standard deviation decreases with increasing flow size. It is the initial packet train(s) that cause a increase in queueing delay. So the longer the flow is the less of an impact the initial burst have on the experienced queueing delay.

Paced Chirping has an increasing queueing delay trend up to 600KB, but then it decreases. One explanation on this decrease is that the flows terminate Paced chirping around 600KB and that the packets after this experiences very little queueing delay. So there is a initial push back that causes high queueing delay before a period with very little queueing delay. Another explanation is that the existing background flow does not manage to reclaim the available capacity in between flow starts due to the additive increase. New flow will thus start with an empty queue. In section 11.3.1 we will see what effect the flow size has on the utilization.

It is clear that Paced Chirping causes higher maximum queueing delay. However, as the flow size grows beyond a certain size the average queue declines because the network becomes under-utilized.

11.3.1 Utilization

When a flow leaves the network it takes some time before the other flows (if any) claim it. During this period the network might be underutilized.

Flow Size (KB)	Paced chirping mean	median	sd	dctcp mean	median	sd
75	93.7	94.2	3.1	93.0	94.2	7.1
150	92.5	94.2	4.9	93.0	94.2	7.1
300	90.0	94.2	8.2	92.9	94.2	7.1
600	87.3	93.7	11.1	93.1	94.2	7.0
1000	87.9	94.2	10.5	93.0	94.2	7.1
2000	89.1	93.7	9.1	92.7	94.2	7.3

Table 11.3: Total utilization in percent over the duration of the experiment. The link capacity is 150 Mbps. The average utilization is significantly less for Paced Chirping compared to DCTCP. The median is comparable because the network is fully utilized most of the time.

Table 11.3 shows the total utilization over the whole duration of the experiment with Paced Chirping and DCTCP. The average utilization when Paced Chirping is used is significantly less compared to when DCTCP is used. The median is pretty similar because most of the time the network is fully utilized.

The lower mean utilization explains the decrease in median queueing delay for larger flows. If the existing flow does not fully utilize the link when a new flow starts it might get fewer marks and react less. It does not get the same shock as when it fully utilizes the network. It could also explain the lower 95th percentile for 1000 and 2000 KB.

DCTCP has a slight decrease in mean utilization as the flow size increases. The amount of capacity the shorter flows claim grows only slightly with increasing flow size.

11.3.2 Discussion

The data suggests that there is a trade-off between FCT and queueing delay. Improving FCT comes at the cost of higher queueing delay. The queue delay is not persistent, it lasts for a couple of round-trip times, and the duration depends on the frequency of new flows. In section 11.6 we will see that the trade-off is only slight. It is possible to greatly improve FCT without having to cause the amount of queueing delay we have seen in this experiment.

It might be possible to replace additive increase with Paced Chirping when a DCTCP flow in congestion avoidance detects that it has lost its regular congestion signals. This could improve the overall utilization when Paced Chirping is used because the remaining flows could adapt faster when a flow leaves.

The gain and geometry can be adjusted to weight FCT differently against queue delay. A high gain and low geometry would benefit FCT at the cost of higher queueing delay and possibly lower utilization.

RTT(ms)	Bitrate(Mbps)	K	CC	min	max	mean	sd
5	100	0.20BDP	chirping	5	17	8.4	3.6
5	100	0.20BDP	dctcp	31	32	31.4	0.5
5	150	0.20BDP	chirping	5	19	9.2	4.0
5	150	0.20BDP	dctcp	36	41	37.5	1.4
20	100	0.05BDP	chirping	6	46	16.4	12.9
20	100	0.05BDP	dctcp	135	136	135.8	0.4
20	150	0.05BDP	chirping	7	61	19.4	13.9
20	150	0.05BDP	dctcp	193	203	194.9	2.0

Table 11.4: The convergence time of flows using paced chirping and plain DCTCP with 1ms marking threshold and varying network conditions. The network is empty when the flow starts. K is the marking threshold. The unit is RTTs.

11.4 Varying ECN marking threshold

The goal of this experiment is to compare Paced Chirping and DCTCP when the marking threshold is significantly less than the recommended 0.17*BDP. Lowering the marking threshold only affect cross-traffic when Paced Chirping is used because the new flow ignores ECN marks until it transitions to congestion avoidance. We decided to use a marking threshold of 1 ms. Previous testing of the L4S service has shown that it can keep average queueing delay under 1 ms without losing link utilization [60].

We have the following factors: RTT (ms) [5, 20], bitrate (Mbps) [100, 150]. Each run has 4 flows started staggered, and is repeated 20 times.

11.4.1 Convergence of first flow

Table 11.4 shows the convergence time of DCTCP and Paced Chirping with four different network conditions. K is the relationship between the BDP and the 1 ms marking threshold.

The convergence time increases drastically for DCTCP when the queue decreases relative to the BDP. Paced Chirping is not affected by the marking thresholds relationship to the BDP. However the variability increases with higher RTT. The reason is that the increased RTT and additive increase magnifies errors in the capacity estimate. This is best explained with an example. Assume two network with the same bottleneck capacity, one with BDP 50 (flow A) and one with BDP 200 (flow B). If the estimate is 70% of the actual capacity for both flows A will stop at 35 packets and B will stop at 140 packets. A needs 5 RTTs to reach 80% of its BDP while B needs 20 RTTs. So even though the estimate error is equal flow B will have a higher convergence time.

The geometry is set to 2 and we have previously seen that the accuracy of the final estimate can be improved by decreasing the geometry.

11.4.2 Illustrative sequence diagrams

Figure 11.6 shows a time-sequence diagram of the of an experiment with RTT set to 20 ms and bitrate set to 150 Mbps.

The first flow illustrates the issue regular DCTCP has with a low marking threshold. Slow start's bursty behaviour triggers the first ECN mark long before the BDP is reached. This can be seen in fig. 11.7 where the queue delay increases in bursts.

With Paced chirping the first flow manages to accelerate quite close to the steady state throughput before it enters congestion avoidance. And fig. 11.7 shows that it does not induce significant queueing. Subsequent flows are more aggressive compared to when DCTCP is used which does improve convergence time. As previously discussed this comes at the cost of higher queueing delay as can be seen in fig. 11.7.

11.4.3 Discussion

Being able to handle a low marking threshold will be important in the future. Not just for keeping low latency but also to work with queues not configured for the specific BDP of each flow. Multiple flows traversing the same link can have different BDPs and it is very hard for an AQM to measure the RTT of each flow traversing it.

With a gain of 2 Paced Chirping can cause queueing delay to exceed the 1ms ECN threshold. In future work it would be wise to reduce the gain in response to congestion indicators such as delay and ECN marks.

