Implementing a timer-based TCP congestion control in the Linux kernel

Bjelke, Christoffer
Limi, Andreas

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

Department of Informatics
The Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Autumn 2023

Implementing a timer-based TCP congestion control in the Linux kernel

Bjelke, Christoffer
Limi, Andreas

Implementing a timer-based TCP congestion control in the Linux kernel

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

Printed: Reprosentralen, University of Oslo

Abstract

This thesis presents the design, development and evaluation of a novel pluggable timer-based TCP congestion control module in the Linux kernel. The conceptual basis for the congestion control is derived from an unpublished paper at the time of writing. The resulting implementation is fundamentally different from traditional TCP congestion control, as it does not exclusively rely on ACK-clocking or a congestion window. Instead, it is based on *time*, where all packet transmissions occur as a consequence of a timer expiration in Slow Start and Congestion Avoidance. The timeout values are calculated from the current sending rate and congestion control algorithm. In essence, this process interpolates the transmission behavior of traditional TCP congestion control, in addition to meticulously distributing packets across time, yielding smooth traffic patterns.

The evaluation of the implementation was conducted in an isolated single flow environment. The findings provided insights into its capabilities and limitations, highlighting areas of further research on implementing timer-based congestion control.

Preface

Acknowledgements

We would like to extend our gratitude towards our supervisors, Michael Welzl and Safiqul Islam, for their guidance and feedback throughout the project. Their knowledge and insights, both in the field of TCP and in the field of research, have been invaluable to us.

We would like to thank the Department of Informatics at the University of Oslo for the opportunity to carry out this project. The department's facilities and welcoming atmosphere have played a significant role in our academic pursuits and have been central to forming strong friendships and a supportive network.

Getting started with Linux kernel development can be a daunting task. We would like to thank Joakim Misund for an insightful conversation in the early stages of the project.

During our time working on this project, we have benefited greatly from the support, discussions, and feedback from our significant others, family, friends, and fellow students.

Contents

Chapter 1

Introduction

1.1 Problem statement

Congestion control is a general scheme to maximize throughput and fairness while minimizing delay when sending data over a network. To achieve this, a congestion control algorithm must find the appropriate rate at which to send data into the network. To not congest the network with bursty traffic, traffic may instead be *paced* to achieve better performance [1]. Pacing involves delaying the transmission of packets to add space between the packets once they get to the wire.

The Transmission Control Protocol (TCP) provides the logic for congestion control in the Linux kernel. To facilitate this, Linux TCP possesses pacing capabilities. This can be generalized to two different approaches: pacing in the scheduler through a *Queuing Disclipline* like *FQ/Pacing*, or through internal pacing done by the TCP stack. FQ/Pacing has good rate conformance at the cost of high CPU utilization [2]. Internal pacing gives congestion control modules a higher degree of control and more precise RTT estimations, in addition to being the default pacing approach in the absence of a Queuing Discipline [3].

Both of these approaches are viable, however, they rely on TCP mechanisms like *ACK-clocking* and maintenance of a congestion window, in addition to being rate-based. This reliance may increase the complexity of Linux TCP while limiting the capabilities of congestion control modules for fine-grained control.

1.2 Contributions

In this master thesis we have implemented and evaluated a pluggable timer-based TCP congestion control module in the Linux kernel. The implementation featured a novel approach to achieve fine grained per-packet pacing in both Slow Start and Congestion Avoidance. The event-driven nature of the implementation allows for fine grained control from

the pluggable congestion control module.

The congestion control logic is devised from the Timer-Based TCP (TBTCP) research paper, which is unpublished at the time of writing. We have extracted and modified parts of the logic from this paper, in order to implement a simplified timer-based congestion control. This simplification only concerns itself with the Slow Start and Congestion Avoidance phases of TCP.

1.3 Research questions

In this thesis, we implement and explore the viability of a pluggable timer-based TCP congestion control module in the Linux kernel.

Replacing established TCP mechanisms

How feasible is it to replace established TCP mechanisms in the Linux kernel, such as ACK-triggered transmissions and the reliance on a congestion window? This is a necessity for our implementation, as it requires us to remove certain standard behaviors of TCP. Can this be done in a backwards compatible manner, such that other congestion control modules work as intended?

Pacing with a High-resolution timer

Can Linux TCP congestion control be *fully* paced in Slow Start and Congestion Avoidance with the use of a single timer? By fully paced, we mean per-packet pacing, where each packet transmission is initiated as a consequence of timer expiration. How does this scale when bandwidth is high?

Giving control to TCP modules

Is it viable to move congestion control logic from the TCP stack to pluggable congestion control modules? In Linux TCP, only a subset of TCP functionality is exposed to congestion control modules through an API. How viable is it to extend this API with new functionality, such as the ability to initiate packet transmissions.

A potential benefit to this is that it may provide developers with more flexibility when developing congestion control modules, such that functionality can be tailored to specific networks and applications.

Delegate recovery handling to TCP

Is it feasible to delegate recovery handling to the underlying TCP stack? To achieve this, we must notify the congestion control module of TCP state transitions. When TCP is in recovery, transmissions should not be initiated from the congestion control module. Additionally, when TCP transitions from recovery to Congestion Avoidance, the congestion control module should start at an appropriate sending rate.

Performance

What performance characteristics can be expected from a timer-based congestion control module? What is the maximum throughput we can achieve from a solely timer-based congestion control module, considering it is per-packet paced?

1.4 Organization

The thesis is organized as follows: the first part of chapter 2 gives the reader relevant background information on congestion control, TCP and pacing. The second part details how TCP is implemented in Linux. An overview of this part is vital to understanding the later chapter on implementation.

Chapter 3 specifies the design and core functions of TBTCP. This specification is what we will base our Linux implementation on. Chapter 4 details the actual implementation of TBTCP in the Linux kernel; both the code and the challenges we faced in the process.

Chapter 5 evaluates the implementation. Here, we analyze the core functionality and behavior of the implementation. The thesis is concluded with Chapter 6, where we summarize our findings and address the research questions.

1.5 Collaboration

This thesis is a collaborative endeavor by two individuals. As such, the majority of the work detailed in this thesis is the result of a collaborative effort. However, the general responsibilities can be divided in the following manner: Christoffer Bjelke has implemented the logic related to ACK-handling and post-recovery synchronization. Andreas Limi has implemented the logic related to packet transmissions and enhancing the congestion control module API.

Chapter 2

Background

In this chapter, we will give an overview of concepts that the reader should be aware of to follow along in this thesis. In section 2.1 we will start by detailing the fundamentals of congestion control and the philosophy behind it. We then move on to TCP in section 2.2, where we detail the fundamentals as well as relevant congestion control algorithms. section 2.3 is dedicated to pacing, as it is an essential part of this thesis. Lastly, we detail the Linux TCP implementation in section 2.4.

2.1 Congestion

The internet has become increasingly diverse since the days of the ARPANET. As a result, the pursuit of minimizing delay and maximizing bandwidth utilization is a frequent topic of research and discussion. The delay is influenced mainly by two factors: distance and queues. The distance component considers the time it takes for a signal to travel from the sender to the receiver, taking into account the speed of light and the specific route the packet follows through the Internet. Even though communication over the Internet is mainly connection-oriented, it does not mean that every single packet in a connection traverses the same path. This is instead decided by packet-switching schemes deployed in nodes throughout the Internet.

A node can experience *congestion* when the rate at which packets arrive exceeds their departure rate. This occurs when the ingress or egress links of the node are saturated; the term *bandwidth* denotes how much data can physically "fit" on a given link per time unit. Instead of discarding the packets, they may instead be buffered at the node. This helps avoid loss of data at the cost of increasing delay. Depending on the node's buffering capacities, excessive queues may form; this phenomenon is called *bufferbloat* [4]. An underlying cause of this is cheap memory in combination with a priority on avoiding packet loss. This has the potential to increase round-trip times from the range of milliseconds to seconds. Queues are

difficult to manage, as their existence can only be attributed to increased delays as seen from the context of the transport layer.

The link on a given path with the lowest bandwidth is commonly called the *bottleneck link*. The bandwidth of the bottleneck link describes the maximum rate at which data can be delivered without increasing delay times. If we multiply the path delay and the bandwidth, we can calculate the *bandwidth-delay product* (BDP) for an arbitrarily complex path. This denotes the maximum amount of data that can be in flight at any time, spread across the entirety of the path. From this, we can conclude that a path can only have maximum utilization if the in-flight data equals the BDP. However, this does not imply that a sender can transmit BDP amounts of data in a single burst, as this would cause a queue to form at the bottleneck link. To remedy this, packets can be *paced* to distribute traffic along a path.

Congestion control is a scheme developed to maximize throughput and fairness while minimizing delay. In the Internet it is deployed on the transport layer, meaning the details of the physical and link layers are abstracted. Because of this, congestion control algorithms must make assumptions for sending rate, delays, BDP and bottlenecks based on indicators like the arrival of acknowledgments and losses. Congestion control is essential for maintaining reliable, fair, and performant communication over networks.

2.2 The Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP) is a communication protocol deployed in end-hosts to facilitate an in-order, reliable, byte-stream for communication between applications. It is often paired with the Internet Protocol (IP) suite, which is often referred to as TCP/IP. This section will give the reader an introduction to TCP, starting with an overview based on RFC 9293[1] [5]. We will then detail the congestion control aspect of TCP based on RFC 5681 [6]. Additionally, we will discuss enhancements that can be implemented to improve the performance of TCP.

2.2.1 Overview

TCP is a transport layer protocol. On a high level, it facilitates communication between user applications over networks. To enable this, user applications can communicate with TCP programmatically through an Application Programming Interface (API) using sockets. This API consists of actions an application might need, like sending and receiving data. Through this abstraction, user applications can be developed independently of the network infrastructure below them. This concept of layering, where each

[1] This RFC is the base specification of TCP with a focus on the required mechanisms all TCP implementations should support. It references companion documents for mechanisms and algorithms outside of this scope.

layer provides a set of services to the layer above it, provides a framework to build new functionality without the need to be aware of the underlying implementation.

Below TCP is the Internet Protocol (IP). IP provides an *unreliable*, *best effort* service for routing data through networks. It achieves this through the use of *IP addresses*, which is a global addressing system for hosts in networks. IP is handed data, or *segments* from TCP, to transfer from a source to a destination host. IP then encapsulates these segments with its own *headers*. An IP header consists of a source and destination IP, along with other metadata like the transport protocol used, and a checksum to detect corruption. Through the use of these headers, nodes throughout the network can route the packet by determining its next *hop* along the forwarding path until it reaches the destination host.

As mentioned, the role of TCP is to enable reliable communication between hosts. TCP achieves this despite the unreliable protocols *below* it. Below are some of the strategies TCP employs:

Interface to applications In Linux, applications can interact with TCP through the use of *sockets*. Sockets act as an API to the underlying stack by providing a set of functionality that the application can invoke. Also called POSIX- or Berkeley-sockets, they originated in 1983 along with the BSD operating system [7]. They have since become the standard for implementing application APIs to interact with TCP/IP. A socket accepts a set of different parameters that specify the protocols to use. To communicate between different hosts, the `domain` of `AF_INET` and a `protocol` of `0` inform the underlying stack to use IP. Pair this with a `type` of `SOCK_STREAM` and the result is a socket supporting a sequenced, reliable, two-way, connection-based stream [8]. This equates to a TCP/IP socket on traditional Linux systems.

Connection oriented TCP is connection-oriented in the sense that a connection between a *client* and a *server* needs to be established before data can be sent[2]. A connection can be described as a data structure containing all state related to the connection. To establish a connection between a client and a listening server, a *three-way handshake* is required. The client initiates this handshake by sending a request to the port that the server is listening on. This connection is maintained until the data transfer and teardown process is finished.

Data streaming When an application wants to send data, it writes data to a TCP socket. The TCP stack will partition this stream of bytes into smaller segments to be transmitted.

[2]Throughout this thesis, we will denote the sender of data as the client, while the server is the receiver. This is the opposite of how traffic conventionally flows in today's internet.

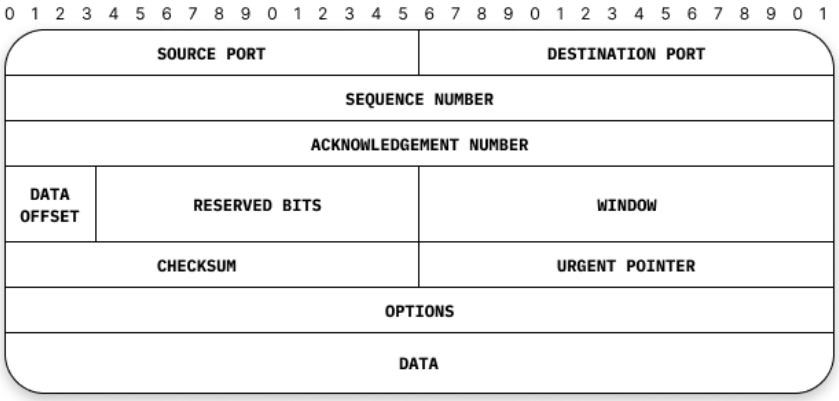


Figure 2.1: An illustrated example of a TCP header. The top row of numbers indicates an index in a bit array, which can be used to determine the field sizes.

Headers TCP employs headers to transmit metadata about a segment or the connection itself. Headers are a collection of bits that can be parsed in a specific manner to identify the individual fields. Among these fields are source and destination ports, sequence number, the receiver window and a collection of flags. Flags, or control bits, are individual bits that indicate the type or purpose of segments. Figure 2.1 provides an overview of what a TCP header consists of.

Reliable The reliability aspect of TCP guarantees recovery from losses, corruption, duplication and out-of-order delivery. TCP accomplishes this by maintaining a sequence number space that maps to every byte sent. When a server receives data, it sends an *acknowledgment* back to the client. In the basic sense, an acknowledgment contains the highest cumulative sequence number the server has successfully received in order up to that moment in time. The client can use this information to determine which packets need to be retransmitted.

In-order delivery Although packets may not arrive at the server in-order, data is still guaranteed to be pushed in-order to applications. This is essential, as applications may rely on this guarantee.

Flow control TCP prevents congestion at the server by implementing a *receive window*. This represents the sequence numbers that the receiver is willing to accept. The server advertises this to the client in the initial handshake and throughout the transmission lifetime. The receive window ensures that the client does not overrun the server with data.

Segmentation

Segmentation refers to the process of dividing a stream of data from the application to TCP segments. As indicated in the overview, the data written from the application may not equate to the amount of TCP segments transferred. TCP may even delay sending to combine data from multiple `SEND` calls from the application. On the receiver side, segments can be buffered to batch writes to the application. To prevent this, the sending host can set the `PUSH` flag in its `SEND` calls to instruct the receiver to push data to the application promptly.

There are several factors to consider when determining the sizing of segments. Sending larger segments limits the number of segments in-flight, effectively limiting the amount of TCP headers needed which results in less processing needed at the hosts. Sending smaller segments results in limiting delayed transmissions and avoiding fragmentation. To determine the size of segments, TCP uses a negotiated MSS and Path MTU Discovery (PMTUD). To prevent sending smaller segments than the determined MSS, TCP may buffer data until a full MSS-sized segment can be sent. This is determined by the Nagle Algorithm [9].

Maximum Segment Size The Maximum Segment Size (MSS), is the upper bound of the size of a TCP segment that the client is allowed to send [10]. It is negotiated during the handshake, where the server suggests a value.

Maximum Transmission Unit Unlike the MSS, which determines what the server can receive, the Maximum Transmission Unit (MTU) denotes the largest segment size that can be transferred across the network path while avoiding fragmentation [11]. The MTU is found by a process called Path MTU Discovery (PMTUD), which involves probing the network with gradually smaller segments until fragmentation does not take place. This algorithm relies on routers returning an ICMP message if a segment is too large to forward without fragmentation occurring [11]. An Internet Control Message Protocol (ICMP) message is a way for communicating hosts to provide feedback when problems arise [12]. An extension of PMTUD that does not rely on ICMP messages is the Packetization Layer Path MTU (PLPMTUD). PLPMTUD does this by injecting gradually larger packets into the network and increases the MTU if the packets are successfully delivered [13]. This removes the reliance on ICMP messages, as PLPMTUD reacts to the loss itself, rather than ICMP messages.

Nagle Algorithm The Nagle algorithm is an optional mechanism for delaying transmission at the client. When in use, TCP will delay sending data until either all unacknowledged data has been acknowledged, or there is enough data buffered to send a full-sized TCP segment [14].

Retransmission Timeout

A Retransmission Timeout (RTO) is a required mechanism that retransmits lost data when the server fails to respond with an acknowledgment. It is implemented by queuing a timer and retransmitting data when the timer expires. How long to enqueue the timer is based on the *smoothed round-trip time* (SRTT) and the *round-trip time variation* (RTTVAR). In essence, the SRTT is a Weighted Moving Average, where recent RTT measurements are given more weight than previous ones. The RTTVAR is a weighted average of the difference between the SRTT and the actual RTT. Initially, the RTO timer is enqueued for 1 second. When RTT measurements are being registered, the time to enqueue the timer is found by the following formula [15]:

$$RTO = SRTT + \max(G, K \times RTTVAR) \tag{2.1}$$

Where G is the clock granularity and K is equal to a constant value of 4.

2.2.2 Congestion Control

In this section, we will discuss TCP congestion control in detail. We will start by defining the necessary concepts and mechanisms for implementing congestion control, and then proceed on to describe the required algorithms.

Receiver window The receiver window (*rwnd*) is a server-side variable that denotes how much data the server is willing to receive. This can be assumed to be related to the available buffer space at the server and is advertised on every acknowledgment.

Congestion window The congestion window (*cwnd*) is a mechanism that limits how much data the client can send to prevent network congestion. As acknowledgments are being received, the congestion window expands, allowing the client to send more data. It is further limited by an upper bound of the *rwnd*, effectively limiting the amount of inflight data by the formula:

$$quota = min(rwnd, cwnd) \tag{2.2}$$

Upon loss events, the congestion window is usually reduced to not induce further congestion. In RFC 5681 [6] the initial value of this variable was required to be between 2 and 4 segments, based on the MSS. However, modern implementations (including Linux [16]) often increase the upper bound to 10, as proposed in RFC 6928 [17].

Acknowledgment clocking TCP relies on the arrival of incoming acknowledgments. This enables both expanding the *cwnd* and the progression of the state machine. When an acknowledgment arrives, TCP will run it through various validation processes in addition to checking if data can be transmitted. This process is essential to maintain the so-called "ACK clock".

To facilitate this, the server should respond with acknowledgments containing the highest cumulative sequence number it has received in response to out-of-order packets. When multiple acknowledgments of the same sequence number are sent, they are referred to as Duplicate Acknowledgements (DupACKs).

In-flight data A client-side variable, *FlightSize*, is introduced to denote the amount of unacknowledged data in the network. Conceptually, this can be thought of as the actual data in flight. However, it is unrealistic to estimate this accurately in actual implementations.

Algorithms

The congestion control aspect of TCP can be condensed into four algorithms; Slow Start, Congestion Avoidance, Fast Retransmit and Fast Recovery. RFC 5681 [6] currently specifies the standard of these algorithms.

Slow Start When a connection is initiated or suffers from heavy losses, the client must probe the network for available bandwidth. To achieve this, the Slow Start algorithm is used. A new variable, *ssthresh* is introduced, which determines when to use this algorithm by the formula $cwnd < ssthresh$. It is initiated to an arbitrarily high value when a connection is initialized and is reduced when congestion is experienced.

The client will initiate the Slow Start phase by sending as many segments as the *cwnd* allows. Upon the arrival of acknowledgments, the *cwnd* is increased by 1**MSS*. Contrary to its name, this is effectively an exponential increase of *cwnd* which facilitates a rapid increase of the sending rate to one that exceeds the bottleneck rate. When congestion is ultimately experienced, the *sstresh* and *cwnd* variables are reduced to progress TCP into Congestion Avoidance.

Congestion Avoidance Congestion Avoidance is the commencing phase after Slow Start, with a more conservative increase of *cwnd*. The congestion window can be increased by at most one *MSS* per RTT, or expressed as the result of $MSS * MSS/cwnd$ for every received acknowledgment of new data (which equates to roughly one *MSS* per RTT[3]).

[3]This is dependant upon the acknowledgment rate of the receiver. We assume delayed ACKs are disabled in this case.

Losses Earlier, we described RTO timeouts in section 2.2.1. Timeouts play a vital role in congestion control in terms of adjusting *ssthresh* and *cwnd*. Upon a timeout, *ssthresh* must be set according to the following rules:

1. If the lost segment has not yet been retransmitted:

$$ssthresh = max(FlightSize/2, 2 * MSS) \tag{2.3}$$

2. If the lost segment has been retransmitted in the past, *ssthresh* is held constant.

The *cwnd* on the other hand, should be set to 1**MSS*, to induce a Slow Start phase until it reaches the adjusted *ssthresh*.

Fast Retransmit Instead of waiting for an RTO timeout (which equates to at least an RTT), TCP can transition to a recovery phase on the arrival of DupACKs. DupACKs indicate that the server receives packets out-of-order, which in turn indicates either loss or reordering. After receiving three DupACKs, TCP retransmits the segment that is determined lost.

Fast Recovery After performing a Fast Retransmit, it is desirable to continue injecting new data into the network. This is under the assumption that packets continue leaving the network based on observed DupACKs.

The Fast Retransmit and Fast Recovery algorithms are coupled together and must be implemented according to the following rules:

1. Upon receiving the first two DupACKs, TCP should transmit a segment of new data for each DupACK, without expanding *cwnd* and *FlightSize*.
2. Upon receiving the third DupACK, *ssthresh* must be adjusted with an upper bound given by Equation 2.3. The segment which contains the lowest unacknowledged sequence number should be retransmitted. Lastly, the *cwnd* must be adjusted to $sstresh + 3 * MSS$. This is done to reflect the three DupACKs that have left the network.
3. On subsequent DupACKs received, the *cwnd* must be increased by *1*MSS*. New data should be sent if Equation 2.2 allows.
4. Recovery should terminate upon receiving an acknowledgment that acknowledges the lost segment, in addition to segments sent in the interval between the initial transmission of the lost segment and the arrival of the third DupACK. Additionally, *cwnd* must be set to *sstresh*. This is done to revert the aggressive expansion of *cwnd* that was done in steps 2 and 3.

2.2.3 Enhancements

Selective acknowledgements

The following contents are based on RFC 2018 [18] (TCP Selective Acknowledgement Options) and RFC 6675 [19] for how to integrate selective ACKs in fast retransmit/recovery.

TCP may perform poorly when multiple segments from the same window are lost. In the scenario where multiple segments are lost, the sender is only capable of detecting the loss of a single segment per RTT. There is simply no information in the acknowledgments that tells the sender which or how many segments are lost.

Selective Acknowledgements (SACKs) can solve this problem by telling the sender which segments have been lost so that only those segments need to be retransmitted. A SACK consists of blocks that describe bounds of data that have been received; data outside these bounds are to be considered lost. Whether or not the receiver is permitted (and the sender is able) to send SACKs is denoted by the SACK-permitted option from the handshake.

Improvements to Fast Retransmit and Fast Recovery

To improve upon the Fast retransmit and Fast Recovery algorithms, one can implement the use of SACKs to determine how many bytes are in flight by maintaining a *pipe* variable. The *pipe* variable holds an estimate of the number of segments that have not been SACKed nor considered lost in the interval from the highest acknowledged sequence number to the highest sequence number that has been transmitted. This estimation uses heuristics to determine if a packet should be considered lost. This is detailed in RFC 6675 [19].