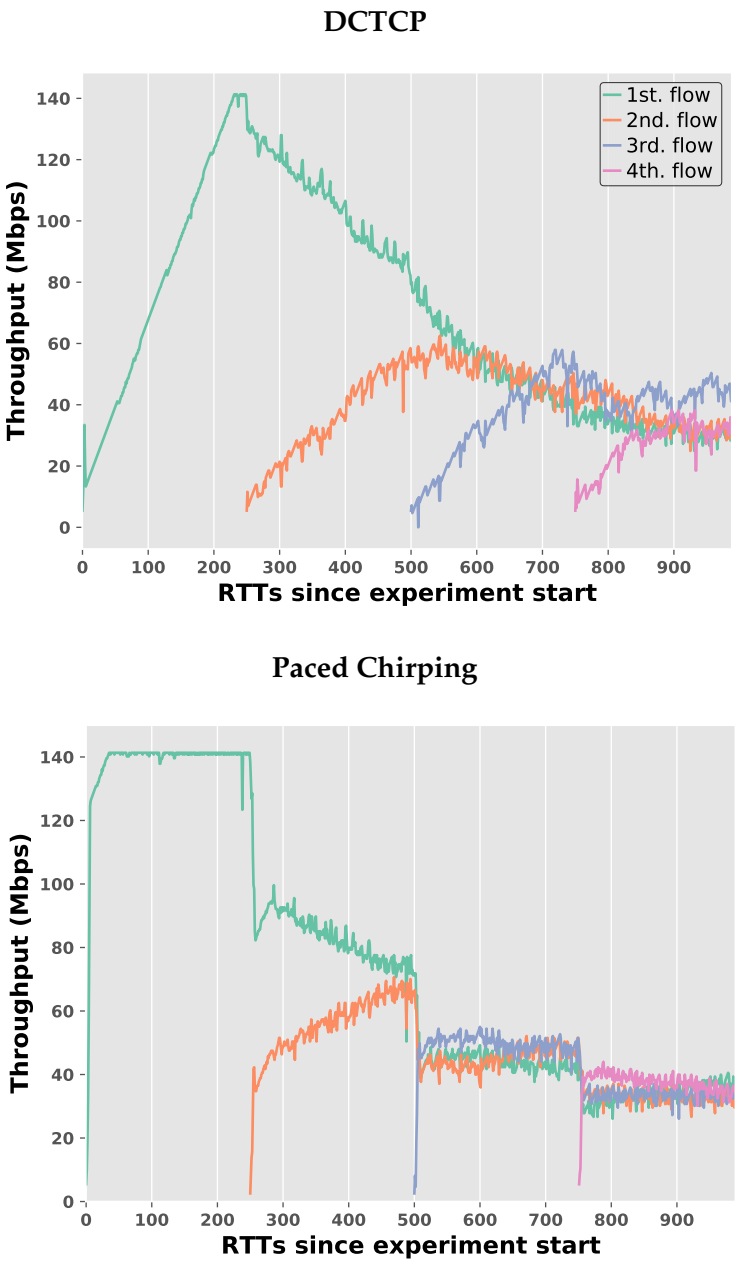


Figure 11.6: Time-sequence diagrams of the throughput of 4 flows starting staggered. The first flow with DCTCP quickly terminates slow start as the marking threshold is exceeded, and spends a long time in additive increase before reaching the capacity. The first flow with Paced Chirping manages to reach the steady state capacity in less than 50 RTTs. As subsequent flows join they push the existing flows back when Paced Chirping is used, while DCTCP terminates slow start right away.

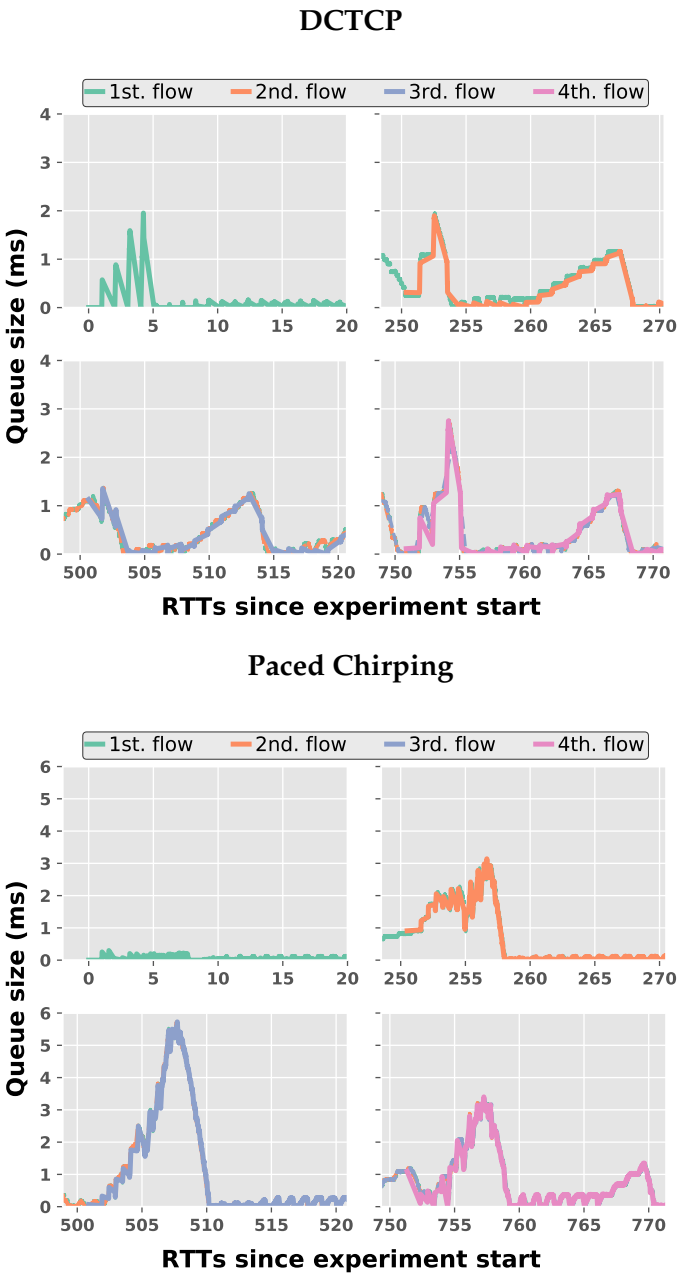


Figure 11.7: Time-sequence diagrams of the queue delay of 4 flows starting staggered.

11.5 UDP background traffic and 1ms marking

The goal of this experiment is to see how Paced Chirping behaves with more realistic traffic that is still controlled enough to understand what is going on.