The downside to this method is that it tends to send large bursts of data when experiencing heavy losses. The reason for this is that the *pipe* estimate can become inaccurate [20]. When the algorithm enters recovery mode, it will first adjust *cwnd* and *ssthresh* to half of the *FlightSize*. It will then retransmit the lost segment and more if *cwnd* allows it. For each received ACK, (*cwnd - pipe*) amount of data will be sent out.

Another potential weakness is the delay between retransmitting the first lost segment and subsequent segments. As *cwnd* and *ssthresh* are set to *FlightSize* / 2 (while we assume the value of *pipe* is close to *FlightSize*), subsequent data will not be sent out until half of the outstanding acknowledgments have been received. This issue is only prevalent when losing at least (*cwnd* / 2) amount of data [20].

Decreasing DupACK threshold

A weakness of standard Fast Retransmit is potentially poor loss detection at the end of a flow. If a loss occurs near the end of a transmission, there might not be enough outstanding segments to generate the three DupACKs required to trigger recovery. The Early Retransmit algorithm proposes to fix this by lowering the DupACK threshold in certain situations. The following conditions must hold to use Early Retransmit:

- The number of outstanding unacknowledged segments must be less than 4.
- The client has no new data to transmit *or* the *rwnd* forbids sending new data.

If the following conditions hold, the client can set the DupACK threshold to $outstanding\_segments - 1$ where *outstanding_segments* are segments sent, but not yet acknowledged.

Adjusting the Slow Start threshold

There are various approaches for how to adjust `ssthresh` during recovery. In Equation 2.3 it is denoted that `ssthresh` should be adjusted to *FlightSize / 2*. The following statements are based on the discussion in [21].

CUBIC is a congestion control algorithm that utilizes a cubic congestion window increase function [22], rather than a linear increase function (this is detailed in section 2.2.4). In CUBIC, which is the default in many Linux distributions today, *ssthresh* is to be set to $cwnd \times beta\_cubic$ where *beta_cubic* is usually set to 0.7 [22].

Adjusting by `cwnd` in this manner can be suboptimal when the sending rate is application-limited. If the sending rate is application-limited, it means that TCP is not able to utilize the available `cwnd` capacity due to the application not writing sufficient amounts of data to the socket. Additionally, TCP may increase the `cwnd` when ACKs arrive on time. As a result of these two factors, `cwnd` may be arbitrarily high. Upon a congestion event, the adjusted `cwnd` value may be higher than `FlightSize`, effectively not reacting to the congestion at all.

On the other hand, `FlightSize` can be inaccurate when packets are dropped at the end of a burst in application-limited transmissions. If the client sends data in a burst, the `FlightSize` will initially increase, before converging to a significantly lower value once acknowledgments for those packets are received. If a congestion event occurs at the end of such a burst, the `cwnd` will be adjusted to a lower value. This value may be disproportionately low as `FlightSize` in this case is not representative of the link capacity, but rather limited by the application.

Window Scaling

As networks evolve, the capabilities of TCP must follow. As illustrated in Figure 2.1, the `window` field, which informs the client about the `rwnd` is only 16 bits long. This allows for a window size with an upper bound of 2^{16} bits, or 64 KiB. Initially, this was sufficient, however, the increased available bandwidth in networks has prompted a need for larger `rwnd` sizes.

To solve this issue, the TCP option *Window Scale* was implemented. This option is communicated in the handshake process and requires both the client and server to set scaling options in the exchanged SYN segments. This option includes a *shift count* that describes how many times to shift the advertised `rwnd` to derive the scaled window size. The shift count can be set to a maximum of 14, resulting in window sizes of up to $(2^{16} - 1)^4 bits * 2^{14} bits = 2^{30} - 1 bits \approx 1 GiB$ [23]. The reason for this limitation is related to the way TCP validates sequence numbers [5].

Proportional rate reduction

Proportional Rate Reduction is an alternative to Fast Recovery to adjust the `cwnd` to a more conservative value during recovery, resulting in a `cwnd` size close to `ssthresh` after recovery. It achieves this by adjusting the `cwnd` proportionally based on network conditions. The following rules are implemented upon entering recovery [24]:

1. If $FlightSize > sstresh$, `cwnd` reductions are spread out across an RTT.
2. Else, reduce `cwnd` based on the following conditions:
    - If losses keep occurring, follow the principle of packet conservation [25] to send as much as delivered.
    - Else, increase `cwnd` by how much data was delivered to quickly reach `sstresh`, much like in regular Slow Start.

PRR is widely deployed as the standard in Linux today [24].

Preventing timeouts

Recent Acknowledgement Tail Loss Probe (RACK-TLP) is a mechanism that attempts to more often induce Fast Recovery [26], instead of waiting for an RTO to occur. This can speed up the time spent in recovery, as losses

[4] The advertised window value must be subtracted by one to prevent wraparound, as it is treated as an unsigned number.

[5] To not discard data as old, TCP must ensure that the client window differs no more than 2^{31} bytes away from the right edge of the server-side window. Since either side of the server and client windows can differ by the window size, it implies that the scaled window must be less than $2^{30} bits$.[23]

are repaired quicker, while `cwnd` reductions are minimized. RTOs may be triggered by the following scenarios:

1. Packet drops for short flows or at the end of an application data flight.
2. Lost retransmissions.
3. Packet reordering.

To improve upon loss detection, RACK utilizes time, rather than counting DupACKs, to trigger Fast Recovery. It does this by keeping timestamps for each segment sent. Additionally, each segment is given an expiration time based on RTT measurements and a *reordering window*, which is an additional delay to account for segment re-ordering [26]. Upon receiving an acknowledgment, RACK will adjust all recorded timestamps based on the measured RTT. If a timestamp expires, the associated segment is marked as lost. It is worth noting that, even though RACK does not rely on DupACKs to arrive, it is still dependent on acknowledgments arriving. Intuitively, this could prove problematic in flows with tail losses.

Tail Loss Probe (TLP) is a mechanism that can be implemented to improve upon RACK, hence the name RACK-TLP. At the end of a flow, TLP will either send a new segment or retransmit the outstanding segment with the highest sequence number, in an attempt to provoke the receipt of an acknowledgment. This lets RACK continue checking for the expired segment to trigger Fast Recovery, rather than waiting for an RTO.

This approach may contribute to quicker recovery phases and less `cwnd` reductions during heavy losses when the client is application-limited or when there is limited in-flight data [26]. The main drawback of RACK-TLP is associated with the requirement of additional state management compared to DupACK counting. Essentially, a RACK implementation requires the sender to keep track of per-packet transmission timestamps [26].

2.2.4 Congestion Control Algorithms

There are several congestion control algorithms in use today, each with its approach to managing network traffic. In this section, we will provide an overview of some of the most widely used congestion control algorithms implemented on top of TCP.

Reno

Reno is the default fallback congestion control algorithm used in Linux today and it is based on Van Jacobson's Slow Start and Congestion Avoidance algorithms [25]. Reno is a loss-based congestion control algorithm, meaning it relies on packet loss to detect congestion in the

network. It employs an exponential increase in Slow Start where two packets are transmitted for each arriving ACK. The Congestion Avoidance phase features an additive increase, where the `cwnd` is expanded by (1/`cwnd`) per arriving ACK.

BBR

Bottleneck Bandwidth and Round-trip propagation time (BBR) is a congestion control algorithm that reacts to actual link congestion rather than packet loss. The idea of BBR is to have the sending rate converge to an optimal value (the Kleinrock point), which is the intersection between minimal congestion and the highest bandwidth utilization [27].

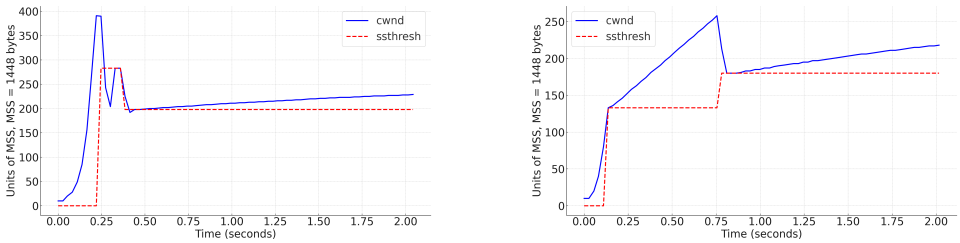
Gauging bottleneck bandwidth To reach this optimal sending rate, BBR continually probes the network for more bandwidth. Two key metrics need to be continually re-measured to achieve this; the RTT and the bottleneck bandwidth. However, only one of these metrics can be measured at any given time. This is because the RTT cannot be re-measured without lowering the sending rate *below* the estimated BDP, while the bottleneck bandwidth needs to be measured with a sending rate *above* the estimated BDP. These two metrics must be continually probed and measured as a transport path can be subject to change (physical path, link capacity, traffic, etc.).

BBR uses these measurements to estimate both the volume and rate at which to send data. Ideally, data should be sent at the rate at which packets leave the network (the rate of the bottleneck link), while the volume of in-flight data should match the BDP [28]. A mismatch between the rate and the volume can cause an increase in delay and losses. If the rate equals the bottleneck rate, but the in-flight data exceeds the BDP, queues will form. Also, if the client sends a BDP worth of packets in a burst, the packets will either be buffered or dropped at the bottleneck.

CUBIC

CUBIC is another congestion control algorithm that modifies the standard linear window growth. The CUBIC increase function is instead cubic, hence the name. It is suitable for networks where the BDP is large, as the increase function allows for a more aggressive increase when the congestion window is far from saturated, while gradually slowing down as it reaches saturation [29]. This should result in a stable and scalable algorithm, and it is the default congestion control algorithm in Linux today.

Increase function The stability of the CUBIC algorithm is achieved by the use of a binary search function to determine how fast the window should grow. The trick is to find the mid-point between the window size at the



(a) CUBIC without Hystart.

(b) CUBIC with Hystart.

Figure 2.2: CUBIC with and without Hystart

last loss event and the maximum window size where no loss occurred throughout an RTT [29]. If these two points are far apart, eg. the current window size is significantly smaller than the window size at the last loss event, the window will aggressively increase to the mid-point. The new range is then set to the mid-point and the maximum window size, making the next window increase smaller. When the previous saturation point is exceeded, the window will continue to increase gradually to probe for more bandwidth.

Hystart To mitigate an overshoot during Slow Start, Linux CUBIC employs a mechanism called *Hystart*. This algorithm is active after a certain window threshold during Slow Start, which attempts to detect congestion before a loss occurs. Hystart monitors the frequency of arriving ACKs, and increases in RTT, to determine if congestion occurs in the network [30]. When congestion is detected, Hystart transitions TCP to Congestion Avoidance, effectively ending Slow Start earlier than traditional TCP.

2.3 Pacing

When sending data over a network, both the volume and sending rate can influence congestion. In TCP, the volume of in-flight data is governed by the *cwnd*. However, this window does not control the rate at which data is sent, potentially allowing for burst traffic. This section aims to explore scenarios where TCP can exhibit bursty behavior and why this can lead to a detriment in performance. To explore whether or not pacing is worth pursuing, we will feature two papers on the topic of pacing performance.

Motivation

Bursty behavior in TCP could occur in the following scenarios:

- Regular Slow Start is bursty by nature, as it sends two new packets for each arriving acknowledgment received. If acknowledgments are generated at the bottleneck rate, TCP is effectively sending data at a

rate double what the bottleneck can handle. This leads to a queue forming at the bottleneck [31], and inevitably packet drops.

- If a lost packet is retransmitted successfully, the client may free up the *cwnd*, allowing a burst of traffic [31]. As acknowledgments are cumulative, the arriving acknowledgment may contain sequence numbers past the packet that was lost, resulting in an expanded *cwnd*.
- *Ack Compression* is a phenomenon where acknowledgments are queued behind data segments on the reverse path [31]. This could eliminate the spacing between acknowledgments that initially left the receiver at the bottleneck rate, thereby resulting in a burst of acknowledgments at the client, freeing up the *cwnd*.
- When a bottleneck link is shared by multiple bursty flows, there is a potential for generating a larger burst than each of the flows would do individually. If the bursty traffic of two flows arrives simultaneously at the bottleneck, the result is an even bigger burst. This will either be absorbed by the bottleneck buffer causing queue buildup, or packet drops.

Intuitively, these issues could be mitigated by *pacing* packets, rather than transmitting them in a burst. Pacing involves distributing packets across a designated time unit, such as a window's worth of packets spread across an RTT. This gradual distribution of packets facilitates a smoother increase toward bottleneck capacity. In other words, the *cwnd* determines the volume of data, and pacing determines the sending rate. From the perspective of queuing theory, burst traffic can in the worst case increase delay linearly with network load. Pacing, on the other hand, can achieve a constant delay until the bottleneck is fully subscribed [31].

Research

The research on whether or not pacing is beneficial has been mixed. The simulation study on pacing in [31] from 2002 showed that pacing traffic results in higher latencies and lower throughput in many situations. This paper simulated paced and non-paced flows in several different scenarios[6]. They used TCP Reno as the non-paced version and a paced modification of Reno, *Paced Reno*, that distributed a window of transmissions across an RTT. They concluded with the following key findings:

- For a single flow, Paced Reno performed better than Non-paced on a bottleneck with buffer capacity equal to 1/4 of the BDP. This can be attributed to the paced flow being able to Slow Start for longer, as the

[6]The setup consisted of a set of clients and servers, connected via a bottleneck router using a FIFO queue. The links from the end hosts to the bottleneck were configured at four times the bandwidth of the bottleneck at a delay of 5ms. The bottleneck link itself was configured with a 40ms delay.

non-paced flow quickly saturates the bottleneck causing congestion. As buffer capacities increase, the non-paced flow performs better as the bursts can be buffered.

- When multiple paced flows compete, there is a possibility for *synchronized drops*. As paced traffic is distributed across time, packets from multiple flows become mixed. As a consequence of this, multiple flows experience congestion when the bottleneck is finally oversubscribed, thereby causing underutilizing of the link, as the flows collectively decrease their sending rate. This results in poor throughput compared to non-paced flows, where some of the flows experience congestion before the bottleneck is fully utilized.

- Pacing can achieve improved normalized fairness due to synchronization. The synchronization of flows results in simultaneous back-offs, which effectively prevents any one flow from using an unproportionate amount of the bandwidth.

- When paced and non-paced flows compete, paced flows are more likely to experience congestion. This could be attributed to the interleaving of paced traffic with bursty traffic.

A later paper from 2006 [1] revisits the performance of pacing and re-evaluates the results in [31]. While this paper made similar observations when conducting the same tests, it offers a more nuanced and updated assessment of the performance of pacing. Contrary to [31], this paper concluded that flow performance is better when all flows are paced, compared to when no flows are paced [1]. It also found that the performance of all flows can increase in networks shared between non-paced and paced flows when the amount of paced flows exceeds a certain threshold.

This paper introduced a new performance metric, *worst-flow latency*, which denotes the latency of the slowest flow to finish a transfer within a set of flows with equal RTTs that are initiated simultaneously to transmit an equal amount of bytes [1]. This metric is especially important to applications that depend on the completion of multiple concurrent flows. Paced flows were shown to outperform non-paced flows in many situations when comparing this metric. Through a series of experiments involving a homogeneous set of flows tested over local area networks (LANs), high-speed TCP variants in wide area networks (WANs), and varying buffer sizes, it was demonstrated that in many scenarios, the pacing of flows resulted in better performance compared to non-paced flows when evaluated against this metric.

When evaluating aggregate throughput in an isolated environment, paced flows generally perform better. These experiments were done with TCP Reno and high-speed TCP variants like BIC-TCP and FAST over networks with varying buffer sizes. Experiments conducted with Reno show that the non-paced flows slightly outperform the paced flows (with buffer sizes

exceeding 1/10 BDP packets) within an upper bound of 25%. For the high-speed variants, paced flows perform better across buffer sizes.

To improve the performance of paced flows when competing with non-paced flows, they introduced a more aggressive pacing scheme. Rather than pacing the current window across an RTT ($RTT/cwnd$), the rate was instead adjusted to correspond to the window for the *next* RTT. This effectively made the window of paced flows grow at the same rate as that of the non-paced flows. This achieved similar aggregate throughput for the paced and non-paced Reno flows when an equal amount of paced and non-paced flows share a network. However, non-paced flows still outperform paced flows when introducing the use of SACKs. This could be attributed to the more efficient detection of burst losses enabled by SACKs. When BIC-TCP flows competed, the non-paced flows were shown to have slightly higher aggregate throughput, while pacing performed better in terms of the *worst-flow throughput*.

Conclusion

The findings in [1] concluded by encouraging the use of pacing. Paced flows performed well in networks with small buffers, particularly in terms of the worst-performing flow. High-speed TCP variants were also shown to perform better when paced. Additionally, paced flows could improve the performance of non-paced flows in a mixed environment.

On the other hand, paced flows achieve lower aggregate throughput when competing with non-paced Reno/BIC-TCP flows. The implications of synchronization in paced flows, highlighted in [31], should also be considered. The emergence of new TCP variants could also affect the performance of pacing, as we have seen with high-speed variants like BIC-TCP. Ultimately, the decision on whether or not to pace should be made based on the specific requirements of the application and network in question.

2.4 TCP in the Linux kernel

The TCP stack in the Linux kernel is a sophisticated state machine that handles data transmissions across data centers worldwide. Comprehensive knowledge of the existing implementation is necessary when we are to implement a re-designed timer-based TCP in the kernel. In this section, we will give an introduction to how TCP is implemented in the Linux kernel. We will take a look at key features and mechanisms that are relevant to our implementation.

State machine

TCP transitions through a set of states through the lifetime of a flow. State transitions occur based on different events and conditions. We will now detail the relevant state machines below.

Connection state

The connection state machine tracks the lifecycle of a flow, from the handshake to the teardown process[7]. The relevant states are highlighted below:

`SYN_SENT` The `SYN_SENT` state is entered when the client initiates an outgoing connection.

`ESTABLISHED` The client enters the `ESTABLISHED` state when receiving an ACK in response to a SYN. The client can now initiate the transmission of data.

Congestion control state

The congestion control state machine tracks the state related to congestion control for an established TCP connection. The different states are as follows[8]:

`OPEN` The `OPEN` state denotes the normal, business-as-usual state, meaning no congestion events have occurred. It also allows fast forwarding of the processing of incoming packets.

`DISORDER` The `DISORDER` state is entered when SACKs or DupACKs arrive which may signify congestion. It is similar to `OPEN` but enforces more thorough processing of incoming packets.

`CWR` This state is entered due to a Congestion Notification event, such as Explicit Congestion Notifications (ECNs) [34] or congestion on the local device. Device congestion may be detected by TCP based on the response from the function call to transmit packets [35].

[7] `linux-kernel-5.19 | tcp_states.h` [32].
[8] `linux-kernel-5.19 | CA states` [33].

`RECOVERY` This state is entered when the client enters recovery, mainly due to DupACKs. In this state, the *cwnd* is reduced and `ssthresh` is adjusted, and packets may be retransmitted. The state is transitioned back to `OPEN` upon receiving a full ACK.

`LOSS` TCP enters `LOSS` due to RTO timeouts or an incoming SACK which indicates that the server retracts previously acknowledged data.

TCP modules

TCP modules are the desired way to implement new TCP variations in the kernel. TCP modules are plug-and-play implementations that consist of a single C file along with its header files. They reside in `net/ipv4/tcp_*.c` and can either be compiled as kernel modules or as part of the kernel.

A TCP module can be thought of as a layer on top of the existing TCP stack in Linux. The Linux TCP stack handles most aspects of TCP like transmits, retransmits, receiving packets, MTU-probing, etc. To allow for the creation of modules, the TCP stack exposes an API that allows for fine-grained control of TCP parameters through a set of predefined callbacks. A module is required to implement a subset of these callbacks. An example of a TCP module in its most basic form is the Reno TCP module in `net/ipv4/tcp_cong.c`, displayed in Listing 2.1.

```
1   /*
2    * TCP Reno congestion control
3    * This is special case used for fallback as well.
4    */
5   /* This is Jacobson's slow start and congestion avoidance.
6    * SIGCOMM '88, p. 328.
7    */
8   void tcp_reno_cong_avoid(struct sock *sk, u32 ack, u32
    acked)
9   {
10      struct tcp_sock *tp = tcp_sk(sk);
11
12      if (!tcp_is_cwnd_limited(sk))
13          return;
14
15      /* In "safe" area, increase. */
16      if (tcp_in_slow_start(tp)) {
17          acked = tcp_slow_start(tp, acked);
18          if (!acked)
19              return;
20      }
21      /* In dangerous area, increase slowly. */
22      tcp_cong_avoid_ai(tp, tp->snd_cwnd, acked);
23  }
24  EXPORT_SYMBOL_GPL(tcp_reno_cong_avoid);
25
```

```
26    /* Slow start threshold is half the congestion window (min
      2) */
27    u32 tcp_reno_ssthresh(struct sock *sk)
28    {
29        const struct tcp_sock *tp = tcp_sk(sk);
30
31        return max(tp->snd_cwnd >> 1U, 2U);
32    }
33    EXPORT_SYMBOL_GPL(tcp_reno_ssthresh);
34
35    u32 tcp_reno_undo_cwnd(struct sock *sk)
36    {
37        const struct tcp_sock *tp = tcp_sk(sk);
38
39        return max(tp->snd_cwnd, tp->prior_cwnd);
40    }
41    EXPORT_SYMBOL_GPL(tcp_reno_undo_cwnd);
42
43    struct tcp_congestion_ops tcp_reno = {
44        .flags       = TCP_CONG_NON_RESTRICTED,
45        .name        = "reno",
46        .owner       = THIS_MODULE,
47        .ssthresh    = tcp_reno_ssthresh,
48        .cong_avoid  = tcp_reno_cong_avoid,
49        .undo_cwnd   = tcp_reno_undo_cwnd,
50    };
```

Listing 2.1: TCP Reno module from `net/ipv4/tcp_cong.c`

Callbacks

TCP modules can access the Linux TCP stack through a set of callbacks[9]. These callbacks fire at different stages during the TCP state handling. Essentially, these callbacks allow TCP modules to control mechanisms like the congestion window, slow-start threshold, segment offloading sizes and pacing rate. The subsequent paragraphs will feature the most relevant callbacks.

`init` / `release` are both required callbacks that fire upon entering the initialization and clean-up phases respectively. The TCP module has a dedicated memory area of 104 bytes[10] within the socket that can be utilized freely. These functions are used to populate and clean up this area in most TCP implementations.

[9] `linux-kernel-5.19 | tcp.h:struct tcp_congestion_ops` [36].
[10] `linux-kernel-5.19 | inet_connection_sock` [37].

`ssthresh` is a required callback and should return the slow-start threshold to be used. It fires upon entering the `LOSS`[11] and `CWR`[12] states. Standard TCP returns half the congestion window.

`set_state` is an optional callback which fires upon a state transition[13]. The new state is provided as an argument.

`cong_avoid` is a required callback that fires towards the end of processing an incoming ACK[14]. It is replaced by the optional callback `cong_control` which is an attempt to combine multiple callbacks into one. The responsibility of these functions is to update `cwnd` and pacing rate. `cong_control` also receives a rate sample that contains delivery rates, losses and round-trip times over an interval of time.

`undo_cwnd` is also a required callback that should return the new value of `cwnd` upon transitioning from `LOSS`[15] or `RECOVERY`[16]. Standard TCP sets this to the value of `cwnd` from before the congestion event occurred[17].

`pkts_acked` is an optional callback that fires when ACKs are received. It is passed an `ack_sample` as an argument which contains the amount of packets acknowledged, RTT sampled from the ACK and the number of in-flight packets.

In addition to controlling the TCP stack, most TCP modules also maintain their own state machine through these callbacks. The state is saved in the mentioned dedicated memory area stored in the socket. An example of this is BBR (`tcp_bbr.c`), which maintains state, measurements and bandwidth estimates in its socket.

Pacing

The Linux kernel supports TCP pacing internally or through a scheduler. In both cases, the rate of pacing is controlled through a pacing rate stored in the socket associated with a flow. This rate is either updated by the TCP stack or at will by TCP modules. If the pacing rate is controlled by the former, it will by default call `tcp_update_pacing_rate` upon receipt of an ACK, which scales the pacing rate by a factor of 2 or 1.2 in Slow Start and Congestion Avoidance respectively[18].

[11] `linux-kernel-5.19 | tcp_enter_loss` [38].
[12] `linux-kernel-5.19 | tcp_cwnd_reduction` [24].
[13] `linux-kernel-5.19 | tcp_set_ca_state` [39].
[14] `linux-kernel-5.19 | tcp_cong_control` [40].
[15] `linux-kernel-5.19 | tcp_try_undo_loss` [41].
[16] `linux-kernel-5.19 | tcp_try_undo_recovery` [42].
[17] `linux-kernel-5.19 | tcp_reno_undo_cwnd` [43].
[18] `linux-kernel-5.19 | tcp_input.c:update_pacing_rate` [44].

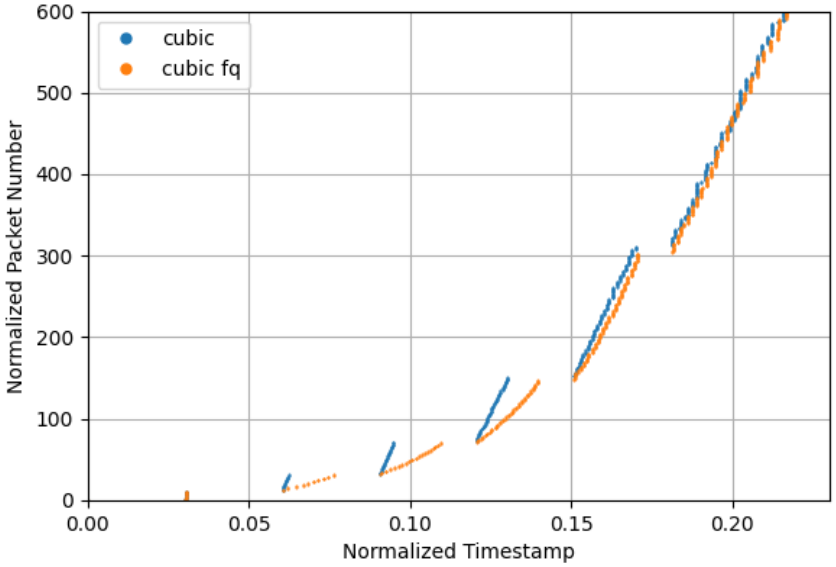


Figure 2.3: A time sequence plot showcasing the difference in slow-start between a non-paced and paced CUBIC flow using standard FQ in Linux. 50Mbit bandwidth and 30 ms RTT.

Pacing in the scheduler One way the kernel enables pacing is through the `sch_fq` (fq/pacing) packet scheduler. `sch_fq`[19] is a Fair Queue packet scheduler with optional pacing capabilities that can be deployed on the egress path of a client. It stores TCP flows in Red-Black trees and serves them in a Round Robin fashion. When serving a flow, the scheduler adds a delay between packets to conform to the rate (`sk_pacing_rate`) set in the socket belonging to the flow. The `sk_pacing_rate` in the socket denotes the maximum amount of bytes that can be transmitted every second[20]. Figure 2.3 compares a non-paced and paced cubic using the standard FQ pacing in Linux.

Internal pacing The alternative to pacing in the scheduler was introduced to accommodate the BBR implementation in Linux. The network stack can now opt-in to use internal pacing. More specifically, pacing can be done from within the TCP stack rather than in the scheduler. The argument and commit for this feature were made by Eric Dumazet [3]. He argued that Linux hosts handling a minor amount of flows did not require the high performance that the scheduler can provide. Additionally, moving pacing mechanisms to the TCP layer eliminates the reliance on a specific scheduler to be deployed on the system.

Internal pacing can be activated by changing a flag (`sk_pacing_status`) in the socket to `SK_PACING_NEEDED`. This tells the network stack to pace

[19] `linux-kernel-5.19 | sch_fq.c` [45].
[20] `linux-kernel-5.19 | sk_pacing_rate` [46].

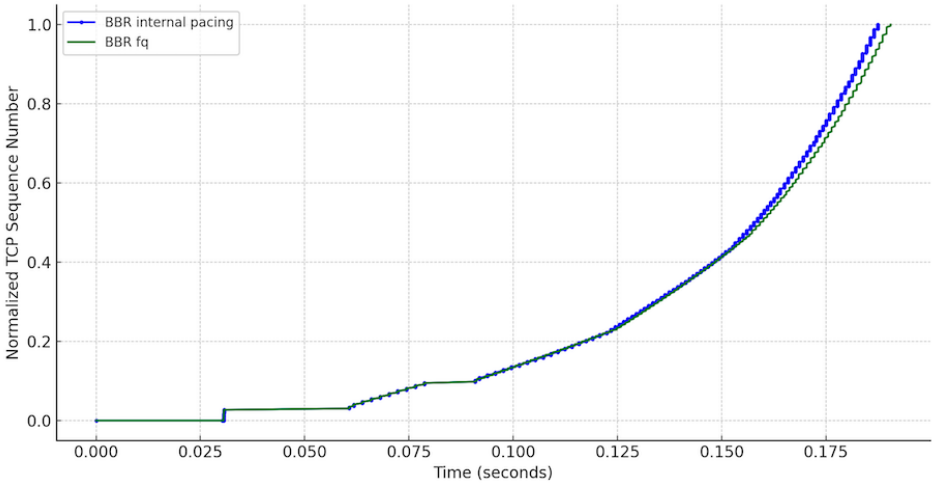


Figure 2.4: Comparison of BBR internal pacing and FQ/pacing.

packets internally. In the Linux BBR implementation, internal pacing and pacing via the scheduler are mutually exclusive. BBR will either rely on the scheduler for pacing if it is deployed, and if not, BBR will fall back to pacing internally[21]. The function responsible for handling internal pacing is `tcp_pacing_check` displayed in Listing 2.2.

```
1   static bool tcp_pacing_check(struct sock *sk)
2   {
3       struct tcp_sock *tp = tcp_sk(sk);
4
5       /* check SK_PACING_NEEDED flag */
6       if (!tcp_needs_internal_pacing(sk))
7           return false;
8
9       /* check if next packet should be sent immediately */
10      if (tp->tcp_wstamp_ns <= tp->tcp_clock_cache)
11          return false;
12
13      /* queue a timer to defer packets */
14      if (!hrtimer_is_queued(&tp->pacing_timer)) {
15          hrtimer_start(&tp->pacing_timer,
16                        ns_to_ktime(tp->tcp_wstamp_ns),
17                        HRTIMER_MODE_ABS_PINNED_SOFT);
18          sock_hold(sk);
19      }
20      return true;
21  }
```

Listing 2.2: `tcp_pacing_check` from `tcp_output.c` [48]

When TCP handles data chunks, it does this in the form of an abstraction called SKBs. An SKB (`sk_buff`) is a data structure representing a network

[21] `linux-kernel-5.19 | bbr_init` [47].

packet to be sent. It contains metadata for the packet in addition to pointers to the packet data itself. What this provides is a unified interface for a data packet that can be passed throughout the network stack.

`tcp_pacing_check` runs whenever the TCP stack attempts to transmit data. The `tcp_wstamp_ns` is a timestamp calculated based on the pacing rate and size of the SKB[22]. If this timestamp is in the past, the packet should be sent immediately and does not need to be paced. However, if the timestamp is in the future, the transmission is deferred by queuing a high-resolution timer (`hrtimer`), referred to in the code as `pacing_timer`. The intention here is to defer the transmission until the rate quota is sufficient to transmit the SKB.

When the queued `pacing_timer` fires, it triggers a callback function named `tcp_pace_kick`[23], which will attempt to transmit packets. In addition to adhering to a pacing rate, this deferred transmission also helps limit the number of packets queued at the packet scheduler, resulting in smaller queues in the qdisc. This technique is known as TCP Small Queues (TSQ), which can potentially lead to lower round-trip times [51]. If pacing is not activated, a TSQ check (`tcp_small_queue_check`) will run regardless when data is to be sent[24].

Performance A key difference between internal pacing and pacing in the scheduler is related to the performance demands. In general, pacing in the scheduler scales better than internal pacing as the number of flows grows large. This is because a single instance of FQ can potentially serve thousands of flows simultaneously, while internal pacing necessitates a separate instance for every single flow. This can be attributed to the difference in timer requirements. Internal pacing requires an `hrtimer` for every flow, while FQ uses a single `hrtimer` to serve all flows associated with the qdisc[25]. This can in turn lead to a higher frequency of interrupts when using internal pacing [3], which is known to be CPU intensive. The reason flows cannot share an `hrtimer` internally, is that a TCP flow lacks the knowledge of other flows on the system. Implementing such a shared timer between flows requires extensive synchronization, which seems unfeasible [54].

In Figure 2.4 we see a comparison of the two pacing methods in use with BBR. Visually, the pacing patterns appear similar. The initial window appears to not be paced by either method. It is also important to note that, while the flows are paced, it is not on a per-packet level, as the pacing is done in terms of distributing small bursts of packets across time.

[22] `linux-kernel-5.19 | tcp_update_skb_after_send` [49].
[23] `linux-kernel-5.19 | tcp_pace_kick` [50].
[24] `linux-kernel-5.19 | tcp_small_queue_check` [52].
[25] `linux-kernel-5.19 | sch_fq.c` [53].

High-resolution timers

High-resolution timers in the Linux kernel are implemented using the `hrtimer` API. This is a completely separate API from the `timer` API, which is used for timers with lower precision. The `hrtimer` API is used for timers with a precision of nanoseconds and is implemented using a Red-Black tree [55].

The `hrtimer` is used where high precision is required such as in the `pacing_timer`. The original `timer` API is best suited for timeouts where it is more important to know that the timer has expired, rather than the exact time it expired [55].

Modes The `hrtimer` can be configured into different modes depending on the needs. The different modes are [56]:

- `HRTIMER_MODE_ABS`: The time value is absolute, that is, it specifies a specific point in the future.
- `HRTIMER_MODE_REL`: The time value is relative to the current time.
- `HRTIMER_MODE_PINNED`: The timer will be bound to a specific CPU core so that it will not be migrated to another core by the scheduler.
- `HRTIMER_MODE_SOFT`: The timer function will be executed in soft-irq context, that is, it can be interrupted or delayed by other processes.
- `HRTIMER_MODE_HARD`: The timer function will be executed in hard-irq context, that is, it cannot be interrupted or delayed by other processes, even on `PREEMPT_RT`.

These modes can be combined to create a more specific timer. For example, the `pacing_timer` is configured to use the `HRTIMER_MODE_ABS_PINNED_SOFT` mode, which means that the timer is absolute, pinned to a CPU core, and executed in soft-irq context.

The `enum hrtimer_mode` is what defines the different modes. The code in Listing 2.3 shows the different modes and their values. The code listing is taken from the Linux kernel source code[26].

```
1  /*
2   * Mode arguments of xxx_hrtimer functions:
3   *
4   * HRTIMER_MODE_ABS       - Time value is absolute
5   * HRTIMER_MODE_REL       - Time value is relative to now
6   * HRTIMER_MODE_PINNED        - Timer is bound to CPU (is
    only considered
7   *                  when starting the timer)
```

[26] `linux-kernel-5.19 | enum hrtimer_mode` [56].

```
8     * HRTIMER_MODE_SOFT     - Timer callback function will be
      executed in
9     *                 soft irq context
10    * HRTIMER_MODE_HARD     - Timer callback function will be
      executed in
11    *                 hard irq context even on PREEMPT_RT.
12    */
13    enum hrtimer_mode {
14        HRTIMER_MODE_ABS    = 0x00,
15        HRTIMER_MODE_REL    = 0x01,
16        HRTIMER_MODE_PINNED = 0x02,
17        HRTIMER_MODE_SOFT   = 0x04,
18        HRTIMER_MODE_HARD   = 0x08,
19
20        (.........) /* Modes hidden in excerpt */
21
22        HRTIMER_MODE_ABS_PINNED_SOFT = HRTIMER_MODE_ABS_PINNED
      | HRTIMER_MODE_SOFT,
23        HRTIMER_MODE_REL_PINNED_SOFT = HRTIMER_MODE_REL_PINNED
      | HRTIMER_MODE_SOFT,
24
25        (.........) /* Modes hidden in excerpt */
26    };
```

Listing 2.3: `enum hrtimer_mode`

TSO autosizing

When sending data, the network stack may segment the packet into several smaller packets. This is called segmentation offloading (TSO), where the network stack segments the packet into MTU-sized packets [57]. TSO needs to balance making packets larger for CPU efficiency, or smaller packets to minimize burstiness. This is done in the function `tcp_tso_autosize`, where the general rule is to use larger packet sizes for flows with lower round-trip times (losses can be repaired quicker), and smaller packets for long-distance flows to maintain ACK clocking[27].

Transmission of new data

A vital part of TCP is deciding when to send data. In this section, we will highlight the two main triggers for sending new data. The first trigger is when the application writes data to a TCP socket. This results in a call to the TCP API, more specifically the `tcp_sendmsg_locked` function[28]. This function will create an SKB for the packet and ultimately transfer execution to the TCP output engine to transmit the packet.

The other trigger for packet sending is through a mechanism called ACK-clocking. This is core functionality in the TCP stack, where the `cwnd`

[27] `linux-kernel-5.19 | tcp_tso_autosize` [58].
[28] `linux-kernel-5.19 | tcp_sendmsg_locked` [59].

expands when packets leave the network. This occurs in the TCP input engine (`tcp_input.c`) in response to incoming ACKs. After processing the incoming packet, TCP will call the function `tcp_data_snd_check`[29] that in turn calls the TCP output engine to attempt a transmission.

The function responsible for handling the transmission of new packets is `tcp_write_xmit` from `tcp_output.c`. This function pops and transmits SKBs from the send buffer for as long as the `cwnd` allows. Before transmitting, the SKB is subjected to a TSQ check, segmentation and fragmentation[30]. Finally, the SKB is passed on to `tcp_transmit_skb` for header construction before handing it off to the IP-layer[31].

[29] `linux-kernel-5.19 | tcp_data_send_check` [60].
[30] `linux-kernel-5.19 | tcp_write_xmit` [61].
[31] `linux-kernel-5.19 | __tcp_transmit_skb` [62].

Chapter 3

Timer-based TCP

In this chapter, we present a comprehensive specification of the algorithm that will be implemented in the Linux kernel. Timer-based TCP (TBTCP) is a complete re-design of TCP that leverages a single timer to progress its state machine instead of relying solely on the "ACK-clock". This makes TBTCP a fully paced TCP variant, where packet transmission is initiated as a consequence of timer expiration.

The scope of this thesis excludes the implementation of recovery mechanisms. Consequently, we exclude the explanation of these mechanisms from this specification. As such, this specification can be thought of as a simplified version of TBTCP. Even though our specification of TBTCP is simplified, we still refer to it as TBTCP in this chapter.

Importantly, TBTCP impacts only the sender, while remaining fully compatible with a standard TCP-receiver. Therefore, it can be deployed in networks comprising both TBTCP and regular TCP clients and servers. Congestion control interactions are left to be studied, however. The contents of this chapter are drawn exclusively from the TBTCP research paper and personal correspondence with its authors. At the time of writing, the research paper is unpublished.

Simplifying assumptions

Our algorithm specification makes a few assumptions about TCP usage for the case of simplicity:

- The TCP Timestamps option RFC 7323 [23] is enabled.
- A sender always has something to send—there are no periods of quiescence.
- All our sequence numbers count packets, not bytes.
- As this specification excludes recovery mechanisms, we assume regular TCP recovery is implemented. This entails that execution

is handed off to TCP to recover losses. Once losses are recovered, execution returns to TBTCP.

3.1 General design

TBTCP leverages a single timer to progress its state machine. When a timer expires, it is re-queued with an updated timeout value. The timeout value is calculated based on the current state of the connection with respect to the current sending rate.

Unlike traditional congestion control algorithms that employ a congestion window, TBTCP regulates its transmission rate by means of a novel sequence number, `Ntx`. This sequence number is incremented for every new packet transmitted, and adjusted upon a congestion event. The algorithm's primary entry point is the timeout-handler, which is triggered when the timer expires. Upon a timeout event, the handler updates the global state, and if allowed by the current state, transmits a new packet.

In addition, the ACK-handler serves as a secondary entry point for the algorithm and is invoked upon receipt of an ACK. The ACK-handler updates the global state and recalculates relevant values for use in the timeout handler.

Variables

Throughout this specification we will refer to the following set of variables to describe the algorithm:

Name	Initial value	Description
RTT	RTT from SYN-ACK	Most recent RTT sample from an ACK.
ssthresh	-	This variable is used to determine the back-off when entering FR.

(a) Well-known TCP variables, used similar to standard TCP

Name	Initial value	Description
NlastAck	0	The highest cumulatively acked sequence number so far. This is called "HighACK" in RFC 6675 [19].

(b) Well-known TCP variables with new names or slightly different usage

Table 3.1: List of known TCP variables used in Functions 1 and 2, and algorithm 1 and their initial values. Variables beginning with "'N" are sequence numbers. An initial value of "-" means that initialization is irrelevant (it could be 0, for example), as the variable is first written by the algorithm.

Name	Initial value	Description
Ttx	now	The next scheduled event.
Ntx	IW	The sequence number used for calculating the pacing rate. Note that this is not an actual packet sequence number, as it begins at 1 and reflects a state associated with a certain sending rate. It is hence reset to 1 upon an RTO. The initial window (IW) is a parameter.
Nak	Ntx	The next expected acknowledgement's sequence number based on Ntx.
Tak	now + RTT	Predicted arrival time (at the sender) for the next expected acknowledgement.
pacingTime	-	The pacing time (time gap from one packet to the next) used for the most recent transmission.
lostPackets	empty	An array to hold the lost packets (with Ntx and Nak at the time of their first transmission, denoted by "lostPackets.front.[Ntx or Nak]") that are going to be re-transmitted.
postRecovery	False	Flag to enable transmitting new data at the end of recovery. This is set to True when TCP enters recovery.
postLoss	False	Flag to enable transmitting new data at the end of loss recovery.
state	-	This variable holds the current congestion control state.
BETA	-	Constant that denotes the backoff factor for the sending rate.

Table 3.2: List of new variables used in Functions 1 and 2, and algorithm 1 and their initial values. Variables beginning with "'N" are sequence numbers, and variables beginning with "T" are timestamps. An initial value of "-" means that initialization is irrelevant (it could be 0, for example), as the variable is first written by the algorithm. In this table and in all code elements, "now" denotes the current time.

3.2 Core functions

TBTCP consists of several core functions. The two entry points for the algorithm are the timeout-handler (Algorithm 1) and the ACK-handler (Algorithm 2).

Delta calculation

The `DT` function calculates the pacing time from one scheduled transmission to the next, in both Slow Start and Congestion Avoidance. Which calculation to use is decided by the `state` variable, which is set based on the difference between `Ntx` and `ssthresh`. The input of the function is the sequence number (Ntx) to calculate from, the `state` and the number of time steps to move ahead, `k`. How many time steps to input is the number of packets the timeline should progress.

The current algorithm uses an exponential increase function (line 3) in Slow Start and a linear increase in the Congestion Avoidance state (line 5), yielding a behavior comparable to a TCP Reno mechanism. The derivation of the equations in these lines can be found in [63]. The result of this function is a factor `f`, where $0 < \texttt{f} < 1$. This factor can be multiplied with the RTT to calculate the pacing delta.

Separating out this logic facilitates replacing equations in order to implement a different congestion control increase behavior, e.g. CUBIC. This is a cornerstone in the algorithm design.

Function 1: DT: delta time (how long to wait before transmitting the next packet) update based on sequence number n, the *state* and k, the number of time steps to move ahead.

```
1 Function DT((n, state, k)):
2     if state == SS then
3         return log2(1+k/n)
4     else
5         return (sqrt(8*(n+k)-7)/2 - sqrt(8*n-7)/2)
6     end
7 end
```

Pace

The `Pace` function will transmit a new packet, update the state, and schedule the next packet transmission. Additionally, `Pace` will add the transmitted packet to `lostPackets` to keep track of in-flight packets. To progress the sending rate, `Ntx` is incremented by one, to signify that a packet was transmitted. The factor returned from `DT` is multiplied with

the current RTT to find the delta with which to schedule the timer. `Pace` is triggered either from timer expiration or when an acknowledgment arrives later than expected.

Function 2: Pace: transmit a packet and schedule the next event.
Global variables used (read-only): *RTT, state*
Global variables used (also changed): *Ntx, lostPackets, pacingTime, Ttx*

```
1 Function Pace():
2     transmit a new packet and update lostPackets
3     pacingTime = DT (Ntx, state, 1)
4     Ntx += 1
5
      /* Schedule the next packet transmission */
6     Ttx = now + pacingTime*RTT
7 end
```

Slow-start threshold calculation

As in regular TCP, `ssthresh` is re-calculated when experiencing losses. In TBTCP, the calculation is based on the amount of in-flight packets at the time the loss occurred. The result determines at which `Ntx` the `state` transitions between slow-start and congestion-avoidance.

All packets transmitted by TBTCP are stored in the `lostPackets` list. At all times, the next packet we want to be acknowledged is found at the head of this list. By calculating the difference between `Ntx` and `Nak` associated with this packet, we find the number of in-flight packets at the time of transmission. This gives an estimate of the amount of in-flight packets at the time of loss, instead of an RTT later, when the loss is discovered. A back-off factor `BETA` is then multiplied with the packet estimate to achieve the final result.

This is especially helpful in slow-start, where the amount of in-flight packets doubles within an RTT, due to the exponential increase of the sending rate. Consider the scenario where `BETA` is set to 0.5 to halve the sending rate upon loss. If we were to use the number of in-flight packets at the time of loss, the sending rate would be halved to approximately the sending rate at the time the packet was transmitted. This has the potential to cause a double drop, as the sending rate is set to the value that was already too high.

Function 3: ssthresh: calculate the new value of `ssthresh` based on the number of in-flight packets multiplied by a back-off factor.
Global variables used (read-only): *lostPackets*

```
1 Function ssthresh():
2 |   lostPacketsFront = lostPackets.front;
3 |   return max(lostPacketsFront.Ntx - lostPacketsFront.Nak) * BETA, 2)
4 end
```

Timeout-handler

The timeout handler is called when the timer expires. There are two paths of execution based on whether or not the timer fired before we expected an ACK. If the timer fires before we expect an ACK, `Pace` will be called to transmit a packet and schedule a new timer. However, if the timer fires after we expect an ACK (`Tak`), the execution will stall until an ACK is received. This can occur as a result of an increase in RTT, causing the ACK to arrive later than expected.