We configured the network with round-trip time 20 ms and capacity 120 Mbps. The marking threshold was set to 1 ms. We added UDP traffic roughly equal to 20% of the total capacity. The inter-send time of the UDP-packets is exponentially distributed with an average corresponding to 20% of the total capacity. Then we added 12 1MB flows and 2 long lived flows all starting staggered.

Figure 11.8 plots the throughput of the 12 flows when DCTCP is used. The low marking threshold makes the 1MB flows terminate slow start quickly making them take several additional RTTs to finish. Each flow needs roughly 30-40 RTTs to finish. The first long lived flow does not manage to accelerate far beyond 60 Mbps before the second long lived flow joins.

Figure 11.9 shows the same scenario but with Pached Chirping. The first 4 flows manages to accelerate despite the low marking threshold and noise from the UDP traffic. The first long lived flow reaches 90 Mbps within a few RTTs and stays at around 80 Mbps until additional flows join. The second long lived flow manages to accelerate quite nicely. The 1MB flows finish within 10-20 RTTs.

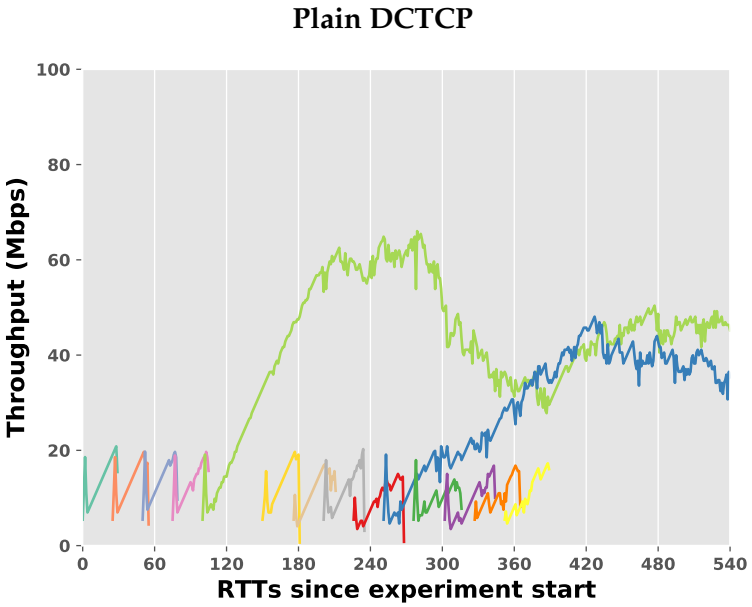


Figure 11.8: Time-sequence diagram of the throughput of 12 1MB-sized flows and 2 long lived flows with UDP background traffic equal to 20% of the total capacity

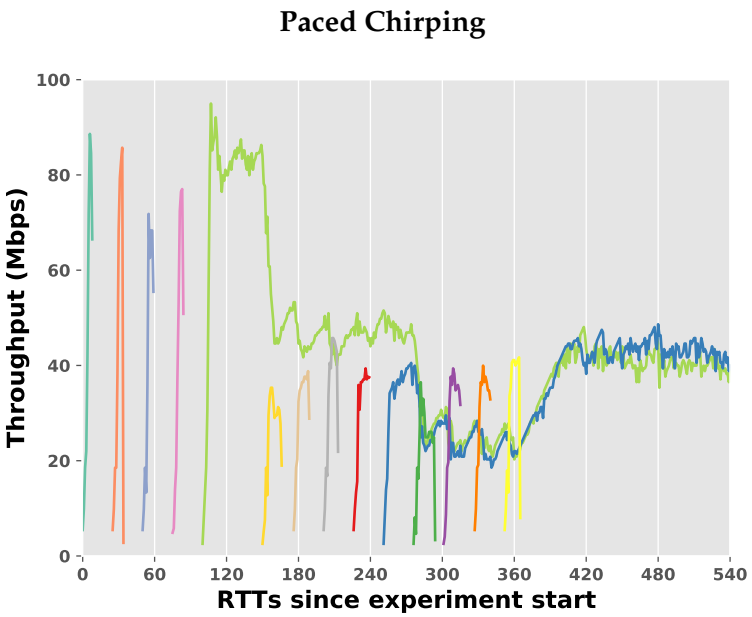


Figure 11.9: Time-sequence diagram of the throughput of 12 1MB-sized flows and 2 long lived flows with uneven UDP background traffic equal to 20% of the total capacity. The UDP traffic is not shown.

Queue delay

Flow size (KB)	With adjustment mean	median	sd	p95	Without adjustment mean	median	sd	p95
75	2974	3117	612	3681	3044	3183	588	3765
150	3040	3133	688	3959	3352	3392	732	4462
300	2459	2599	973	3798	4169	4026	1195	6236
600	2149	2095	1526	4427	3281	3510	2024	6314
1000	1743	1601	1441	4104	2063	944	2195	5988
2000	1581	1560	1095	3326	1402	626	1734	5286

Table 11.5: Queue delay in microseconds experienced by flows during the FCT experiment. Data for "without adjustment" is retrieved from table 11.2. The adjustment algorithm has less variation and causes less queueing delay, except for the highlighted areas.

11.6 Naive gain and geometry adaptation

We created a very simple adaptation algorithm for the gain and the geometry to see how the behaviour changes.

11.6.1 Algorithm

We have introduced a variable for the mean absolute deviation of the gap estimate called MAD. MAD is initialized to 0 and updated every time we get a new gap estimated with the following EWMA.

$$\text{MAD} \leftarrow h * \text{MAD} + (1 - h) * \text{error},$$

where error is abs (prev gap estimate − new gap estimate), and $h = \frac{1}{2}$.

Further, we define a threshold of 30 µs. Every time the MAD is updated the gain and geometry is also updated. If MAD is greater than the threshold we set the gain to 1.2 and increment the geometry by 0.4. If MAD is less than or equal to threshold we increment the gain by 0.2 and decrement geometry by 0.2. The bounds of the gain are 1.2 and 3. The bounds of the geometry are 1.2 and 2.

11.6.2 Homing in on the gap

Figure 11.10 shows the inter-packet gaps at the sender and receiver for Paced Chirping with and without this naive adjustment algorithm The upper plot is the same as fig. 8.4, copied here to make comparison simpler.

Comparing the two we can see that, with the adjustment, the search range decreases in response to low variance. This improves the accuracy of the estimate as we saw in section 11.1.

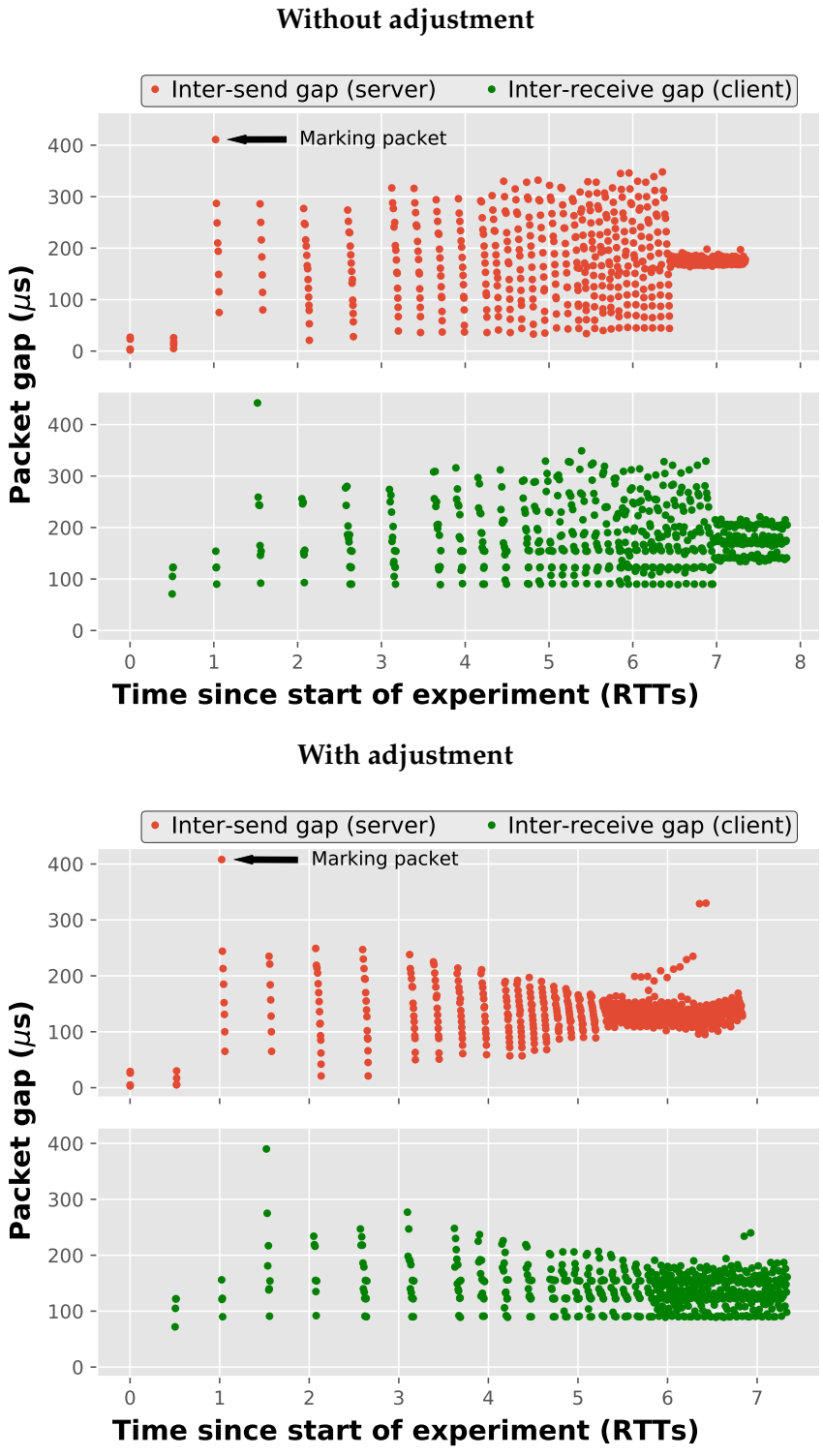


Figure 11.10: Inter-packet gap at sender and receiver when a flow starts in a unutilized network with gain and geometry set to 2. The geometry is reduced during flow start because the variance is low. This can be seen by the shrinking range in gaps.

Total throughput

Flow size (KB)	With adjustment mean	median	sd	Without adjustment mean	median	sd
75	93.7	94.2	3.1	93.7	94.2	3.1
150	93.1	94.2	4.1	92.5	94.2	4.9
300	91.3	94.2	6.9	90.0	94.2	8.2
600	90.4	94.2	8.5	87.3	93.7	11.1
1000	89.7	94.2	9.4	87.9	94.2	10.5
2000	90.9	94.2	7.5	89.1	93.7	9.1

Table 11.6: Utilization in percent over the duration of the experiment. Data for ”without adjustment” is retrieved from table 11.3. The link capacity is 150 Mbps.

.

11.6.3 Rerun of FCT experiment

To see how the naive algorithm alters queueing delay and overall utilization we re-ran the flow completion time experiment. Table 11.5 and table 11.6 show the queueing delay and total utilization with and without the adjustment algorithm. With the adjustment algorithm the total utilization increases and queueing delay decreases. The 95th percentile is significantly decreased by close to 2 ms for flow sizes above 150KB. For flow sizes 1000 KB and 2000 KB the median queue size is smaller when the adjustment is not used. This is expected because the existing flows are pushed back more, which leaves the bottleneck less utilized. This is shown in table 11.6. With the adjustments the mean utilization is higher and slightly more stable.

Figure 11.11 plots the FCT of DCTCP and Paced Chirping with and without the adjustment algorithm. It uses the same data for DCTCP and Paced Chirping as in fig. 11.5. The FCT for Paced Chirping with the adjustment is marginally higher than without the adjustment. The standard deviation is comparable to that of DCTCP. This is expected as the adjustment uses noisy time measurements.

The naive adaptation causes significantly smaller queue delay, marginally higher utilization and slightly worse FCT. The lower queueing delay will improve real-time applications without hardly affecting the FCT of smaller flows. However, as stated before, this is a naive adaptation that we expect can be improved in future work.