Algorithm 1: Timeout handling.
Global variables used (read-only): *Ttx, Tak*

```
1 ON EVENT: now == Ttx
  /* Normal transmission:  we may send, or stall if we wait for an ACK
     */
2 if Ttx < Tak then
3 |   Pace()
4 end
```

ACK-handler

To further understand the timeout-handler, it is helpful to examine the role of the ACK-handler as the two coincide. The ACK-handler is responsible for detecting increases in the round-trip time (RTT), which typically arises from network congestion due to queue buildup.

When in either Slow Start or Congestion Avoidance mode, the ACK-handler leverages the RTT and the incoming sequence number to compute a global variable referred to as `Tak`, which denotes the expected time of arrival of the next ACK. The timeout handler utilizes this variable in its operations.

The function is executed once an ACK is received. First, it will update `lostPackets` by removing acknowledged packets, as these are no longer needed. If TBTCP execution resumes after a loss event, either `postLoss` (RTO) or `postRecovery` (Fast recovery) will be set to true. In either of these events `Ntx` and `Nak` are adjusted to decrease the sending rate. In

postLoss, the sending rate will be completely reset, while in postRecovery the sending rate is adjusted based on ssthresh. In both cases, Nak is set equal to Ntx to signify that there are no packets in flight. It is worth noting that there may be packets in flight from the recovery phase handled by TCP, however, TBTCP does not account for these for simplicity. For this reason, Tak is set to an RTT from now. In the regular case, Tak is adjusted by adding the computed delta to now.

Lastly, the function checks if Ttx <= now. This condition is true if the transmission of packets has stalled, as described in the timeout-handler. In the case of stalling, we can now safely resume transmission of packets by calling Pace.

Algorithm 2: ACK arrival.

Global variables used (read-only): *ssthresh*

Global variables used (also changed): *RTT, NlastAck, Nak, Ntx, lostPackets, postRecovery, postLoss*

Local variables used: *packetsAcked*

Inputs: *ACK_N.cum:* Highest cumulatively ACKed sequence number in the arriving ACK. RTTSample: RTT from ACK sample

```
1  ON EVENT: ACK arrival
2  Update lostPackets from DupACK and SACK information, if available
3  if ACK_N.cum <= NlastAck then
4      return
5  RTT = RTTSample
6  packetsAcked = number of previously unacknowledged packets covered by ACK
7  NlastAck = ACK_N.cum
8  if postLoss then
       /* Returning from loss recovery */
9      Ntx = Nak = 1
10     Ttx = now
11     Tak = now + RTT
12     postLoss = false
13 else if postRecovery then
       /* This will happen if we receive a full-ACK */
14     Ntx = Nak = ssthresh * (ssthresh - 1) / 2 + 1
15     Ttx = now
16     Tak = now + RTT
17     postRecovery = False
18 else
19     Tak = now + DT(Nak, state, packetsAcked)
20 Nak = Nak + packetsAcked;
21 if Ttx <= now then
22     Pace() // This was supposed to happen, had the ACK arrived at Tak
```

3.3 Differences from TCP

Even though the goal of the TBTCP algorithm is to mimic the behavior of TCP, the way TBTCP achieves this is drastically different. This section will

describe the differences between TBTCP and TCP.

Pacing

Pacing is a common concept of regular TCP implementations, but TBTCP's pacing is quite different. Regular TCP still uses a congestion window and only uses pacing to limit bursty behavior. TBTCP is an entirely paced algorithm that does not use a congestion window in Slow Start and Congestion Avoidance. Instead, TBTCP uses pacing to dictate both how much and how fast to transmit data. This is a fundamental difference between TBTCP and regular TCP.

Congestion Window

TBTCP does not use a congestion window in Slow Start and Congestion Avoidance, but rather keeps track of a sending rate using `Ntx`. This is a radical difference from regular TCP. The congestion window is used to limit the number of packets that can be sent without receiving an ACK. Instead, the sending rate of TBTCP is determined by the delta calculations, as well as an additional mechanism based on `Tak`, that stalls transmission if an ACK should have arrived. Despite this different philosophy, TBTCP will act similarly to regular TCP.

Optimistic transmissions

From the lack of a traditional congestion window in combination with how the pacing scheme is partially disconnected from the ACK-clock, TBTCP will send packets earlier than regular TCP. This phenomenon is most prevalent in the Slow Start phase where TBTCP transmits packets where regular TCP would instead wait for arriving ACKs before expanding the `cwnd` and sending a new burst.

However, this early packet transmission in TBTCP still depends on the receipt of ACKs, as transmissions will halt if ACKs stop arriving. This is done in order to not overwhelm the network.

Chapter 4

Linux implementation

We have chosen to implement TBTCP in the Linux operating system. Linux was appealing as it is a widely used and highly relevant open-source project. Furthermore, it is straightforward to compile, deploy and test on actual hardware. This is especially true for systems already using Linux, as compiled kernels can be installed and booted with ease. Additionally, its support for TCP modules was appealing as it allows developers to develop new TCP functionality on top of the underlying network stack. We have chosen to carry out our implementation in the 5.10 Linux kernel, as this kernel was supported by the hardware we used for testing.

4.1 Scope

The scope of this implementation is to implement the pacing and ACK logic of TBTCP in Slow Start and Congestion Avoidance, as detailed in chapter 3. This involves handing off control to the TCP stack whenever TCP experiences losses and initiates recovery. A few challenges arise from this. Most notably, we must be able to determine an appropriate rate at which to send when returning from recovery.

We will achieve this by implementing a pluggable TBTCP module to act as the state machine and controller. Furthermore, the underlying network stack should be modified to facilitate implementing the TBTCP algorithm in a module.

4.2 Strategies

Encapsulating logic

Our implementation strategy involves implementing as much logic as possible within a TCP module, rather than modifying the underlying kernel stack. However, we acknowledge that the functionality required by

TBTCP is outside the scope of a TCP module. Thus, we intend to modify the kernel such that it exposes the necessary functionality to TCP modules while ensuring the support of existing TCP modules in our modified kernel.

The majority of our implementation is encapsulated in a TCP module, referred to as the TBTCP module from now on. The TBTCP module resides in a single file, `tcp_tb.c` located in `net/ipv4`. Other noteworthy changes are made in `tcp.c`, `tcp_timer.c`, `tcp_input.c` and `tcp_output.c` all located in `net/ipv4`, in addition to `include/net/tcp.h`. We have chosen to omit to document changes not directly relevant to the implementation of TBTCP.

Naming conventions

We primarily follow the naming convention of the Linux kernel except for specific TBTCP variables. This is done to make our implementation more relatable to the TBTCP algorithm specification. These variables are detailed in Table 3.2.

4.3 Modifying the network stack

This section will detail the changes implemented in the underlying network stack. These are both changes required to change the behavior of TCP as well as extend the TCP module API with new functionality.

4.3.1 Event handler callback

We introduce a new callback to the TCP module API called `event_handler`. The existing TCP module API is provided in section 2.4 for reference. This function will serve as the callback associated with the *event timer*. The objective of this callback is to allow a TCP module to control a general-purpose timer with an associated callback that can be set by the module. The module is responsible for both queuing, re-queuing and if necessary, stopping the timer in the teardown phase. Additionally, the callback will allow a TCP module to trigger the transmission of new data.

Initializing the timer

TBTCP needs a single timer to maintain its state machine. This event timer will effectively handle the pacing aspect of TBTCP, which requires the timer to fire at a high frequency. To achieve this, we will make use of a single high-resolution timer, or `hrtimer` in Linux. High-resolution timers were previously described in section 2.4. This timer will be referred to as `event_timer` from now on.

The `event_timer` will reside in the socket structure that the network stack maintains for every flow. The timer is initialized in the function `tcp_init_xmit_timers`, where a callback function is specified to be invoked upon timer expiration. The timer is initialized with the mode `HRTIMER_MODE_ABS_PINNED_SOFT`, which most notably gives us a timer using absolute timestamps. The use of absolute timestamps is essential to implement TBTCP logic.

Timer callback

As previously mentioned, we aim to implement as much logic as possible in our TCP module. Therefore, the timer callback will simply call the TCP module and let it decide what to do next. The handler in Listing 4.1 is defined in `tcp_output.c` and will first find the expired timer, then proceed to invoke the event handler callback defined by the TCP module. Lastly, the function returns the enum `HRTIMER_NORESTART` which specifies not to restart the timer, as this is to be done from the TCP module. We call the event handler inside a critical region, as we do not want race conditions and unintended behavior when dealing with packet transmissions.

```
1  enum hrtimer_restart tcp_event_handler(struct hrtimer
   *timer) {
2      struct tcp_sock *tp = container_of(timer, struct
   tcp_sock, event_timer);
3      struct sock *sk = (struct sock *)tp;
4
5      const struct tcp_congestion_ops *ca_ops =
   inet_csk(sk)->icsk_ca_ops;
6
7      bh_lock_sock(sk);
8      ca_ops->event_handler(timer);
9      bh_unlock_sock(sk);
10
11     tcp_check_space(sk);
12     sock_put(sk);
13
14     return HRTIMER_NORESTART;
15 }
```

Listing 4.1: tcp_output.c: event timer callback

Differences from internal pacing

As detailed in section 2.4, Linux already employs a single `hrtimer` for each TCP flow. As explained, internal pacing paces packets to respect a set rate. It defines intervals where TCP can transmit packets, in addition to when transmissions should be deferred. In this section, we will explain how our

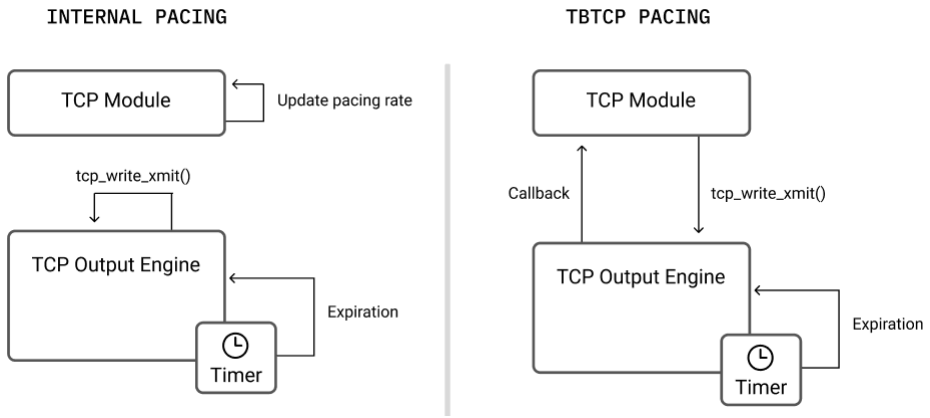


Figure 4.1: Illustration highlighting the architectural difference between internal pacing in Linux and the TBTCP pacing framework. In TBTCP, the output engine calls the TCP module to initiate transmissions. Internal pacing only allows control by the module through a pacing rate, while transmissions are initiated by the output module.

implementation of pacing is fundamentally different from Linux's internal pacing mechanism.

Time based Our implementation of pacing is based on time, rather than a rate. This fundamental difference allows for the implementation of pacing schemes using timestamps, rather than calculating the amount of bytes to send in an interval.

Fully paced The TBTCP pacing framework allows for fine-grained pacing of packets. This facilitates complete control of every single packet transmission. Internal pacing is limited to only inferring pacing times through a set pacing rate.

TCP module control The TBTCP framework can be fully controlled through a TCP module. Internal pacing in Linux supports controlling the rate of pacing through modules, however, the transmission itself is initiated by the underlying TCP stack. Our pacing framework allows for initiating packet transmission through the use of `tcp_write_xmit`[1], directly from the module.

Performance requirements We expect our pacing framework to introduce a tradeoff between control and performance. Triggering a timer for every single packet transmission is CPU-intensive, as the rate of interrupts will be high. Internal pacing mitigates this by allowing pacing in intervals, and deferring transmissions when needed through the use of the timer.

[1] `linux-kernel-5.19 | tcp_write_xmit` [61].

4.3.2 Disable sending triggers

We have previously described certain sending triggers in section 2.4. One of these is a response to incoming ACKs, which advances the state of a TCP flow. Among these state advances may be expanding the `cwnd` and trigger the transmission of new packets. However, these ACK-clocked transmissions from the TCP stack are counter-intuitive to TBTCP, as we want to have fine-grained control over when packets are sent. Thus, we must disable these sending triggers entirely when TCP is not recovering from losses.

Disable ACK triggered transmissions

The function responsible for checking if data can be sent is `data_snd_check` in `tcp_input.c`. This function must be modified to ensure that new data is not transmitted when TBTCP is operational.

```
1    static inline void tcp_data_snd_check(struct sock *sk)
2    {
3        if (tcp_is_tb(sk) && sk->sk_state == TCP_ESTABLISHED
     && (inet_csk(sk)->icsk_ca_state != TCP_CA_Recovery &&
     inet_csk(sk)->icsk_ca_state != TCP_CA_Loss))
4            return;
5
6        tcp_push_pending_frames(sk);
7        tcp_check_space(sk);
8    }
```

Listing 4.2: tcp_input.c: send new data on ACK

The modified function in Listing 4.2 will perform an early return if TBTCP is the module in use and TCP is in the `ESTABLISHED` state while not recovering from losses. Notice that we still allow for ACK-clocked transmissions when TCP is recovering from losses. This is vital for TCP recovery to function properly.

Disable application triggered transmissions

As we do not want application pushes to influence the behavior of TBTCP, we disable them entirely. This can be achieved by modifying the API exposed to the application socket in the `tcp.c` file.

```
1    if (!tcp_is_tb(sk)) {
2        if (forced_push(tp)) {
3            tcp_mark_push(tp, skb);
4            __tcp_push_pending_frames(sk, mss_now,
     TCP_NAGLE_PUSH);
```

```
5         } else if (skb == tcp_send_head(sk))
6             tcp_push_one(sk, mss_now);
7     }
```

Listing 4.3: tcp_input.c: excerpt from tcp_sendmsg_locked

```
1     if (!tcp_is_tb(sk))
2         __tcp_push_pending_frames(sk, mss_now, nonagle);
```

Listing 4.4: tcp_input.c: excerpt from tcp_push

```
1     if (!tcp_is_tb(sk)) {
2         if (forced_push(tp)) {
3             tcp_mark_push(tp, skb);
4             __tcp_push_pending_frames(sk, mss_now,
      TCP_NAGLE_PUSH);
5         } else if (skb == tcp_send_head(sk))
6             tcp_push_one(sk, mss_now);
7     }
```

Listing 4.5: tcp_input.c: excerpt from do_tcp_sendpages

Here we disable code related to pushing of data in the `tcp_sendmsg_locked`, `tcp_push` and `do_tcp_sendpages` functions if TBTCP is the active TCP module.

4.4 TBTCP module

With necessary modifications made to the underlying TCP stack, we can now implement a pluggable TBTCP module on top of it. The following section will be spent thoroughly describing the implementation of this module.

4.4.1 Private fields

To hold the state for each TBTCP flow, we define a data structure to be contained in the private field of a socket. As explained in section 2.4, the private field is limited to 104 bytes. We define the TBTCP data structure as follows:

```
1     struct tcp_tb {
```

```
2         u64 Ttx;
3         u32 Ntx;
4         u32 ssthresh_Ntx;
5         u32 NlastAck;
6         u32 Nak;
7         u64 Tak;
8         u32 HighData;
9         u64 rtt;
10        bool postRecovery;
11        enum tcp_ca_state state;
12        struct list_head lostPackets;
13        bool stalled;
14    };
```

Listing 4.6: tcp_tb.c: TBTCP data structure

The structure totals 80 bytes and contains the data necessary to maintain our TBTCP state machine. The majority of the fields can be mapped to the TBTCP specification in chapter 3. The remaining fields are explained below:

- `rtt` contains the latest RTT sample.
- `postRecovery` is an indicator that recovery has ended. The TBTCP module must know this to synchronize state when transitioning back to `OPEN` from a recovery phase.
- `state` is used to hold the current `tcp_ca_state`. These states are described in section 2.4.
- `lostPackets` is a linked list of packets sent, but not yet acknowledged. It is used to synchronize the sending rate when transitioning back to `OPEN` from a recovery phase.
- `stalled` is a flag that determines if transmissions are stalled in anticipation of an ACK.

4.4.2 Utility

Throughout this section, several utility functions will be used in the code listings. We detail these functions briefly below:

- `start_event_timer` queues a given timer with a given timestamp.
- `cancel_event_timer` cancels the given timer.
- `tcp_tb_in_slow_start` compares `Ntx` and `ssthresh_Ntx` to check whether or not TBTCP is in slow start.

Tracking transmitted packets

To maintain the `lostPackets` array we have used the Linux kernel's linked list implementation. We define a data structure to hold the information for each packet in the list:

```
1  struct tb_packet {
2      u32 Ntx;
3      u32 Nak;
4      u32 seq;
5      u64 time;
6      struct list_head list;
7  };
```

Listing 4.7: tcp_tb.c: `struct tb_packet`

We store the `Ntx` of the sent packet, the `Nak` at the time of transmission, the TCP sequence number and the time of transmission. We have to store the TCP sequence number to be able to know which packets to remove when acknowledgments are received.

To more easily manage this list in the rest of the implementation we created two utility functions `add_packet` and `remove_packets`.

```
1  static int add_packet(struct tcp_tb *ca, u64 bytes_sent,
   u32 Ntx, u32 Nak, u64 time) {
2      struct tb_packet *packet;
3      if (ca->lostPackets.next == NULL ||
   ca->lostPackets.prev == NULL) {
4          return -1;
5      }
6      packet = kmalloc(sizeof(struct tb_packet), GFP_ATOMIC);
7      if (!packet) {
8          return -1;
9      }
10
11     packet->Ntx = Ntx;
12     packet->Nak = Nak;
13     packet->seq = seq;
14     packet->time = time;
15
16     list_add_tail(&packet->list, &ca->lostPackets);
17
18     return 0;
19 }
```

Listing 4.8: tcp_tb.c: `add_packet`

`add_packet` adds a packet to the `lostPackets` list. This function is used in the `tcp_tb_pace` function to add a packet to the list when it is sent.

```
1   static int remove_packets(struct tcp_tb *ca, u32 seq) {
2       struct tb_packet *packet;
3       struct list_head *pos, *q;
4       int i = 0;
5
6       if (ca->lostPackets.next == NULL ||
    ca->lostPackets.prev == NULL) {
7           return -1;
8       }
9       list_for_each_safe(pos, q, &ca->lostPackets) {
10          packet = list_entry(pos, struct tb_packet, list);
11          if (before(packet->seq, seq)) {
12              list_del(pos);
13              kfree(packet);
14              i++;
15          }
16      }
17
18      return i;
19  }
```

Listing 4.9: tcp_tb.c: `remove_packets`

`remove_packets` removes all packets from the `lostPackets` list that have been acknowledged. This function is used in the `tcp_tb_pkts_acked` function to remove acknowledged packets from the list.

4.4.3 TCP callbacks

In this section, we will describe the callbacks used for our TBTCP module. This will give a general overview of the TBTCP module. TCP module callbacks are detailed in section 2.4.

init and release

The `init` and `release` callbacks will be responsible for initializing and cleaning the module state. `init` will in addition be responsible for the initial queuing of the event timer.

```
1   u64 now = ktime_get_ns();
2   u32 IW = tp->snd_cwnd;
3
4   ca->postRecovery = false;
5   ca->rtt = max(tp->srtt_us >> 3, 1U) * 1000;
6
7   ca->HighData = 1;
8   ca->ssthresh_Ntx = U32_MAX;
9   tp->snd_ssthresh = TCP_INFINITE_SSTHRESH;
10  ca->NlastAck = tp->snd_una;
```

```
11
12      ca->Ttx = now;
13      ca->Ntx = IW; // initial window size
14      ca->Nak = ca->Ntx;
15      ca->Tak = now + ca->rtt;
16      ca->stalled = false;
17      INIT_LIST_HEAD(&ca->lostPackets);
18
19
20      /* start initial timer */
21      start_event_timer(sk, timer, ca->Ttx);
```

Listing 4.10: tcp_tb.c: excerpt from `tcp_tb_init`

In `tcp_tb_init`, we first initialize the `tcp_tb` data structure. The TBTCP-specific variables are initialized according to Table 3.2. The `rtt` field is initialized to the smoothed RTT value from the TCP socket, as we have yet to receive an RTT sample from an ACK.

Lastly, we trigger the event timer with an instantaneous expiration. We are able to start the timer here, as `init` is called as TCP enters the `ESTABLISHED` state. The timer is set to trigger immediately as there is no need to pace the first packet in the flow. We therefore set the estimated ACK arrival time, `Tak` to the current RTT.

```
1       struct hrtimer *timer = (struct hrtimer*) &tp->event_timer;
2       cancel_event_timer(timer);
```

Listing 4.11: tcp_tb.c: excerpt from `tcp_tb_release`

When the transmission closes, `tcp_tb_release` will be invoked to cancel the timer in the case that it is still running.

ssthresh

We implement the mandatory `ssthresh` callback with the function `tcp_tb_ssthresh`. Specifically, this function is called when TCP enters recovery to calculate the new value of `ssthresh`. This is necessary as we want to set `ssthresh` to a value that corresponds to the sending rate that was used for the lost packet.

```
1     struct tcp_tb *ca = inet_csk_ca(sk);
2     struct tb_packet *lostPacketsFront =
      list_first_entry(&ca->lostPackets, struct tb_packet, list);
3     u32 ssthresh;
4
5     (...)
6
7     ssthresh = max(((lostPacketsFront->Ntx -
      lostPacketsFront->Nak) * beta) / TBTCP_BETA_SCALE, 2U);
8
9     (...)
10
11    ca->ssthresh_Ntx = ssthresh;
12    return ssthresh;
```

Listing 4.12: tcp_tb.c: excerpt from `tcp_tb_ssthresh`. `(...)` indicates that code that is not revlevant to this heading has been removed.

To achieve this, we find `Ntx` of the packet that was lost by checking the `lostPackets` list. The `Ntx` in this case describes the sending rate at the moment the lost packet was sent. Subsequently, we multiply with a module parameter `beta` and then divide this sending rate by `TBTCP_BETA_SCALE`, which is a constant that denotes the scale of which the `beta` parameter should exist in. `TBTCP_BETA_SCALE` is set to 1024, so to halve the sending rate, `beta` should be set to 512. Lastly, we set `ssthresh_Ntx` to the calculated value and return it.

event_handler

The `event_handler` callback is responsible for handling the event timer. This callback will fire upon timer expiration. Since we have not implemented the recovery mechanisms of TBTCP, the callback is trivial. It will simply call `tcp_tb_pace` to pace the next packet if the state allows for it and we are not in recovery.

If the timer fires after we should have received an ACK, we will set the `stalled` flag to true, to wait for an ACK before transmitting a new packet. The `stalled` flag will be read by the ACK-handler to handle the stall once an ACK is received.

```
1     if (ca->Ttx < ca->Tak && ca->state < TCP_CA_Recovery) {
2         tcp_tb_pace(timer);
3     } else {
4         ca->stalled = true;
5     }
```

Listing 4.13: tcp_tb.c: excerpt from `tcp_tb_event_handler`

tcp_tb_pkts_acked

The `tcp_tb_pkts_acked` callback is responsible for handling ACKs. It is called when an ACK is received. The callback will update the state of the TBTCP flow and call `tcp_tb_pace` if the `stalled` flag is active.

```
1     if (sample->rtt_us > 0)
2         ca->rtt = rtt_ns;
3
4     if (before(tp->snd_una, ca->NlastAck) || tp->snd_una ==
    ca->NlastAck) {
5         return; // ignore duplicate acks
6     }
7
8     ca->NlastAck = tp->snd_una;
9     removed = remove_packets(ca, tp->snd_una);
10
11    if (ca->state == TCP_CA_Loss) {
12        return;
13    }
14
15    if (ca->state == TCP_CA_Recovery) {
16        ca->postRecovery = true;
17        return;
18    }
19
20
21    if (ca->postRecovery) {
22        ca->Ntx = tp->snd_ssthresh * (tp->snd_ssthresh - 1) /
    2 + 1;
23        ca->Nak = ca->Ntx;
24        ca->Ttx = now;
25        ca->Tak = now + ca->rtt;
26        ca->postRecovery = false;
27    } else if (removed > 0) {
28        ca->Tak = now;
29        kernel_fpu_begin();
30        delta = delta_time_mult_rtt(ca, ca->Nak,
    sample->pkts_acked, ca->rtt);
31        kernel_fpu_end();
32        ca->Tak = ca->Tak + delta;
33        ca->Nak = ca->Nak + sample->pkts_acked;
34    }
35
36    if (ca->stalled) {
37        ca->stalled = false;
38        tcp_tb_pace(&tp->event_timer, now);
39    }
```

Listing 4.14: tcp_tb.c: excerpt from `tcp_tb_pkts_acked`

Here the RTT is updated on every ACK to make sure we always keep the latest RTT sample. We then check if the ACK is a duplicate. If it is, we simply ignore it. If it is not, we remove all packets up to the ACKed

sequence number from the list of un-ACKed packets.

If TCP is in the `LOSS` state, we return, as we let the TCP stack handle this. If we are in recovery, we set the `postRecovery` flag to true and return. This flag will be used to determine if we should reset the state of the flow when we exit recovery.

If we are not in recovery, we check if we are in the post-recovery state. To remedy not knowing the timestamps of potential packet transmissions while TCP is in recovery, we instead reset the TBTCP state. To do this, we must first set a value for `Ntx`, to start at an appropriate sending rate. We calculate this by translating the standard TCP `ssthresh` value to the corresponding `Ntx`. This is done according to this quadratic function:

$$\texttt{ssthresh} * (\texttt{ssthresh} - 1)/2 + 1;$$

We then update `Ttx` and `Nak` to expect the next ACK to arrive at an RTT from `now`.

If no packets were removed from the `lostPackets` list based on the sequence number in the ACK, we simply update the next expected ACK arrival time to wait for the next ACK. This could occur if we receive ACKs out-of-order. The execution flow is visualized in Figure 4.2.

Remaining callbacks

To fulfill the requirements for implementing a TCP module, which involves the implementation of the required callbacks, we have opted to use the Reno implementation for the remaining callbacks. We choose to not go into detail on these callbacks, as they are not relevant to our TBTCP implementation. These callbacks are *cong_avoid* and *undo_cwnd*, both of which are essential for maintaining the `cwnd` for when TCP enters recovery.

4.4.4 Other functions

In this section, we will describe the other functions that are used in the implementation of TBTCP.

tcp_tb_pace

The `tcp_tb_pace` function is responsible for pacing the next packet. It will calculate the time for the next packet to be sent and queue a timer to trigger at that time.

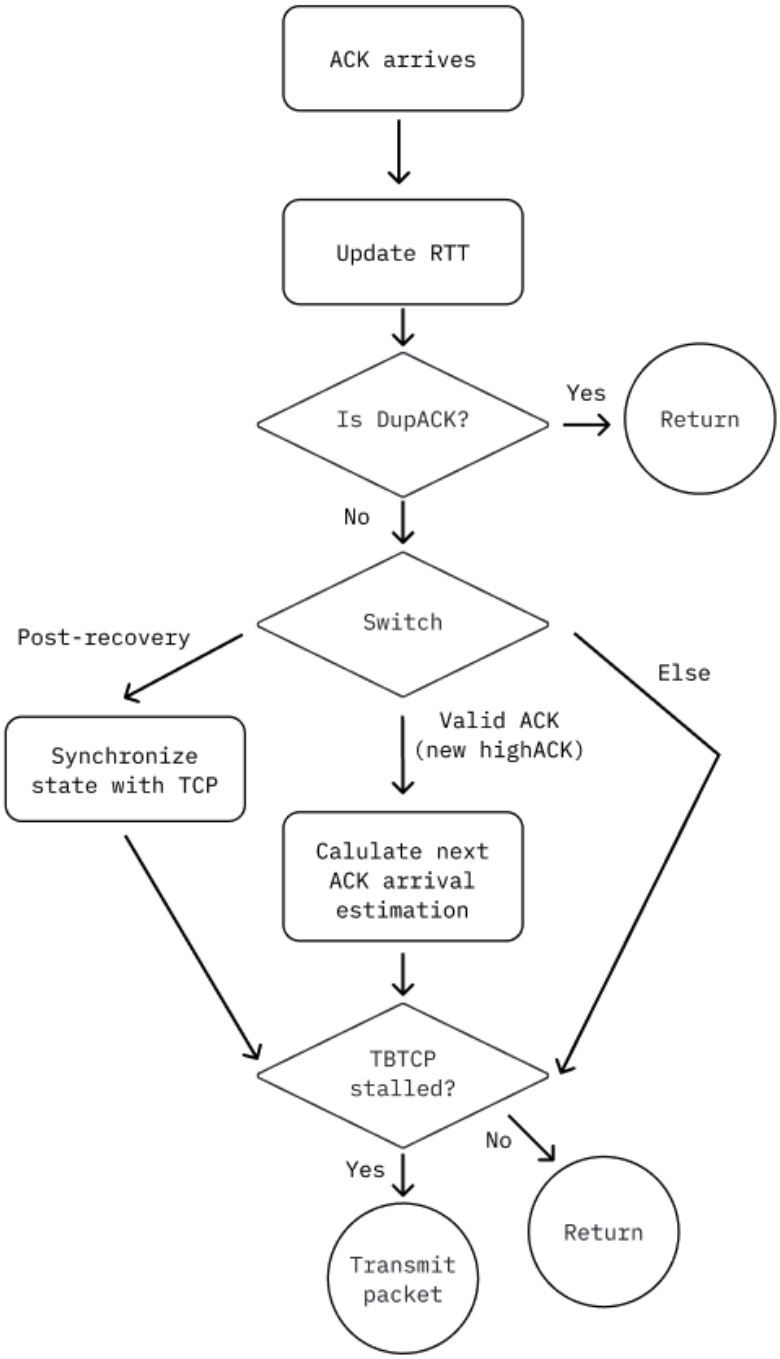


Figure 4.2: Flowchart detailing an ACK arrival event. The flowchart is based on the code in Listing 4.14.

```
1     snd_nxt = tp->snd_nxt;
2     if (!sock_owned_by_user(sk)) {
3         tcp_write_xmit(sk, tcp_current_mss(sk), TCP_NAGLE_OFF,
      2, sk_gfp_mask(sk, GFP_ATOMIC));
4     } else {
5         start_event_timer(sk, timer, ktime_get_ns());
6         return;
7     }
8
9     /* start a new initial timer if nothing was sent */
10    if (tp->bytes_sent == 0) {
11        ca->Ttx = ktime_get_ns();
12        ca->Tak = ktime_get_ns() + ca->rtt;
13
14        start_event_timer(sk, timer, ca->Ttx);
15        return;
16
17    }
18
19    add_packet(ca, snd_nxt, ca->Ntx, ca->Nak, now);
20    ca->HighData = ca->HighData + 1;
21
22    kernel_fpu_begin();
23    pacingTime = delta_time_mult_rtt(ca, ca->Ntx, 1, ca->rtt);
24    kernel_fpu_end();
25
26    ca->Ntx = ca->Ntx + 1;
27    ca->Ttx = now + pacingTime;
28
29    /* queue next event */
30    start_event_timer(sk, timer, ca->Ttx);
```

Listing 4.15: tcp_tb.c: excerpt from `tcp_tb_pace`

Here we call `tcp_write_xmit` to transmit the next packet. We invoke `tcp_write_xmit` with the `push_one` argument equal to 2. This tells TCP to force the transmission of at most one packet, regardless of the `cwnd` quota. We then add the transmitted packet to the list of un-ACKed packets. Subsequently, we calculate the time for the next packet transmission and queue a timer to trigger at that time.

We use the total number of bytes sent from the TCP socket as the cumulative sequence number when we add packets to our list of un-ACKed packets. This is because the TCP socket will send all packets in order, so the cumulative sequence number will be the same as the sequence number of the last packet sent. This corresponds to the cumulative bytes ACKed in our ACK handler where we remove packets from the list.

We have an extra check to see if `tcp_write_xmit` actually sent a packet. If it did not, we set `Ttx` and `Tak` to the initial values and restart the timer. We saw this occurring occasionally at the initial phase of a transmission. We suspect this is due to an empty send buffer. The complete execution flow is visualized in Figure 4.3.

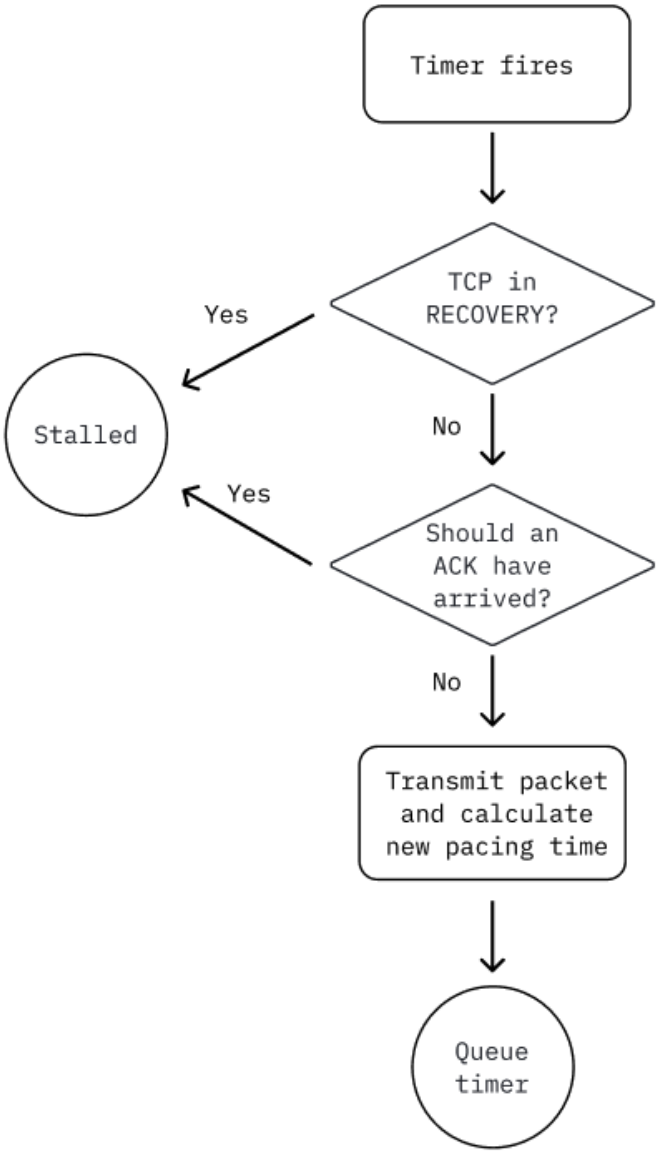


Figure 4.3: Flowchart detailing a timer expiration event. The flowchart is based on the code in Listing 4.15.

4.5 Implementing floating-point support

In this section, we will detail how we made our TCP module support floating-point calculations. We will first speak on why we need this feature. We will then look into how, if at all, the kernel handles floating-point calculations. Lastly, we will describe how we handle floating-points in our implementation.

The need for floating-point calculations

As discussed in chapter 3, TBTCP queues a timer for every single packet to transmit. To determine the pacing time from one transmission to the next, the function `delta_time` is used. In the basic implementation TBTCP, the `delta_time` calculation for the slow-start branch is based on the logarithm base 2 of `(1+(k/n))`. This results in a factor `f`, where `f < 0`. The problem remains the same in the congestion-avoidance branch, where instead of logarithms, square-root calculations are used. Division operations are also used in both branches. To be able to calculate these factors at all, we will need support for floating-point calculations.

Kernel support for floating-points

Currently, the kernel is not engineered to support floating-point operations in kernel space. The main reason for this is that it is expensive to support the use of a floating-point unit (FPU). In essence, this would require the kernel to save and restore additional state every time a kernel task is preemptively switched out.

Another reason for the lack of FPU support is that in most cases, the kernel does not need floating-points. Most operations either do not require floating-point precision or can be converted to using integers. In our case, conversion to using integers is not trivial.

Enabling FPU support

Even though it is not encouraged by the kernel development community, there is a way to enable support for FPU operations. In a mail from Linus Torvalds he speaks on how to perform floating-point calculations "safely" on `x86` architectures [64]. Here, he explains the use of two macros to disable preemption temporarily.

```
1 kernel_fpu_begin();
2 /* non-premptive code goes here */
3 kernel_fpu_end();
```

Listing 4.16: Usage of FPU macros

The use of `kernel_fpu_begin` and `kernel_fpu_end` will temporarily disable preemption for the enclosed instructions[2]. This sequence will initially disable preemption with the macro `preempt_disable`. This is followed by saving the current FPU state to memory through the use of the `FXSAVE` instruction. This is done to preserve the state of the current process. The kernel is now able to manipulate these registers freely without corrupting the state of user processes. It is important to note that when writing non-preemptive code, to not execute code that has the potential to cause faults or traps to prevent undefined behavior.

Floating point calculations

As indicated, calculating logarithms and square roots of floating point numbers proved to be non-trivial in the kernel. To work around this, we made use of approximations of these calculations.

Calculating the base 2 logarithm

To achieve this, we make use of a log2 approximation implemented in the Fastapprox library [66]. The function shown in Listing 4.17 is borrowed from this library which implements an approximation of a log2 calculation. This fits our use case well, as we are able to trade off accuracy in the calculation as there is significant overhead related to queuing the timer.

```
1  float log2_of_number(float x) {
2      union {
3          float f;
4          uint32_t i;
5      } vx = {x};
6      union {
7          uint32_t i;
8          float f;
9      } mx = {(vx.i & 0x007FFFFF) | 0x3f000000};
10     float y = vx.i;
11     y *= 1.1920928955078125e-7f;
12     return y - 124.22551499f
13             - 1.498030302f * mx.f
14             - 1.72587999f / (0.3520887068f + mx.f);
15 }
```

Listing 4.17: Usage of FPU macros

Calculating square roots

To calculate the square root of a floating point number we have made use of an iterative approach. The Babylonian or Heron's method is a way of

[2] `linux-kernel-5.19 | x86 fpu api.h` [65].

approximating the square root of a number iteratively [67]. It works by starting with an initial guess and tolerance, then iterating on the guess to approach a tolerable approximation.

```
double sqrt_double(double x) {
    double guess = x / 2;
    double epsilon = 0.000001;
    int i;

    for (i = 0; i < 12; i++) {
        double diff = guess * guess - x;
        if (diff < 0) {
            diff = -diff;
        }
        if (diff < epsilon) {
            break;
    }
        guess = (guess + x / guess) / 2;
    }

    return guess;
}
```

Listing 4.18: Iterative calculation of square roots

Delta calculation

Given that we can now perform floating-point operations, we can compute the delta calculation described in chapter 3. The `delta_time_mult_rtt` function accepts a sequence number (`Ntx`), time steps (`k`) and the RTT, to calculate the pacing time for the next packet transmission.

```
    float res, div;
    if (tcp_tb_in_slow_start(ca)) {
        div = (float)((float) k / (float) seq_num);
        res = log2_of_number(1 + div);
        return res * rtt;
    } else {
        res = (float)(sqrt_float((8*(seq_num + k)-7)) /
    (float)2) - (float)(sqrt_float((float)(8*seq_num - 7)) /
    (float)2);
        return res * rtt;
    }
```

Listing 4.19: delta_time_mult_rtt

Depending on whether or not TBTCP is in Slow Start, determined by the return value of the `tcp_tb_in_slow_start` function, influences the increase function to use. If TBTCP is in Slow Start, the increase is exponential, while a linear increase is used in Congestion Avoidance. Multiplying the RTT with the resulting factor equals the pacing time to be used for the next

packet transmission. This function requires the use of `kernel_fpu_begin` and `kernel_fpu_end` macros.

Compiler attributes

To convince the compiler to compile these functions, we give both functions an attribute of `sse2` to indicate that they are to be compiled with `sse2` instructions. `sse2` instructions are a set of x86 instructions that extend the capabilities related to floating point operations [68]. This is done by prefixing the function definition with `__attribute__((target("sse2")))`.

4.6 Configurable module parameters

To make the TBTCP module configurable at runtime, we add two parameters, `beta` and `ssthresh_cwnd_based`. The `beta` parameter multiplied by a constant `TCP_BETA_SCALE` (set to 1024) is used in the *ssthresh* calculation to facilitate a user-defined backoff factor. For example, to use a backoff factor of 0.5, the `beta` parameter should be set to 512.

The `ssthresh_cwnd_based` parameter is used to switch to a conventional *ssthresh* calculation rather than using the TBTCP calculation. The main difference between these is that TBTCP calculates the new *ssthresh* value by multiplying the backoff factor with the `cwnd` (`Ntx`) value at the time a loss occurred. The conventional calculation multiplies the current `cwnd` with the backoff factor.

```
1     struct tcp_tb *ca = inet_csk_ca(sk);
2     struct tb_packet *lostPacketsFront =
    list_first_entry(&ca->lostPackets, struct tb_packet, list);
3     u32 ssthresh;
4
5     if (ssthresh_cwnd_based) {
6         ssthresh = max((tcp_sk(sk)->snd_cwnd * beta) /
    TBTCP_BETA_SCALE, 2U);
7     } else {
8         ssthresh = max(((lostPacketsFront->Ntx -
    lostPacketsFront->Nak) * beta) / TBTCP_BETA_SCALE, 2U);
9     }
10
11    ca->ssthresh_Ntx = ssthresh;
12    return ssthresh;
```

Listing 4.20: Modified `tcp_tb_ssthresh` with runtime parameters.

Chapter 5

Implementation assessment

In this chapter we first describe how we evaluate our implementation. We detail how and on what we run our kernel and how we record and assess the features of our TBTCP implementation.

We perform experiments to test and document specific features of our implementation, to ensure they work as expected. Tools and experiments are automated through a collection of scripts.

5.1 Testbed

We have conducted all our assessment experiments on physical machines. We have chosen this, as it is the most realistic environment to test the limitations of our implementation. This is particularly true when testing the capabilities of the `hrtimer` as our implementation is sensitive to inaccuracies and delays that can be introduced by virtualization. We have three Linux machines running the following specification:

- 64-bit, x86 Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz CPU
- 64 GB memory
- 10G X550T Network Interface Card
- Linux 5.10

Our topology is visualized in Figure 5.1. Here, the client (sender) is connected to the router through a Gigabit link. The egress path on the router connects to the server (receiver) and is configured to act as the bottleneck link. It accomplishes this by stacking several queuing disciplines (qdiscs); `netem` to add delay, `htb` to set bandwidth and a `bfifo` queue to act as the buffer.

The client machine is running our modified compiled kernel. In an attempt to isolate the behavior of our implementation, we have disabled

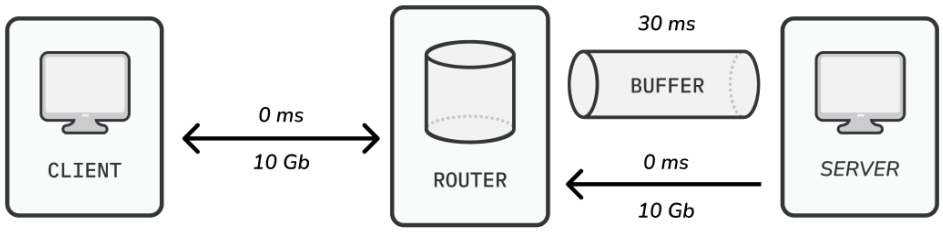


Figure 5.1: Testbed topology running on physical desktop machines. The `buffer` component is the egress interface from the router to the server. It is configured with different delay, bandwidth and buffer sizes across our experiments.

the following hardware offloading optimization mechanisms - TSO, GSO, LRO, GRO and UFO. For TCP-specific mechanisms, we have disabled Explicit Congestion Notifications (ECNs) and enabled the use of Selective Acknowledgements (SACKs).

5.2 Testing and measurement tools

iPerf

To initiate TCP flows, we have chosen `iPerf` as our tool of choice. `iPerf` generates network traffic by transferring arbitrary data from a client to a server. Among the supported benchmarking metrics are throughput and latency [69]. It supports a variety of options such as transfer duration, socket buffer sizes, and which congestion control algorithm to use.

netem

`netem` is a qdisc that supports Network Emulation. Amoung its features are emulation of packet loss, delay, duplication and corruption [70]. We only utilized the delay functionality in our assessment.

Tcpdump

To capture packets for later analysis we use `tcpdump`. This application can monitor a given network interface and write the packet data to a `pcap` file [71].

Synthetic Packet Pairs

SPP is an algorithm for measuring the RTT in networks. It works by measuring packets IP traffic between hosts without requiring precise time synchronization. We used this post transmission to analyze the `.pcap` files

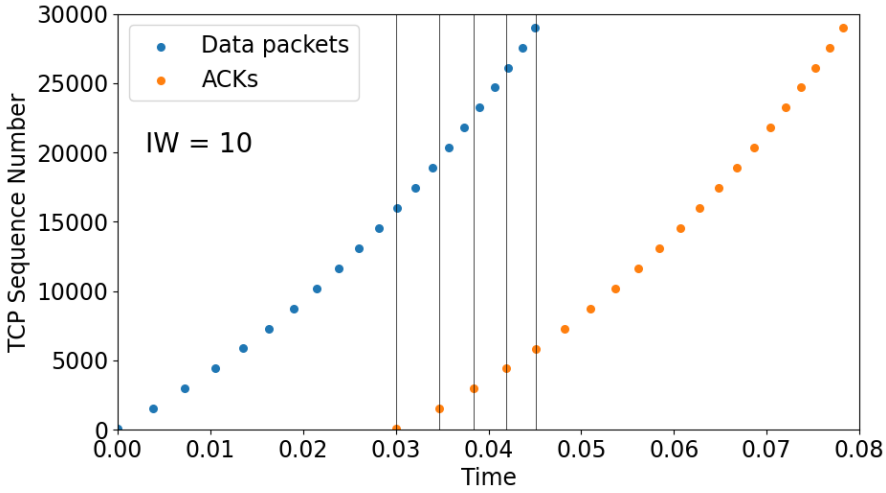


Figure 5.2: Time sequence plot displaying TBTCP per-packet pacing in Slow Start.

of the client and server. The output is a list of RTT measurements and a timestamp for each entry [72].

5.3 Experiments

Here we will detail the experiments we have conducted to assess the effectiveness of our implementation. We will test various aspects of the implementation, to determine if it exhibits the intended behavior.

5.3.1 Pacing behaviour

TBTCP is inherently paced, meaning all data packets are to be transmitted at a specific point in time given by `delta_time`. This is done by calculating the delta between packets using the sequence number `Ntx` - when a packet is transmitted, `Ntx` is incremented, and the timer is queued with the calculated delta. The rate of increase is exponential in Slow Start, and linear in Congestion Avoidance. This pacing scheme is intended to mimic the behavior of regular TCP, without relying on the `cwnd`. The pacing scheme can also be altered by simply modifying the calculation in `delta_time`.