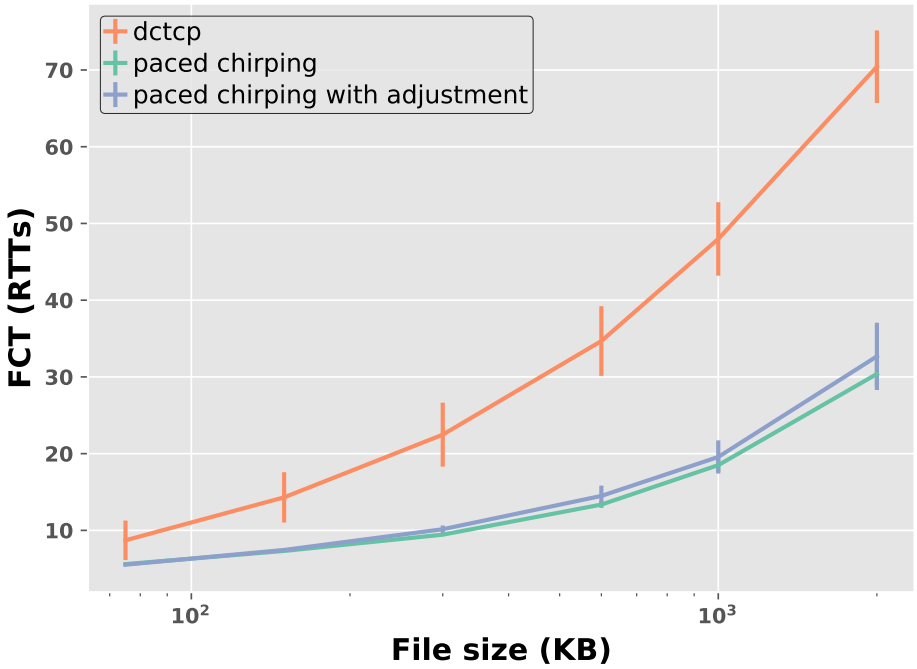


Figure 11.11: Flow completion time of DCTCP and Paced Chirping with and without the adjustment algorithm.

11.7 Implementation performance

In this section we will discuss the performance of the (unoptimized) implementation.

Memory usage

Each chirp of 16 packets requires 768 Bytes on the testbed machines. The metadata structure for each chirp uses 256 bytes. Each pacing rate entry uses 32 bytes.

The memory requirement for filling a certain BDP with chirps follows the following function.

$$\text{Memory requirement} = \text{ceil}\left(\frac{\text{BDP}}{N}\right) * (256 + 32 * N)\text{Bytes},$$

where the BDP is in units of packets and N is the number of packets per chirp.

Section 9.2.2 discusses the pros and cons of implementing chirps within the kernel, which would use less memory but be less flexible than the list structure characterized here.

When Paced Chirping terminates all the memory it used can be freed.

Accuracy limitations

When the average gap goes below 30-20 microseconds our testbed struggles to get accurate enough inter-send gaps for the algorithm to work properly. It deems a significant amount of chirps to be invalid.

One possible explanation is that the CPU does not manage to generate interrupts frequently enough. The load on the system might be too great. It might also be caused by the way pacing is implemented. It limits the sending-rate to the specified rate, but it does not attempt to enforce the rate. This might be a deliberate choice to minimize CPU usage.

Execution time

Table 11.7 shows measured execution time of the four most important functions in the implementation. Note that dctcp_acked has calls to analyze_chirp and schedule_chirp which contributes to its execution time in some executions. The data is from two different runs. This is neccessary to avoid measuring the overhead of trace_printk which is used for logging the times.

The maximum execution time for those functions that interact with the pacing lists is roughly 11 µs. This is rather alarming. It seems to be for only a few calls as the 99th percentile is no more than 1.5 µs. We do not know the exact cause of these high values, but we think that it has to do with memory accesses to the pacing lists. Further investigation will be needed once the code has been optimized.

Execution time (ns)

Function	N	min	max	mean	sd	p99
analyze_chirp	682	197	996	407	138	782
schedule_chirp	1568	100	11079	430	346	1114
dctcp_update_alpha_slow_start	11041	30	10732	146	161	501
dctcp_acked	10980	26	10887	172	352	1503

Table 11.7: Execution time for four functions. dctcp_acked includes analyze_chirp and schedule_chirp in some executions.

A known removable inefficiency is the use of two divisions for converting time to rate and then back again to time as discussed in section 9.2.2.

Part IV

Conclusion

Chapter 12

Conclusion and future work

12.1 Conclusion

We have explored slow start in context of Data Center TCP, and designed, implemented and evaluated a replacement algorithm called Paced Chirping for use in TCP Prague. Paced Chirping is a promising solution because it generally avoids queue overshoot and capacity undershoot. In both utilized and unutilized networks the frequent marks and low marking threshold cause new flows to enter congestion avoidance with poor throughput. The implementation in Linux and the evaluation on a real testbed shows that Paced Chirping is feasible.

ECN marks with a step marking threshold exhibit a bursty behaviour which is difficult to use in the start-up phase. In addition ECN cannot signal how severe the congestion is when the queue is persistently over the marking threshold. While ECN is a great signal in steady state we question its usefulness in slow start. It will probably have a more supportive role.

Initially we explored the effects of a naive pacing approach in an unutilized network. When paced, slow start does not exhibit bursty behaviour which prevents it from triggering ECN-marks before it reaches the capacity. This improves throughput, but it also leads to more queueing delay. As the pacing rate increases each round it gets absolutely no warning information until it suddenly exceeds capacity and causes a large queue.

The naive solution estimated the capacity of network and used it and alpha to prevent overshoot and prevent undershoot. It showed that it is possible to push existing flows back without causing excessive queueing. The solution could not adapt to changes after the first two RTTs and had issues with the termination condition.

Paced Chirping is a purely delay based algorithm that uses available capacity estimates from chirps to find the available capacity. It does so by gently stressing the bottleneck repeatedly to cause queuing which can be measured and used to estimate the available capacity. This gives a series of estimates which are used to adapt the estimate of the available capacity. Since Paced Chirping is purely delay based it is not affected by marking behaviour, and can be used over bottlenecks that do not support ECN.

Paced Chirping has two variables, gain and geometry, that control

aggressiveness and estimate accuracy. The geometry can be set lower to improve accuracy, but set too low it runs into the problem that regular pacing has; does not get any feedback before it is too late. The gain controls how fast the network is filled at the estimated capacity. We have shown that it is possible to adapt both variables in response to variability in the estimates from previous rounds as an indication of the variability of existing flows.

Regular slow start has to increase the amount of data to seek higher capacities. Paced Chirping partially decouples the amount of data used and seeking capacity. It can get a better estimate without increasing the number of packets. This is significant because it means that if Paced Chirping is uncertain about the estimate it can keep the same amount of data and still improve the estimate.

Section 11.4 showed how the relation between the marking threshold and the BDP affected performance of regular slow start. It is not possible to configure an AQM to be optimal for all flows passing through if the flows have different RTTs.

In section 11.3 we saw that Paced Chirping can significantly improve FCT at the cost of higher queueing delay for small flows in a fully utilized network. Section 11.6 presented a naive adaption algorithm and a rerun of the FCT experiment. The result was reduced queueing delay, improved overall utilization marginally, and slightly worse FCT.

12.2 Future work

Experiments

We have not evaluated flows with different RTTs. A flow with a long RTT reacts more slowly than a flow with a short RTT. So we expect to see different behaviour with different RTTs.

Networks with multiple hops have to be considered because we might see more conservative estimates from the initial packet trains. The average dispersion rate (ADR) discussed in section 5.2 is determined by all of the links on a path and not just the bottleneck. Flows starting in a utilized network might get a lower initial rate estimate and be less aggressive than when there is only one hop.

Goal of flow start

A question that remains is how aggressive a new slow should be. Should it aim at reaching the fair share, or something else, and how much queue delay should it sacrifice to reach it faster? This has to be debated and explored.

Delayed ack option

Paced chirping needs delayed acks to be turned off because it currently does not have a mechanism to handle delayed acks.

Changing the heuristic in the Linux kernel is not sufficient. We think that the simplest solution is to have a bit of the TCP header communicate the preferred ack-policy. This way the sender can ask the receiver to not use delayed acks in flow start, and keep using delayed acks in steady state.

If we want to use chirps in congestion avoidance a heuristic has to be really sophisticated to operate correctly and the sender has to assume that the heuristic works. It is much better to have an explicit signal.

Dynamics during congestion avoidance

Paced Chirping should also be applicable in congestion avoidance whenever the available capacity suddenly increases or a cross-traffic flow departs. In steady state DCTCP induces 2 marks per round trip, so spare capacity could be detected by lack of regular CE-marks over one or two round trips. In such cases, switching from the additive increase of 1 packet per RTT to a Paced-Chirping-like algorithm could improve responsiveness to dynamics tremendously. This would require delayed acks to be turned on and off during congestion avoidance as well as during flow start.

Paced Chirping is not yet used after idle periods, but this should be easy to implement. This could improve performance of applications with this kind of behaviour.

Adaptive gain and geometry

The chirp geometry and gain can both be adapted during flow start to improve estimate accuracy and reduce queue delay. It is not clear, at this moment, what the best approach is. We think that the stability in the estimate can be a good candidate.

This can reduce queueing delay because of overshoot and improve the accuracy of the estimate.

One issue with decreasing the geometry is that the chirp become more susceptible to pacing variability. The analysis does not currently handle the case where the inter-send time increases regularly. An increase in inter-send time might allow the queue delay to relax enough for the analysis to overestimate.

Improving analysis

Paced chirping is purely delay based. Finding a way to make use of ECN signals might improve the algorithm. It can certainly be used to reduce queue delay, but it is not clear what the best approach is.

To improve the analysis accuracy of the chirps we would like to replace the RTT with the one-way delay.

Termination condition

Currently Paced Chirping exits once the duration of a round becomes greater than the round-trip time. It ignores the information from chirps

received after it has terminated. The information from these chirps could prove useful.

If we could determine that a flow has converged based on ECN-signal when the network is utilized the termination condition could be based on this.

Continuous adjustments

Currently the number of chirps is only updated once per RTT and the algorithm operated in rounds. It might be beneficial to have continuous adjustment.

Kernel changes and Pacing implementation

Changing the kernel code to accept time units instead of rates for the inter-send gaps would improve the run-time performance.

We also want to investigate different options for communicating the inter-send gaps to the pacing implementation. The linked list that is currently used is flexible, but it is probably not the best option. One option is to have a callback. This would remove the need for scheduling chirps ahead of time, thus use the most recent information. Another is to do the calculation in the pacing implementation.

Resource limited devices

We had to turn power management capabilities off on the server CPUs to get accurate enough inter-send gaps for Paced Chirping to work properly. This would be a difficult tradeoff on smart-phones and laptops.

This issue has to be explored further.

Appendices

Appendix A

Code

All code and its documentation should be available at https://github.uio.no/joakimmi/MasterCode and https://github.com/JoakimMisund/PacedChirping. The former might become unavailable when – if – I leave the university.

Bibliography

[1] John Nagle. *Congestion Control in IP/TCP Internetworks*. Request for Comments 896. RFC Editor, Jan. 1984. 9 pp. URL: https://rfc-editor.org/rfc/rfc896.txt.

[2] Van Jacobson. "Congestion Avoidance and Control". In: *Proc. ACM SIGCOMM'88 Symposium, Computer Communication Review* 18.4 (Aug. 1988), pp. 314–329. URL: http://doi.acm.org/10.1145/52324.52356.

[3] Robert T. Braden. *Requirements for Internet Hosts - Communication Layers*. Request for Comments 1122. RFC Editor, Oct. 1989. 116 pp. URL: https://rfc-editor.org/rfc/rfc1122.txt.

[4] Sally Floyd and Van Jacobson. "Random Early Detection Gateways for Congestion Avoidance". In: *IEEE/ACM Trans. Netw.* 1.4 (Aug. 1993), pp. 397–413. DOI: 10.1109/90.251892. URL: http://dx.doi.org/10.1109/90.251892.

[5] Lawrence S. Brakmo and Larry L. Peterson. "TCP Vegas: End-to-End Congestion Avoidance on a Global Internet." In: *IEEE Journal on Selected Areas in Communications* 13.8 (Oct. 1995), pp. 1465–80. URL: http://ieeexplore.ieee.org/iel1/49/9740/00464716.pdf?arnumber=464716.

[6] Sally Floyd et al. *TCP Selective Acknowledgment Options*. Request for Comments 2018. RFC Editor, Oct. 1996. 12 pp. URL: https://rfc-editor.org/rfc/rfc2018.txt.

[7] Dr. Joseph D. Touch. *TCP Control Block Interdependence*. Request for Comments 2140. RFC Editor, Apr. 1, 1997. 11 pp. URL: https://rfc-editor.org/rfc/rfc2140.txt.

[8] Frank P. Kelly, Aman K. Maulloo, and David K. H. Tan. "Rate control for communication networks: shadow prices, proportional fairness and stability". In: *Journal of the Operational Research Society* 49.3 (1998), pp. 237–252. URL: http://www.statslab.cam.ac.uk/~frank/rate.html.

[9] A. Aggarwal, S. Savage, and T. Anderson. "Understanding the performance of TCP pacing". In: *Proceedings IEEE INFOCOM 2000.* Vol. 3. Mar. 2000, pp. 1157–1165. DOI: 10.1109/INFCOM.2000.832483. URL: https://ieeexplore.ieee.org/abstract/document/832483/.

[10] Joanna Kulik et al. *A simulation study of paced TCP*. Technical Report. Jan. 2000, p. 26. URL: https://ntrs.nasa.gov/search.jsp?R=20000021225.

[11] Jitendra Padhye, Sally Floyd, and Mark J. Handley. *TCP Congestion Window Validation*. Request for Comments 2861. RFC Editor, June 2000. 11 pp. URL: https://rfc-editor.org/rfc/rfc2861.txt.

[12] Matthew Podolsky et al. *An Extension to the Selective Acknowledgement (SACK) Option for TCP*. Request for Comments 2883. RFC Editor, July 2000. 17 pp. URL: https://rfc-editor.org/rfc/rfc2883.txt.