Pacing in Slow Start

With an initial window of 10 (as is standard in Linux) the exponential per-packet pacing pattern is showcased in Figure 5.2. Visually, this results in a smooth initial increase of the transmission rate. This concept is further visualized as a delta time plot in Figure 5.3. This depicts fine-grained

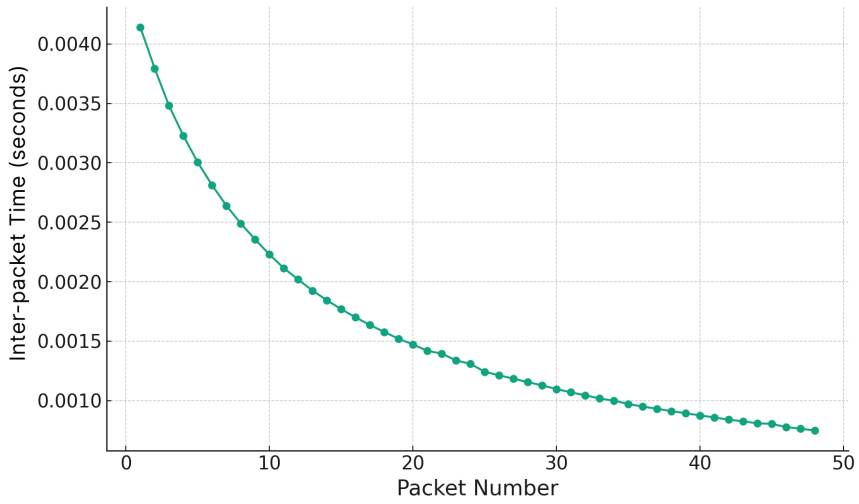


Figure 5.3: Delta time plot displaying TBTCP per-packet pacing in Slow Start.

pacing, where, after a packet transmission, a new delta is calculated to queue the timer for the next packet.

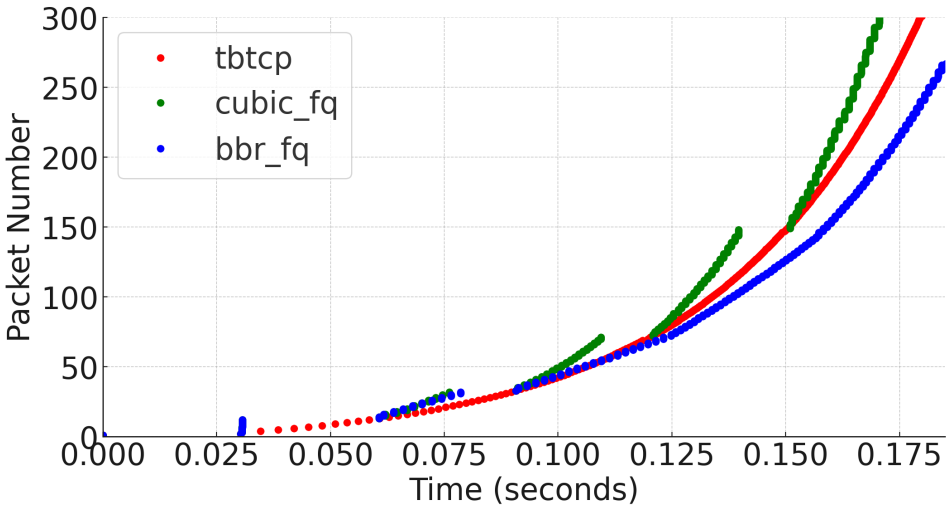


Figure 5.4: Comparing TBTCP with other pacing schemes.

The per-packet pacing in TBTCP separates itself when compared with other pacing schemes. In Figure 5.4, we compare TBTCP with BBR and CUBIC, both utilizing FQ/pacing. TBTCP exhibits a significantly more consistent curve due to the per-packet pacing, so much so, that the FQ/pacing variants appear bursty in comparison.

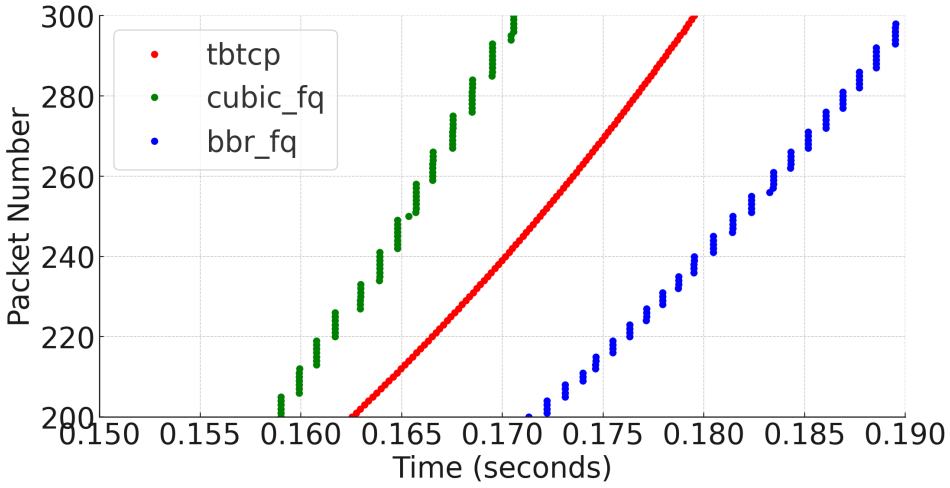


Figure 5.5: Zoomed in version of Figure 5.4.

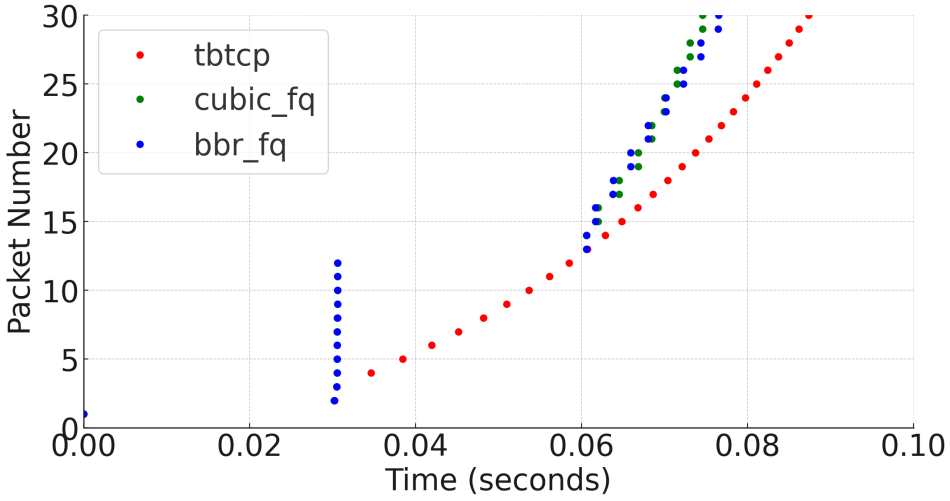


Figure 5.6: Comparing initial window (IW) pacing.

Pacing the initial window It is notable that neither internal pacing nor FQ/pacing supports pacing the initial window (IW) of packets. Linux uses an initial window value of 10 by default. On the contrary, in TBTCP, the initial window of packets remains fully paced. The reason for this is that TBTCP lacks a concept of an initial window. Rather, it is based on an *initial rate*, which is calculated from the initial window size. Figure 5.6 showcases this. While studying this diagram, keep in mind that the initial two plotted packets are related to the handshake process.

Pacing in Congestion Avoidance

The pacing pattern in Congestion Avoidance is linear, however, the pacing is still done on a per-packet level. Figure 5.7 displays this behavior. The

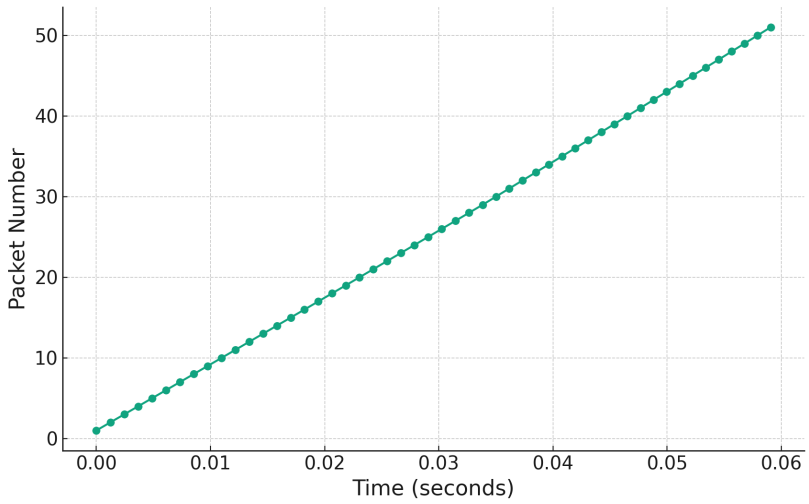


Figure 5.7: Delta time plot displaying TBTCP per-packet pacing in Congestion Avoidance.

increase pattern can now be viewed as gradual and linear. When we view the zoomed-in delta time plot in figure Figure 5.9, we can see how the inaccuracies introduced by various sources (timer, write to socket, kernel scheduler) affect the pacing pattern. These fluctuations are on the scale of mere microseconds, but they are noteworthy nonetheless.

5.3.2 Stalling logic

A vital part of the TBTCP algorithm is the calculation of `Tak` from algorithm 2, which denotes the expected arrival time of the next expected ACK. It is used to determine if we are allowed to send new data, or if the algorithm should stall to wait for an ACK.

Expected behaviour

If the client does not receive ACKs within the expected time `Tak`, the sending should stall. We expect at most 1-2 packets to be sent, from the moment where the last ACK was received.

Experiment: discarding acknowledgements

We will conduct this experiment by artificially discarding incoming ACKs at a random point in time during the Congestion Avoidance phase of a flow. We do this by creating an `iptables` rule on the server to temporarily drop all outgoing packets to the client.

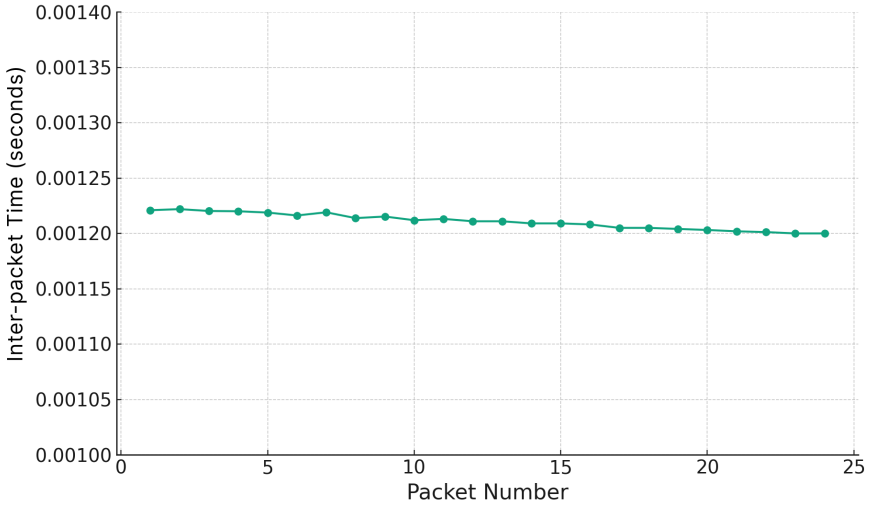


Figure 5.8: Delta time plot displaying TBTCP per-packet pacing in Congestion Avoidance.

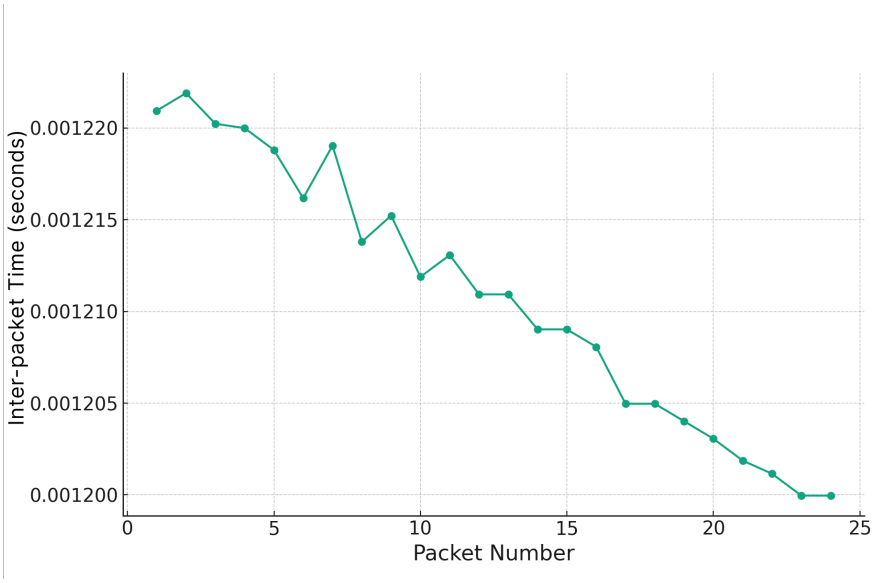


Figure 5.9: Zoomed in version of the delta time plot from Figure 5.8.

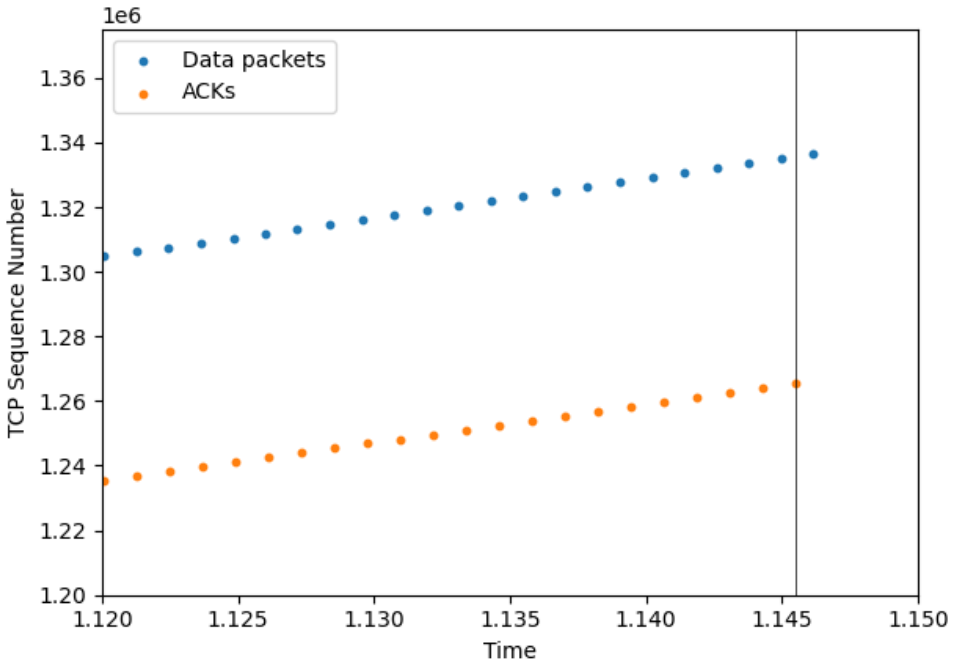


Figure 5.10: Time/Sequence plot that displays the arrival of ACKs and transmission of new data packets, both captured on the client interface. The vertical line on the x-axis marks the point in time where the last ACK was received. Parameters used: 10 Mbit bottleneck-bandwidth, 30 ms RTT, 1xBDP queue length.

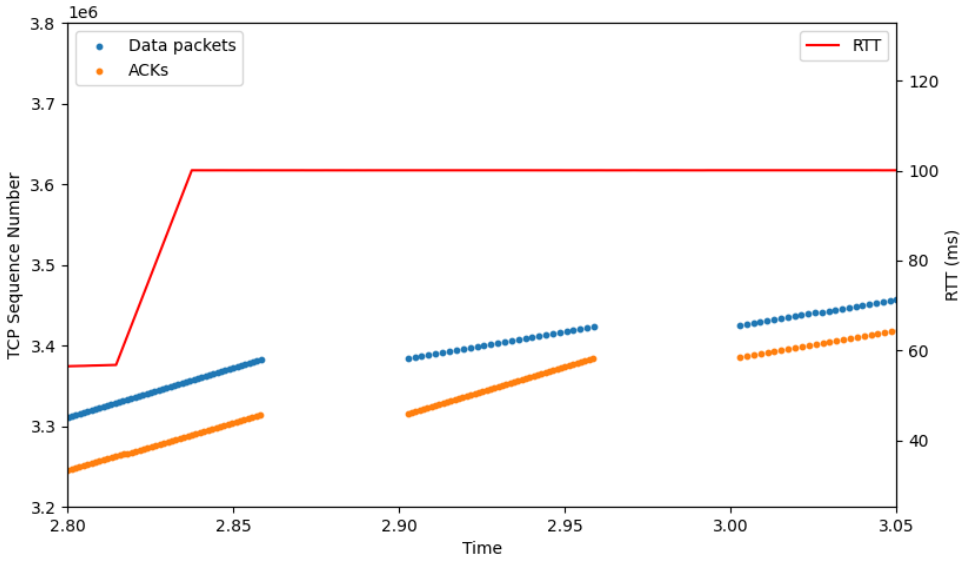


Figure 5.11: Time/Sequence plot that displays the arrival of ACKs and transmission of new data packets with RTT measurements.

Figure 5.10 displays the arrival of ACKs and the transmission of new data packets over time. We see that after receiving the last ack, the client is allowed to transmit a single data packet before it stalls.

Experiment: sudden increase in RTT

The calculation of the next expected ACK arrival time, `Tak` is based on the RTT. Consequently, stalling can occur when sudden increases in RTT are detected. In a realistic scenario, sudden RTT increases could be introduced when sharing a link with a bursty flow, or when new flows are initiated over the same link. This experiment is conducted by increasing the RTT of a link that was previously 30ms, to 100ms, using `netem`.

In Figure 5.11 we observe an instantaneous increase in RTT from approximately 60ms to 100ms. This change results in delaying incoming ACKs, causing the client to stall. If we examine the effective RTT increase of about 40ms, we notice that the first stall lasts for about this duration. When the ACK finally arrives, sending resumes, only to stall again as subsequent ACKs are delayed due to the initial stall.

Figure 5.12 showcases the same experiment on a larger timescale. We notice that this subsequent stall pattern smoothes out when the sending rate increases. When the sending rate increases over time, the stalling duration also decreases gradually before the sending rate finally catches up. We attribute this to a consequence of ACK-clocking (`Tak`), as the same behavior was observed when testing TCP Reno (Figure 5.13).

[2]Parameters used in *Experiment: sudden increase in RTT*: 10 Mbit bottleneck-bandwidth, 30ms RTT, 1xBDP queue length.

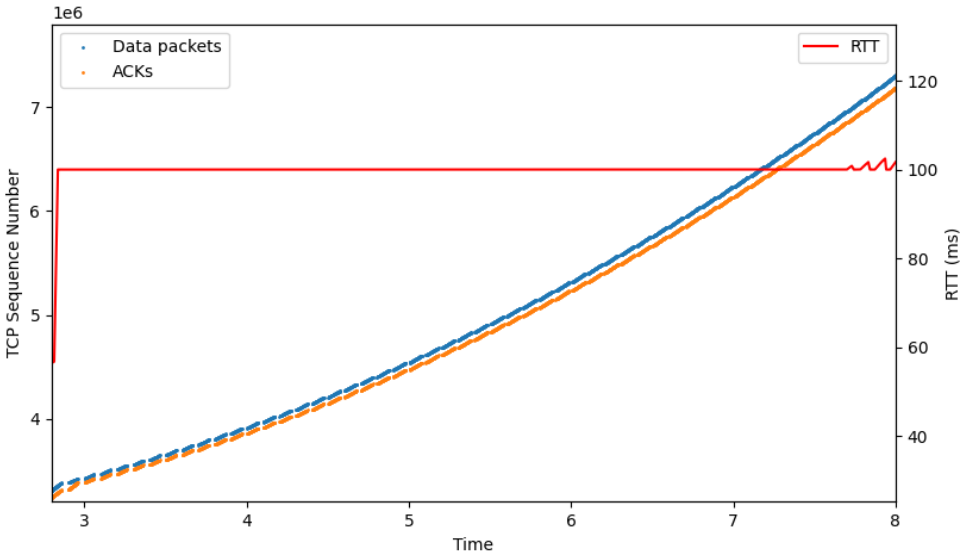


Figure 5.12: Zoomed out version of Figure 5.11.

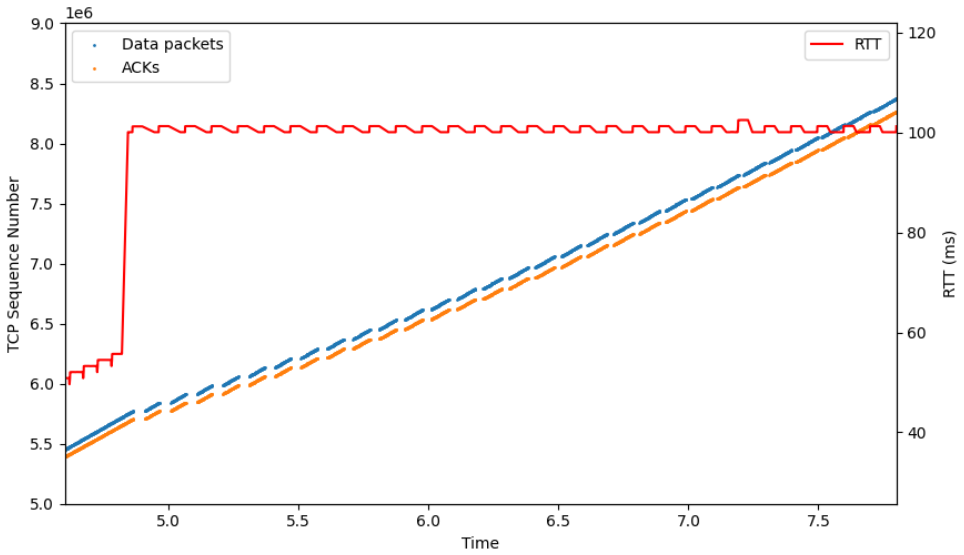


Figure 5.13: Plot showcasing TCP Reno behavior when experiencing sudden increases in RTT.

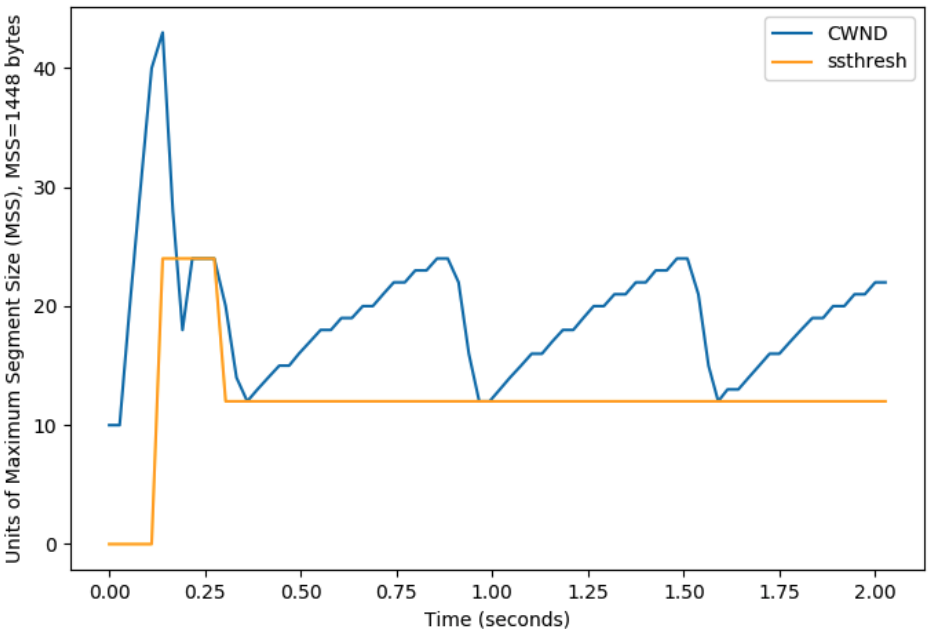


Figure 5.14: Plot showcasing the congestion window (*cwnd*) and Slow Start threshold (*ssthresh*) values over time for a Reno flow. RTT: 30ms, Bottleneck-bandwidth: 5 Mbit, BDP: 18750 bytes. Bottleneck-buffer: 18750 bytes. Average sampling rate 27ms.