[13] Jonathan Stone and Craig Partridge. "When the CRC and TCP Checksum Disagree". In: *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '00. Stockholm, Sweden: ACM, 2000, pp. 309–319. DOI: 10.1145/347059.347561. URL: http://doi.acm.org/10.1145/347059.347561.

[14] Sally Floyd, Dr. K. K. Ramakrishnan, and David L. Black. *The Addition of Explicit Congestion Notification (ECN) to IP*. Request for Comments 3168. RFC Editor, Sept. 2001. 63 pp. URL: https://rfc-editor.org/rfc/rfc3168.txt.

[15] Dina Katabi, Mark Handley, and Charlie Rohrs. "Congestion Control for High Bandwidth-Delay Product Networks". In: *Proc. ACM SIGCOMM'02, Computer Communication Review* 32.4 (Oct. 2002), pp. 89–102. URL: http://conferences.sigcomm.org/sigcomm/2002/papers/xcp.pdf.

[16] Vinay J. Ribeiro et al. "pathChirp: Efficient Available Bandwidth Estimation for Network Paths". In: *Passive and Active Measurement Workshop (PAM'03)*. 2003. URL: https://scholarship.rice.edu/handle/1911/20252.

[17] Srisankar S. Kunniyur. "AntiECN Marking: A Marking Scheme for High Bandwidth Delay Connections". In: *Proc. ICC'03*. IEEE. May 2003. URL: http://repository.upenn.edu/cgi/viewcontent.cgi?article=1053&context=ese_papers.

[18] Constantinos Dovrolis, Parameswaran Ramanathan, and David Moore. "Packet-dispersion techniques and a capacity-estimation methodology". In: *IEEE/ACM Transactions On Networking* 12.6 (2004), pp. 963–977. URL: http://ieeexplore.ieee.org/abstract/document/1369287/.

[19] D Leith. *Linux TCP implementation issues in high-speed networks*. Technical Report. 2004. URL: http://www.hamilton.ie/net/LinuxHighSpeed.pdf.

[20] Nandita Dukkipati et al. "Processor Sharing Flows in the Internet". In: *Proc. International Workshop on QoS (IWQoS'05)*. June 2005. URL: http://yuba.stanford.edu/rcp/RCP-IWQoS.pdf.

[21] Yong Xia et al. "One more bit is enough". In: *Proc. ACM SIGCOMM'05, Computer Communication Review* 35.4 (2005), pp. 37–48. DOI: 10.1145/1080091.1080098. URL: https://doi.acm.org/10.1145/1080091.1080098.

[22] D.J.Leith B.Even Y.Li. "Evaluating the Performance of TCP Stacks for High-Speed Networks". In: *PFDLnet2006*. ACM. 2006. URL: http://eprints.maynoothuniversity.ie/1781/.

[23] Bob Briscoe. "Flow Rate Fairness: Dismantling a Religion". In: *SIGCOMM Comput. Commun. Rev.* 37.2 (Mar. 2007), pp. 63–74. DOI: 10.1145/1232919.1232926. URL: http://doi.acm.org/10.1145/1232919.1232926.

[24] Sally Floyd et al. *Quick-Start for TCP and IP*. Request for Comments 4782. (Status: Experimental). RFC Editor, Jan. 2007. URL: http://www.icir.org/floyd/quickstart.htmlt.

[25] Yee-Ting Li, Douglas Leith, and Robert N. Shorten. "Experimental Evaluation of TCP Protocols for High-speed Networks". In: *IEEE/ACM Trans. Netw.* 15.5 (Oct. 2007), pp. 1109–1122. DOI: 10.1109/TNET.2007.896240. URL: http://dx.doi.org/10.1109/TNET.2007.896240.

[26] Sangtae Ha, Injong Rhee, and Lisong Xu. "CUBIC: a new TCP-friendly high-speed TCP variant". In: *SIGOPS Operating Systems Review* 42.5 (July 2008), pp. 64–74. DOI: 10.1145/1400097.1400105. URL: http://doi.acm.org/10.1145/1400097.1400105.

[27] M. Allman, V. Paxson, and E. Blanton. *TCP Congestion Control*. Request for Comments 5681. RFC Editor, Sept. 2009. URL: http://www.ietf.org/rfc/rfc5681.txt.

[28] Ilpo Järvinen and Markku Kojo. "Improving processing performance of linux tcp sack implementation". In: *International Workshop on Protocols for Future, Large-Scale and Diverse Network Transports (PFLDNeT)(May 2009)*. 2009.

[29] Vishnu Konda and Jasleen Kaur. "RAPID: Shrinking the Congestion-control Timescale". In: *Proc. IEEE Conference on Computer Communications (Infocom'09)*. IEEE. Apr. 2009. DOI: 10.1109/INFCOM.2009.5061900. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5061900.

[30] Aleksandar Kuzmanovic et al. *Adding Explicit Congestion Notification (ECN) Capability to TCP's SYN/ACK Packets*. RFC5562. RFC Editor, June 2009. URL: https://tools.ietf.org/html/rfc5562.

[31] Mohammad Alizadeh et al. "Data Center TCP (DCTCP)". In: *SIGCOMM Comput. Commun. Rev.* 40.4 (Aug. 2010), pp. 63–74. DOI: 10.1145/1851275.1851192. URL: http://doi.acm.org/10.1145/1851275.1851192.

[32] Mirja Kühlewind and Bob Briscoe. "Chirping for Congestion Control - Implementation Feasibility". In: *Proc. Int'l Wkshp on Protocols for Future, Large-scale & Diverse Network Transports (PFLDNeT'10)*. Nov. 2010. URL: http://pfld.net/2010/paper/session2-3.pdf.

[33] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. "Analysis of DCTCP: Stability, Convergence, and Fairness". In: *Proc. ACM SIGMETRICS'11*. San Jose, California, USA: ACM, 2011, pp. 73–84. DOI: 10.1145/1993744.1993753. URL: http://doi.acm.org/10.1145/1993744.1993753.

[34] Jim Gettys and Kathleen Nichols. "Bufferbloat: Dark Buffers in the Internet". In: *Queue* 9.11 (Nov. 2011), 40:40–40:54. DOI: 10.1145/2063166.2071893. URL: http://doi.acm.org/10.1145/2063166.2071893.

[35] Matt Sargent et al. *Computing TCP Retransmission Timer*. Request for Comments 6298. RFC Editor, June 2011. 11 pp. URL: https://rfc-editor.org/rfc/rfc6298.txt.

[36] Mohammad Alizadeh et al. "Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center". In: *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*. Apr. 2012. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/alizadeh.