5.4 Backoff

TCP Reno in Linux will almost always experience a double `ssthresh` back-off after Slow Start. This is because `ssthresh` is set to half of the `cwnd` at the point when loss is detected, and that is almost always exactly too much, resulting in an additional backoff.

In TBTCP we do the `ssthresh` back-off a little differently. We keep track of the in-flight packets throughout the entire transmission. That way we know the rate at which the loss occurred, instead of when the loss was detected. We can then set `ssthresh` to half of the in-flight packets at the point where the loss occurred. This way, we can avoid the unwanted double back-off.

The double back-off of Reno is illustrated in the plot in Figure 5.14. The plot in Figure 5.15 shows the backoff behavior of TBTCP. We see that TBTCP does not experience the double back-off, and instead resumes at a more appropriate sending rate.

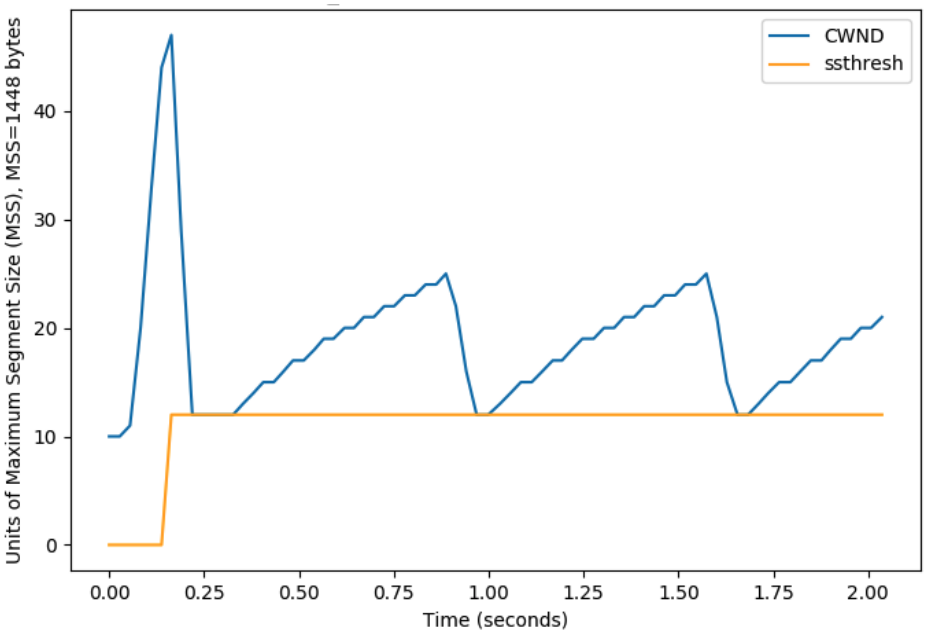


Figure 5.15: Plot showcasing the congestion window (*cwnd*) and Slow Start threshold (*ssthresh*) values over time for a TBTCP flow. RTT: 30ms, Bottleneck-bandwidth: 5 Mbit, BDP: 18750 bytes. Bottleneck-buffer: 18750 bytes. Average sampling rate 27ms

5.5 Pursuing peak throughput

In this section, we address the steps taken to reach maximum throughput values. This was an iterative process where we continually assess and optimize the implementation. As such, this section will not only contain the assessment of the throughput, but also how we improved the implementation in the process.

The following subsections highlight the specific challenges we faced and our solutions to reach peak throughput values. When we came across issues that were not fully resolved, suggestions are provided on how they might be addressed in the future.

5.5.1 Timer overhead

From the research questions in section 1.3, we ask about the correlation between the performance bottleneck of our implementation and the performance of the *hrtimer*. TBTCP leverages a single high-resolution timer (*hrtimer*) to insert a delay between packet transmissions. To adhere to the sending rate given by the calculated delta from *Ntx*, it is crucial to minimize discrepancies between the calculated *Ttx* and the actual expiration time of the timer. In our tests, a failure to conform to the sending rate given by *Ntx*, concurrent with an increasing *Nak* as ACKs are being received, was observed to precipitate a gradual deterioration of the sending rate until it

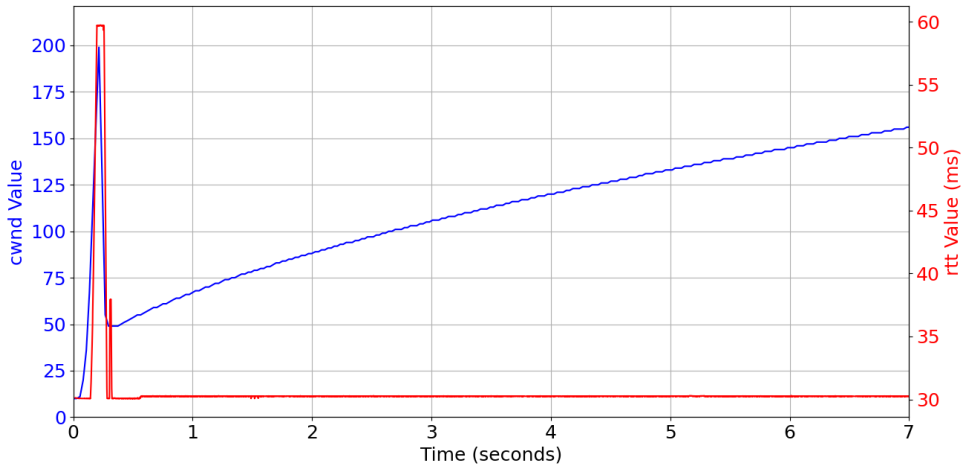


Figure 5.16: This plot displays the deterioration of the sending rate after Slow Start. The sending rate deteriorates to a packet-conserving state. This is indicated by the slow increase of the congestion window, while the RTT barely moves. Parameters used: 30ms RTT, 20Mbit link, 1xBDP queue.

becomes strictly packet-conserving. In practice, this meant that the link would remain underutilized for the rest of the transmission, resulting in no losses and a near-constant RTT, as can be seen in Figure 5.16. Figure 5.17 shows the stagnating throughput for the same transmission.

In the scenario where a packet is set to be transmitted at time Ttx given by the rate Ntx, the subsequent ACK should arrive by time Tak given by Nak. However, if a delay d is introduced from timer overhead, the packet is instead transmitted at time $Ttx + d$. If the next packet to be sent is queued relative to this delayed expiration time, the added delays accumulate, effectively displacing the intended timeline of packet transmission.

The problem becomes apparent as the expected ACK arrival time, Tak, fails to compensate for this accumulated delay in the sending rate. If the deviation between Ttx and Tak becomes sufficiently large, we see from the algorithm that the condition for pacing ($Ttx < Tak$) will fail, causing stalls. The occurrence of undesired stalls diminishes the difference between Ntx and Nak, primarily due to a gradual decrease in in-flight packets. This effectively means that we expect ACKs faster than we can transmit new packets, as Nak grows faster than Ntx.

To investigate this, we measured the timer overhead in Figure 5.18. This process entailed comparing the discrepancies between the intended expiration time Ttx and the actual expiration time *now* (obtained via *ktime_get_ns()*). The *now* value is compared with Ttx in the TBTCP module *event_handler*, within a critical region where the socket is locked. It is noteworthy that these measurements also include the interval of time from the firing of the timer until execution reaches the *event_handler*.

In our implementation, we account for these timer delays by queueing the subsequent timer based on the enqueued expiration time Ttx, rather than

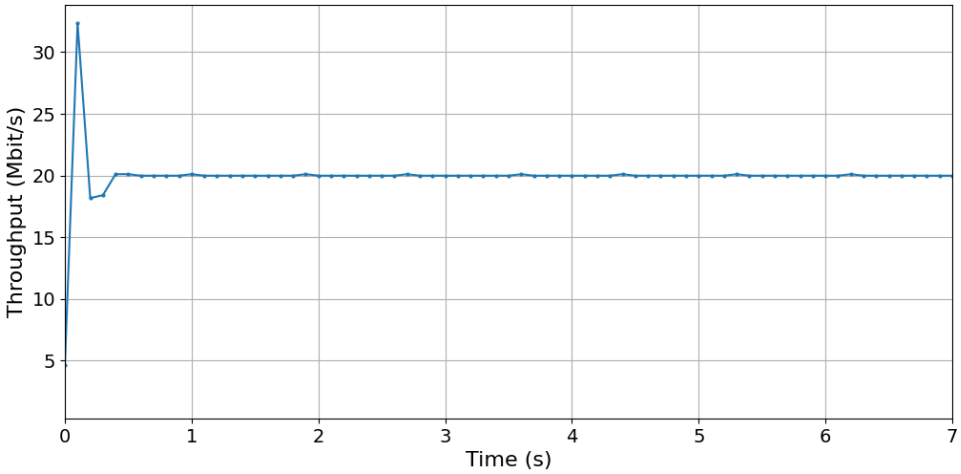


Figure 5.17: This plot displays the stagnating throughput measured on a 100ms interval. Ref. Figure 5.16 for transmission parameters.

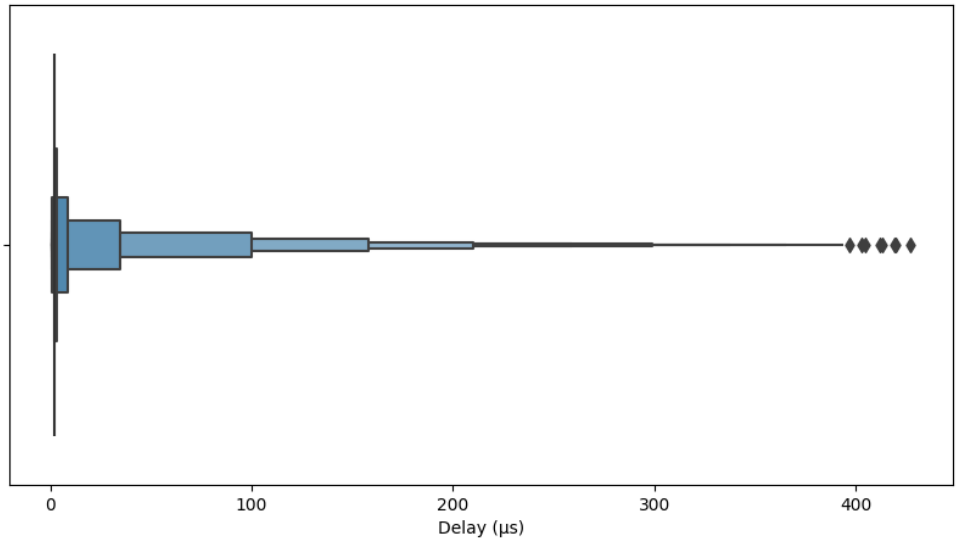


Figure 5.18: A Letter-Value Plot of the intended versus the actual expiration time of the *hrtimer*, rounded to the nearest microsecond. The boxes represent different quantiles in our data. The median delay was observed to be 2 µs, while the average delay was 6.32 µs. Over 40 000 measurements are plotted.

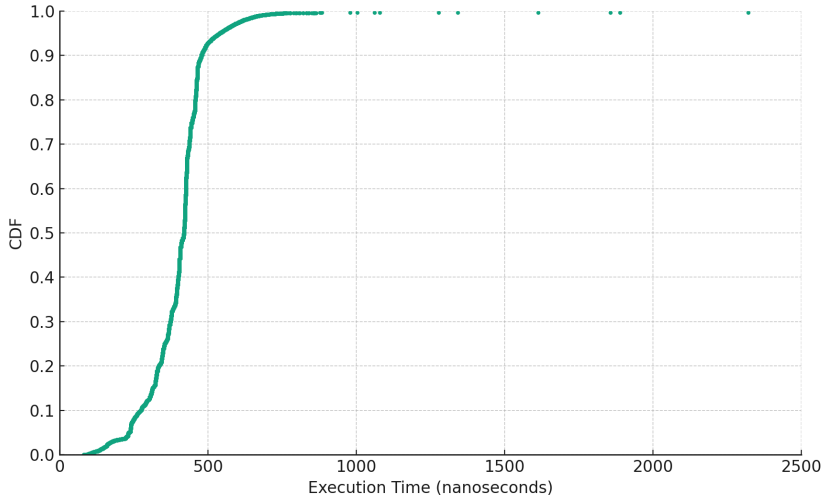


Figure 5.19: A CDF of the execution time of a delta calculation in nanoseconds. The median execution time was observed to be 420 ns, while the average execution time was 442 ns. Outliers have been removed for more clarity in the data.

the actual expiration time *now*. This mitigates the effects of timer overhead, as we effectively circumvent displacing the pacing timeline. However, this approach comes with a trade-off, as *Ttx* has the potential to lag behind *now* when the sending rate demands that the timer fires with intervals smaller than the timer overhead. In addition, one has to account for the overhead related to tasks such as packet transmission (Figure 5.20) and delta calculation (Figure 5.19).

This requires modifications in `tcp_tb_pace`, where instead of using `ktime_get_ns()` to fetch the current timestamp, we use the timestamp that the expired timer was enqueued with (`Ttx`). This timestamp is then added with the calculated delta to determine the next expiration time.

5.5.2 iPerf buffer sizes

When running `iPerf` on higher bandwidths, we encountered issues related to the size of the socket buffer, specifically on the client side. `iPerf` will push data to the TCP socket in intervals, thereby filling up the buffer. Based on our findings, the combination of small socket buffers and high sending rates may potentially lead to a scenario where the sending rate exceeds the rate at which `iPerf` can push data to the socket. This resulted in significant gaps in the sending pattern, where TBTCP was shown to exhibit bursty behavior when the sending finally resumed.

The behavior was identified in the packet trace, where a data packet with the `PSH` flag set was observed. After said packet, there was a gap in

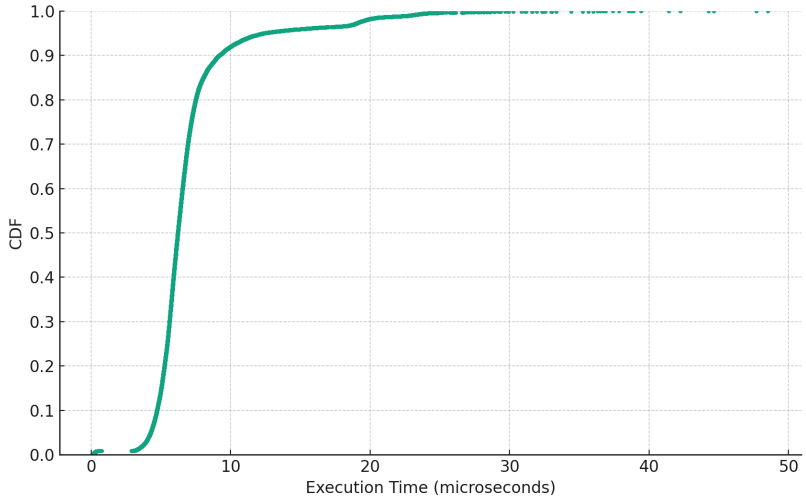


Figure 5.20: A CDF of the execution time of the function *tcp_write_xmit*, reponsible for queuing packets for transmissions. The median execution time was observed to be 6 µs, while the average execution time was 6.95 µs. Outliers have been removed for more clarity in the data.

the sending rate for multiple milliseconds, although ACKs were being received. The `PSH` flag indicates that the receiver should not wait for subsequent data packets before pushing data to the application [5]. If the `PSH` flag is not set, the receiver may combine the data from subsequent packets before pushing data to the application. The client may set this flag when transmitting the last buffered segment from the application. This indicated that the behaviour could be introduced by `iPerf`. We theorize that this frequent writing to the TCP socket results in a *lock-out* effect, where the TCP stack is unable to use the socket when it is being written to. This causes the TBTCP timer to fall behind, leading to a burst of packets when sending resumes, ref. Figure 5.21.

To limit overhead from `iPerf` data writes on connections with higher bandwidths, we may increase the socket buffer write size on the client. On Linux systems, the maximum send socket buffer size is set in `/proc/sys/net` via the option `wmem_max` [73]. Figure 5.22 displays the results of an increased client write buffer. There are still short gaps in the sending, however the interval between these gaps are significantly larger.

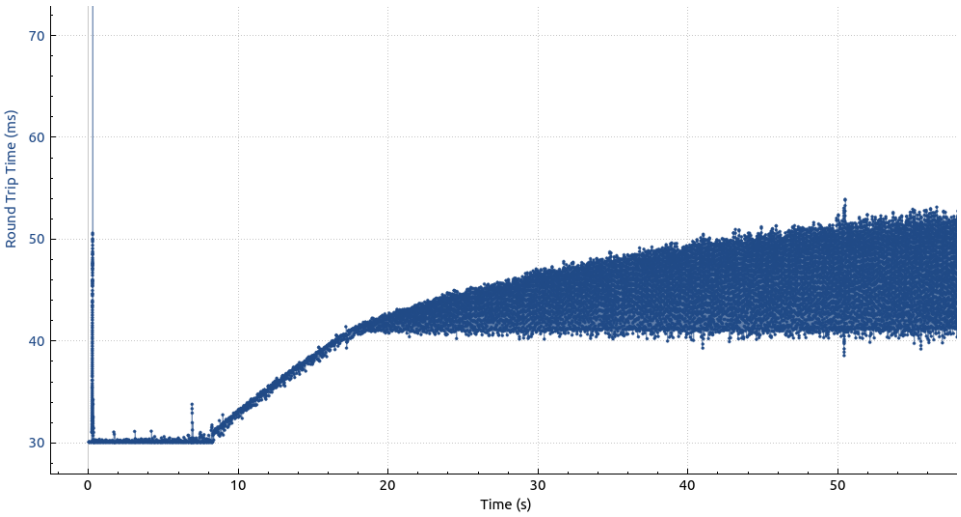


Figure 5.21: Plot of the RTT over time for a TBTCP transfer with 416 Kb sender buffer size. After a certain sending rate is reached, the RTT drops at a regular interval due to gaps in transmission.

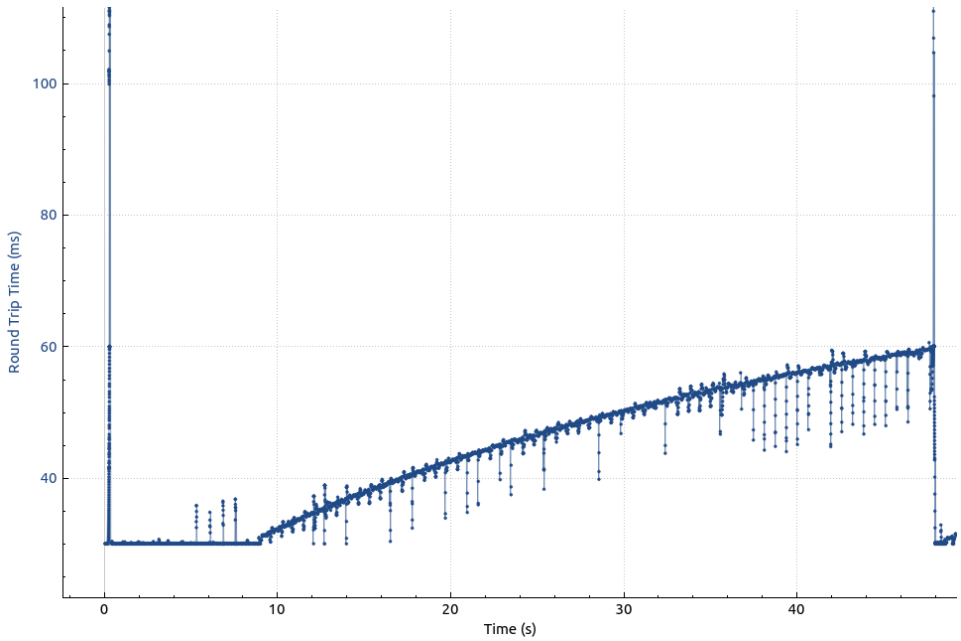


Figure 5.22: Plot of the RTT over time for a TBTCP transfer with increased sender buffer size.

5.5.3 Experiencing loss without reaching link speed

During our throughput testing, we encountered unexpected packet loss before reaching the anticipated link speed limits. We grappled with identifying the root cause until we examined the `netem` configuration in the router. `netem`, a tool for emulating network conditions, has a `limit` parameter which dictates the maximum number of packets its queue can accommodate when adding delay [70]. Once this threshold is reached, we suspected that subsequent incoming packets are dropped. To give an example of the `limit` needed for a high bandwidth transfer of 1 Gbps bottleneck bandwidth and 50 ms delay:

$$
\begin{aligned}
1 \text{ Gbps} &= 1,000,000,000 \text{ bits per second} \\
&= 125,000,000 \text{ bytes per second} \\
\frac{125,000,000 \text{ bytes per second}}{1500 \text{ bytes MTU size}} &= 83,333 \text{ packets per second} \\
83,333 \text{ packets per second} \times 0.05 &= 4,166 \text{ packets every 50 ms}
\end{aligned}
$$

In a mail from Stephem Hemminger [74] he also recommends setting the `limit` parameter to 50% more than the $rate \times delay$, making the final packet limit from the example $4166 \times 1.5 = 6249$.

Our configuration was initially set to a `limit` of 1000 packets. Upon increasing this `limit` in `netem`, we eliminated the losses, allowing for a more comprehensive assessment of our implementation's true throughput capabilities and limitations.

It's important to note that this particular challenge was not directly linked to any intrinsic flaw or limitation within our TBTCP implementation itself. Instead, it underscores the significance of ensuring that testing environments and tools are meticulously configured to prevent unintended obstructions during performance evaluations.

5.5.4 ACK Processing

We were aware that the socket gets locked during both send and ACK processing. Given that our algorithm paces packets individually and doesn't bundle them, the socket remains locked more frequently than in standard TCP transmissions.

While the socket is locked, no other socket operations can be executed. This means that while processing an ACK, we cannot send packets and visa versa. This is a problem, as we want to send packets as frequently as possible to achieve the highest throughput. The more time the socket is locked, the less time we have to send packets.