[37] Jonathan corbet. *TCP small queues*. July 17, 2012. URL: https://lwn.net/Articles/507065/ (visited on 03/12/2018).

[38] Fernando Gont and Steven Bellovin. *Defending against Sequence Number Attacks*. Request for Comments 6528. RFC Editor, Feb. 2012. 12 pp. URL: https://rfc-editor.org/rfc/rfc6528.txt.

[39] Richard Scheffenegger, Mirja Kühlewind, and Brian Trammell. *Additional negotiation in the TCP Timestamp Option field during the TCP handshake*. Internet Draft draft-scheffenegger-tcpm-timestamp-negotiation-05. (work in progress). Internet Engineering Task Force, Oct. 2012. URL: https://datatracker.ietf.org/doc/html/draft-scheffenegger-tcpm-timestamp-negotiation-05.

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

[41] Mirja Kühlewind, Sebastian Neuner, and Brian Trammell. "On the State of ECN and TCP Options on the Internet". In: *Proc. 14th Int'l Conf. on Passive and Active Measurement (PAM 2013)*. Springer, Mar. 2013, pp. 135–144. DOI: 10.1007/978-3-642-36516-4_14. URL: http://dx.doi.org/10.1007/978-3-642-36516-4_14.

[42] Richard Scheffenegger, Mirja Kühlewind, and Brian Trammell. *Encoding of Time Intervals for the TCP Timestamp Option*. Internet Draft draft-trammell-tcpm-timestamp-interval-01. (work in progress). Internet Engineering Task Force, July 2013. URL: https://datatracker.ietf.org/doc/html/draft-trammell-tcpm-timestamp-interval-01.

[43] David Borman et al. *TCP Extensions for High Performance*. Request for Comments 7323. RFC Editor, Sept. 2014. 49 pp. URL: https://rfc-editor.org/rfc/rfc7323.txt.

[44] Yuchung Cheng et al. *TCP Fast Open*. Request for Comments 7413. RFC Editor, Dec. 2014. URL: http://www.rfc-editor.org/rfc/rfc7413.txt.

[45] Taylor IoT Kidd. *Power Management States: P-States, C-States, and Package C-States*. Apr. 17, 2014. URL: https://software.intel.com/en-us/articles/power-management-states-p-states-c-states-and-package-c-states (visited on 03/12/2018).

[46] Radhika Mittal et al. "Recursively Cautious Congestion Control". In: *NSDI*. 2014, pp. 373–385. URL: https://www.usenix.org/system/files/conference/nsdi14/nsdi14-paper-mittal.pdf.

[47] Mike Belshe, Roberto Peon, and Martin Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. Request for Comments 7540. RFC Editor, May 2015. 96 pp. URL: https://rfc-editor.org/rfc/rfc7540.txt.

[48] Martin Duke et al. *A Roadmap for Transmission Control Protocol (TCP) Specification Documents*. Request for Comments 7414. RFC Editor, Feb. 2015. 57 pp. URL: https://rfc-editor.org/rfc/rfc7414.txt.

[49] Gorry Fairhurst, Arjuna Sathiaseelan, and Raffaello Secchi. *Updating TCP to Support Rate-Limited Traffic*. Request for Comments 7661. RFC Editor, Oct. 2015. 21 pp. URL: https://rfc-editor.org/rfc/rfc7661.txt.

[50] Radhika Mittal et al. "TIMELY: RTT-based Congestion Control for the Datacenter". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. London, United Kingdom: ACM, 2015, pp. 537–550. DOI: 10.1145/2785956.2787510. URL: http://doi.acm.org/10.1145/2785956.2787510.

[51] Koen De Schepper et al. "PI^2 : A Linearized AQM for both Classic and Scalable TCP". In: *Proc. ACM CoNEXT 2016*. ACM, Dec. 2016, pp. 105–119. ISBN: 978-1-4503-4297-1.

[52] Marcelo Bagnulo and Bob J. Briscoe. *ECN++: Adding Explicit Congestion Notification (ECN) to TCP Control Packets*. Internet-Draft. (Work in Progress). Internet Engineering Task Force, Oct. 2017. 37 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-tcpm-generalized-ecn/.

[53] Stephen Bensley et al. *Data Center TCP (DCTCP): TCP Congestion Control for Data Centers*. Request for Comments 8257. RFC Editor, Oct. 2017. 17 pp. URL: https://rfc-editor.org/rfc/rfc8257.txt.

[54] Bob J. Briscoe, Mirja Kühlewind, and Richard Scheffenegger. *More Accurate ECN Feedback in TCP*. Internet-Draft draft-ietf-tcpm-accurate-ecn-04. (Work in Progress). Internet Engineering Task Force, Oct. 30, 2017. 43 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-tcpm-accurate-ecn/.

[55] Yuchung Cheng, Neal Cardwell, and Nandita Dukkipati. *RACK: a time-based fast loss detection algorithm for TCP*. Internet-Draft draft-ietf-tcpm-rack-02. (Work in Progress). Internet Engineering Task Force, Mar. 13, 2017. 22 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-tcpm-rack.

[56] Koen De Schepper et al. *DualQ Coupled AQM for Low Latency, Low Loss and Scalable Throughput*. Internet Draft draft-ietf-tsvwg-aqm-dualq-coupled-01. (Work in Progress). Internet Engineering Task Force, July 2017. URL: http://tools.ietf.org/html/draft-ietf-tsvwg-aqm-dualq-coupled.

[57] Lars Eggert, Gorry Fairhurst, and Greg Shepherd. *UDP Usage Guidelines*. RFC 8085. Mar. 2017. DOI: 10.17487/RFC8085. URL: https://rfc-editor.org/rfc/rfc8085.txt.

[58] Dr. Joseph D. Touch et al. *TCP Control Block Interdependence*. Internet-Draft draft-touch-tcpm-2140bis-03. (Work in Progress). Internet Engineering Task Force, Jan. 12, 2017. 23 pp. URL: https://datatracker.ietf.org/doc/html/draft-touch-tcpm-2140bis.

[59] Q. Yin, J. Kaur, and F. D. Smith. "TCP Rapid: From theory to practice". In: *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. May 2017, pp. 1–9. DOI: 10.1109/INFOCOM.2017.8057179.

[60] Bob J. Briscoe, Koen De Schepper, and Marcelo Bagnulo. *Low Latency, Low Loss, Scalable Throughput (L4S) Internet Service: Architecture*. Internet-Draft draft-ietf-tsvwg-l4s-arch-02. (Work in Progress). Internet Engineering Task Force, Mar. 2018. 32 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-tsvwg-l4s-arch.

[61] Dominik Brodowski et al. *CPU frequency and voltage scaling code in the Linux(TM) kernel*. URL: https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt (visited on 03/12/2018).