To measure this we minimized the time spent processing ACKs by enabling

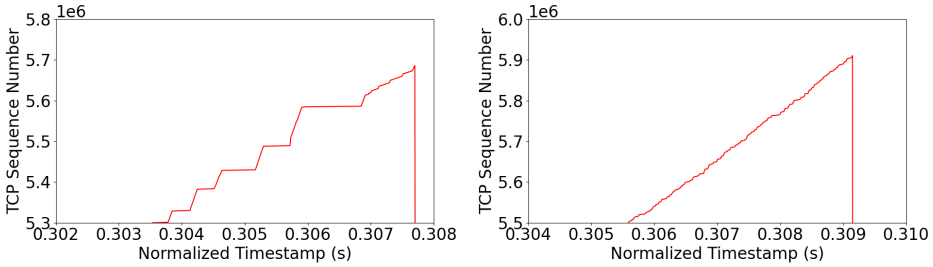


Figure 5.23: 10 Gbps, 30ms. Plot illustrating the end of Slow Start with delayed ACKs enabled (right) and without (left). Configured with an undersized netem limit, clarifying the end of Slow Start.

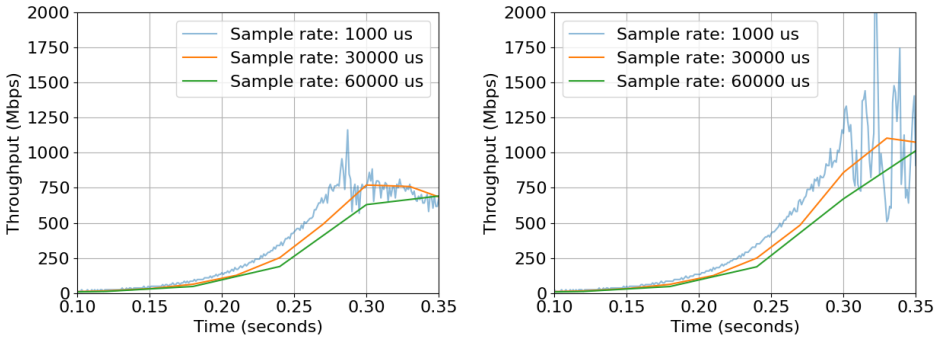


Figure 5.24: 10Gbps link speed, 30ms delay, adequate netem limit, no loss. This plot showcases the peak throughput with (right) and without (left) delayed ACKs activated.

delayed ACKs on the receiver. This is not a responsibility of the TBTCP-host, but rather the receiver, so it is an "impossible" optimization of TBTCP.

As anticipated, enabling delayed ACK (delack) enhances maximum throughput. With fewer ACKs from the receiver, the sender can dispatch more packets before the socket gets locked by ACK processing.

The left graph of Figure 5.23 shows very bursty throughput at the end of Slow Start. The system has to process a considerably higher number of ACKs, leading to more frequent socket locking. However, after enabling delayed ACKs and rerunning the test, the right graph of the same figure shows a smoother performance with minimal bursts.

With fewer socket locks, the maximum throughput should increase. The tests in Figure 5.23 were executed with a netem limit of 1000, resulting in losses during the Slow Start, hence a distinct Slow Start conclusion.

The peak throughput, both with and without delayed ACKs, is depicted in Figure 5.24. The results indicate that the peak stable throughput with delayed ACKs is approximately 33% higher, reaching values close to 1000 Mbps, compared to about 750 Mbps without them.

We have further verified that the processing of ACKs can indeed pose

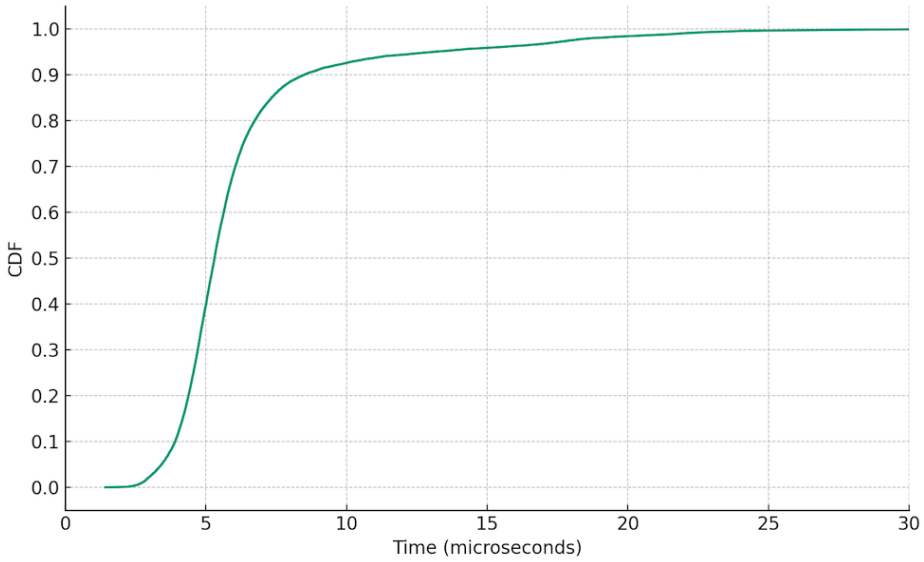


Figure 5.25: CDF of the time it takes to process an ack.

a constraint. When transmitting at 750 Mbps with packets that are 1500 bytes each, TCP needs to dispatch 62500 packets every second. This means the transmission timer is triggered every 16 μs. To draw a comparison with the time it takes for ACK processing, we clocked the function `tcp_rcv_established` on the sender's end, which is responsible for managing incoming packets. From over 25,000 timings, we found the mean duration to be 6 μs and the median to be 5 μs. Looking at Figure 5.25, which presents these timings, roughly 5% of the recorded times surpassed 15 μs in processing. This highlights that there are instances when the transmission timer expires towards the upper limit of the ACK processing duration range. Hence, there are situations where ACK processing hasn't concluded by the time the timer expires, and this occurrence grows more common at increased speeds, like at 1 Gbit/s where the timer would expire every 12 μs.

5.6 Optimizations

5.6.1 Double packet drops

We experienced that the standard calculation of the pacing time (the time between each packet transmission) during Congestion Avoidance was too aggressive.

When developing and testing the algorithm, we saw a pattern of multiple packet drops per recovery event. These double drops did not always occur, but they were a common occurrence nonetheless. Figure 5.26 displays this behavior. In this figure, we see two retransmissions of different packets, indicated by the two decreases on the sequence number y-axis. Keep in

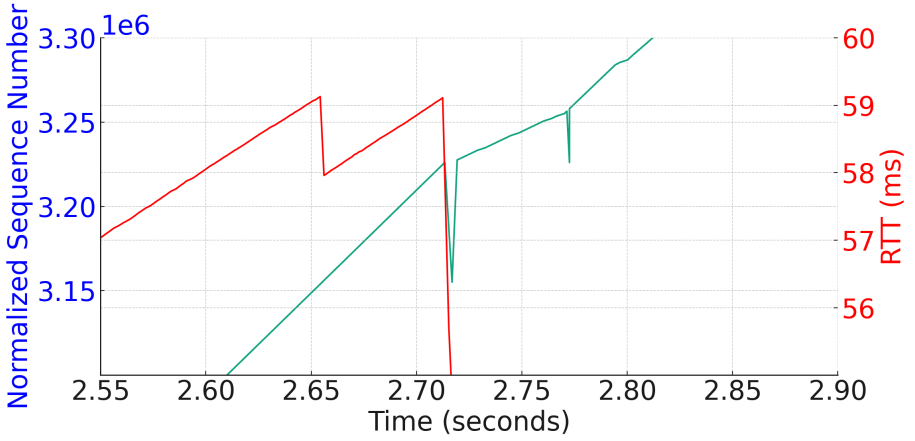


Figure 5.26: Plot showcasing the sequence numbers and RTT over time where a double drop occurs. When looking at this diagram, the drops in the sequence number line indicate a retransmit. Parameters used: a single flow, 30 ms RTT, 10Mbit link and 1xBDP buffer.

mind that the packet drops occur an RTT before the retransmissions. We see the initial packet drop causes the RTT to drop slightly. The RTT then proceeds to increase until another drop occurs about an RTT later. At first, we thought this was related to recovery being too aggressive, however, this was proven not to be the case.

After analyzing the packet traces further, we concluded that the second drop did not occur while TCP was in recovery, but rather in Congestion Avoidance. The packet traces showed that the DupACKs for the first drop arrived directly after the second dropped packet was transmitted, as displayed in Figure 5.27. This tells us that the second dropped packet was transmitted *before* the first loss was discovered by TCP, meaning TCP was in Congestion Avoidance at the time of transmission.

This makes sense, as the TBTCP sending rate grows by quite precisely 1 packet per RTT. We now know that the second packet drop occurs within one RTT from the first drop. We suspect that the first packet drop drained the queue sufficiently for more packets to be received. This explains why the packet drops do not occur subsequently, but rather with a difference of an RTT. As to why these double drops do not always occur, we theorize that small fluctuations introduced by sources like the timer, propagation delay, queue processing, queue alignment, etc. are sufficient to prevent the queue from filling up completely. It is also important to note, that even though two packets were dropped, only a single congestion event occurred.

Mitigating double drops

What the above analysis tells us, is that TBTCP is very proficient at utilizing the queue capacity; we attribute this to TBTCP pacing every single packet.

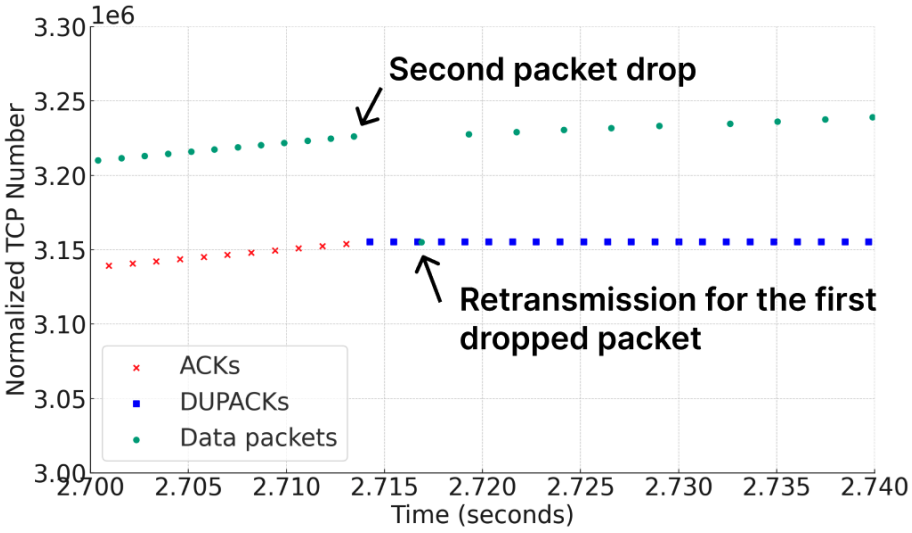


Figure 5.27: Time sequence plot showing data packets, ACKs and DupACKs, to display the timing of the second packet drop. The displayed flow is the same as in Figure 5.26.

The transmission of one more packet before the sender receives notification of a previous packet loss (through DupACKs) is sufficient to overload the receiver, leading to another lost packet. We suspect that these delayed congestion signals are especially prone to occur with a single flow in a controlled environment, as there are not many external factors that affect queue growth.

To mitigate the double drops, we can modify the rate increase function to slow down the rate increase. We assumed that ideally, we would allow the rate to grow a tiny bit slower than one packet per RTT to prevent double drops. To achieve this, we propose a modification to the existing rate increase function.

The current function is given by the following as defined in algorithm 1:

$$(sqrt((8 * (n + k) - 7))/2) - (sqrt((8 * n - 7))/2)$$

Where k represents the many time steps to move ahead, and n is the sequence number Ntx.

To adjust the rate of increase, we can introduce a minor multiplicative factor, for example, 1.000001:

$$(sqrt((8 * (n + k) - 7)) * 1.000001/2) - (sqrt((8 * n - 7))/2)$$

This modification effectively slows down the rate by a small fraction. From our experiments, this seemed to be sufficient to mitigate the double drops.

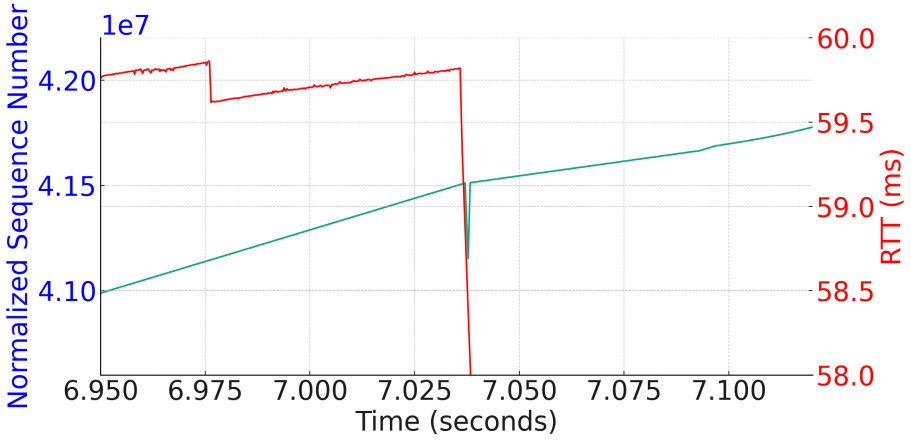


Figure 5.28: Plot showcasing the sequence numbers and RTT over time with a modified rate increase. Parameters used: a single flow, 30 ms RTT, 50Mbit link and 1xBDP buffer.

This is displayed in Figure 5.28, where a single loss event is plotted. We see that in this case, there is also a preliminary drop in the RTT from the packet drop. The RTT then proceeds to increase until a congestion event is detected, rather than dropping a second packet. It is important to note that this factor is somewhat arbitrary, and could be fine-tuned further to optimize better queue utilization while preventing a double drop.

Chapter 6

Conclusion

This chapter summarizes the findings of this thesis, including answering the research questions and giving recommendations for further work.

6.1 Research findings

In this section, we will summarize how we have answered the research questions section 1.3 from the introduction.

Replacing established TCP mechanisms

This was detailed in section 4.3, where we modified the Linux kernel TCP stack. Here, we describe how we disabled the standard sending triggers of TCP. This ensured that all transmissions in Slow Start and Congestion avoidance were initiated from the TBTCP congestion control module. These sending triggers where disabled only if the TBTCP congestion control module is active, thereby maintaining compatibility with other congestion control modules.

Pacing with a High-resolution timer

We accomplished this by expanding the congestion control module API with a new callback, `event_handler`. We associated this callback with a high-resolution timer. When the timer fired, the TBTCP module was notified that it was time to transmit a new packet. After a packet transmission, the timer was requeued with a timestamp calculated from the current sending rate.

The timestamp calculation was achieved by performing floating-point operations and estimations as described in section 4.5.

Giving control to TCP modules

Our implementation enabled initiation of packet transmissions directly from the TBTCP module. This was done by calling `tcp_write_xmit` from the module, instead of relying on the ACK-clock to initiate transmissions from the TCP stack.

Delegate recovery handling to TCP

This was accomplished by notifying the TBTCP module of TCP state transitions through the congestion control module API. Transmissions from the TBTCP module were disabled when TCP was in recovery or loss.

When the TCP state transitions from recovery to Congestion Avoidance, a synchronization step was performed. The synchronization process involved finding the appropriate sending rate based on the amount of in-flight packets at the time where the lost packet was transmitted. This required maintaining a list of in-flight packets along with a record of the in-flight packet count at the time each packet was transmitted.

Performance

When evaluating the implementation we found that the per-packet pacing resulted in a very consistent pacing curve. When comparing TBTCP pacing with other pacing schemes as in Figure 5.4, TBTCP appears a lot smoother. TBTCP also paces the initial window of packets, which the other pacing schemes did not.

A consequence of this fine-grained per-packet pacing makes TBTCP proficient at utilizing the available queue capacity. This was demonstrated in subsection 5.6.1, where a rate increase of precisely one packet per RTT was enough to make double packet-drops a common occurrence.

We were able to achieve fairly stable throughput values up to 750 Mbps without delayed ACK enabled, and up to 1000 Mpbs with delayed ACK enabled. This is further described in subsection 5.5.4.

6.2 Further work

6.2.1 Evaluation in a heterogenous flow environment

The evaluation of our implementation was limited in terms of comparing performance and behavior in an environment with other TCP flows. Instead, our evaluation focused on verifying the capabilities that was expected of our algorithm according to the TBTCP specification in chapter 3, in addition to throughput metrics. As such, the tests were performed in an isolated environment with a single flow.

A continuation of the evaluation would be to assess our implementation in a heterogenous flow environment.

6.2.2 Implementing loss recovery

As our implementation only concerned itself with the Slow Start and Congestion Avoidance phases of TCP, the next step would be to implement loss recovery. The TBTCP research paper (to be published at the time of writing) describes this algorithm.

6.2.3 Pace from hardware

Our implementation utilizes high-resolution software timers within the Linux kernel to pace packets. As we reached higher speeds, we observed undefined and bursty behavior. We believe this is mainly due to overhead introduced by the timer and ACK-processing detailed in section 5.5.

A suggestion to improve on this is to perform the pacing from hardware, instead of using software timers. This would ideally require an implementation where congestion control modules can control the rate of pacing by passing per-packet transmission timestamps to the hardware. Such an implementation could be deployed in a hardware Network Interface Card (NIC). Work related to performing pacing from a NIC can be found in [63] and [75].

6.2.4 Minimizing the impact of pacing overhead

In subsection 5.5.4, we detailed the maximum throughput achieved. We suspected the limiting factor was related to the overhead from processing ACKs in addition to the constraint of only transmitting a single packet per timer expiry.

To optimize this, we would have to either spend less time sending or minimize time spent processing ACKs. The only optimizations that can be done by TBTCP is to optimize sending. To achieve this, a suggestion would be to fire fewer timer events, such that multiple packets can be sent per expiration event. To preserve the per-packet pacing on lower sending rates, this feature could be enabled only after the sending rate reaches a certain threshold.

6.3 Recommendations

For those who seek to implement their own TCP module in a custom Linux kernel, or continue our work, we have the following recommendations to give a head start.

Use bare metal

When testing our custom Linux kernel, we initially used a virtual machine. This worked for initial testing, however, to test the actual performance of an implementation, we recommend using bare metal. In our experience, the performance and behavior observed when using virtual machines are not representable of testing on bare metal.

Automate the testing setup

We automated the testing process through a collection of scripts. These scripts did everything from pushing source code to the test machine, compiling, installing, rebooting and running the transmissions. We recommend creating such a pipeline for kernel deployment and testing, as it significantly reduces the manual labor needed during development.

Disable offloading mechanisms

It is in a developer's best interest to minimize external factors that may affect the results of tests. We recommend disabling the various hardware offloading mechanisms on the test machine. One should also be aware of TCP-specific mechanisms like the use of ECNs, SACKs and delayed ACKs. If the implementation has high CPU demands, enabling delayed ACKs should be considered to limit the ACK processing overhead.

Tune the TCP socket

Be aware of application overhead when running high-bandwidth tests. When using `iperf` in combination with a small socket buffer, we experienced large gaps in the transmission pattern. We theorize this was due to the application not being able to keep up with the transmission rate. We mitigated this issue partially by increasing the socket buffer sizes. In general, the dedicated socket memory should be tuned to match the bandwidth-delay product demands.

Netem on higher speeds

When testing our implementation at higher speeds, we experienced unexpected packet loss. When inspecting further, we found that the packet loss occurred in the `netem` qdisc. To prevent this, we caution to remember to increase the `netem limit` so that it corresponds to the buffering requirements as described in subsection 5.5.3.

6.4 Closing remarks

Benefits	Disadvantages
High queue utilization	Overhead from per-packet pacing
Only requires implementation on the sender-side	
Pluggable increase function	Limited throughput capabilities
Minimized burstiness	
Pacing the Initial Window	

Table 6.1: Highlighted benefits and disadvantages of the TBTCP implementation.

Implementing a timer-based TCP congestion control can provide several benefits as displayed in Table 6.1 . Primarily, the inherent per-packet pacing yields a notably smoother traffic pattern compared to other TCP pacing variants. It can be adopted with ease, as it would require implementation only on the client-side. Furthermore, the increase function is a simple replaceable equation making it versatile in the case where a different increase behavior is desired.

Among the drawbacks observed were limited throughput capabilities caused by the overhead from ACK processing and per-packet pacing. We achieved throughput values close to 1000 Mbps with only minor fluctuations. Past this point, we noticed bursty and undefined behavior.

We have outlined some possible solutions to these problems in the previous sections, and remain excited to see how this algorithm will perform if these issues are addressed.

In summary, we have implemented and evaluated a timer-based TCP congestion control module in the Linux kernel. We evaluated its capabilities and limitations both in regards to the TBTCP specification and throughput metrics. Through our research, we have shown that the implementation was feasible and provided benefits, however, there are still discoveries to be made regarding how such an implementation would perform in a real-world scenario.

Appendix A

Source code

The modified Linux kernel can be retrieved from
https://github.com/andreaslimidev/tbtcp-linux.
The most significant changes are located in `net/ipv4` directory, most notably the files `tcp_tb.c`, `tcp_output.c` and `tcp_input.c`.

Code references

[16] `linux-kernel-5.19 | TCP_INIT_CWND`. commit. Accessed on April 8, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/include/net/tcp.h#L231.

[24] `linux-kernel-5.19 | tcp_cwnd_reduction`. commit. Accessed on March 2, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/net/ipv4/tcp_input.c#L2614.

[30] `linux-kernel-5.19 | cubic hystart_update`. commit. Accessed on October 16, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/net/ipv4/tcp_cubic.c#L386C55-L387C1.

[32] `linux-kernel-5.19 | tcp_states.h`. commit. Accessed on April 25, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/include/net/tcp_states.h#L12.

[33] `linux-kernel-5.19 | CA states`. commit. Accessed on April 25, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/net/ipv4/tcp_input.c#L2224.

[35] `linux-kernel-5.19 | __tcp_transmit_skb`. commit. Accessed on November 1, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/net/ipv4/tcp_output.c#L1406C3-L1406C3.

[36] `linux-kernel-5.19 | tcp.h:struct tcp_congestion_ops`. commit. Accessed on March 1, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/include/net/tcp.h#L1053.

[37] `linux-kernel-5.19 | inet_connection_sock`. commit. Accessed on March 2, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/include/net/inet_connection_sock.h#L135.

[38] linux-kernel-5.19 | tcp_enter_loss. commit. Accessed on March 2, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/net/ipv4/tcp_input.c#L2154.

[39] linux-kernel-5.19 | tcp_set_ca_state. commit. Accessed on March 2, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/net/ipv4/tcp_cong.c#L44.

[40] linux-kernel-5.19 | tcp_cong_control. commit. Accessed on March 2, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/net/ipv4/tcp_input.c#L3902.

[41] linux-kernel-5.19 | tcp_try_undo_loss. commit. Accessed on March 2, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/net/ipv4/tcp_input.c#L2574.

[42] linux-kernel-5.19 | tcp_try_undo_recovery. commit. Accessed on March 2, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/net/ipv4/tcp_input.c#L2529.

[43] linux-kernel-5.19 | tcp_reno_undo_cwnd. commit. Accessed on March 2, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/net/ipv4/tcp_cong.c#L473.

[44] linux-kernel-5.19 | tcp_input.c:update_pacing_rate. commit. Accessed on February 24, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/net/ipv4/tcp_input.c#L897.

[45] linux-kernel-5.19 | sch_fq.c. commit. Accessed on March 1, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/net/sched/sch_fq.c#L1.

[46] linux-kernel-5.19 | sk_pacing_rate. commit. Accessed on March 2, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/include/net/sock.h#L460.

[47] linux-kernel-5.19 | bbr_init. commit. Accessed on April 25, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/net/ipv4/tcp_bbr.c#L1077.

[48] linux-kernel-5.19 | tcp_pacing_check. commit. Accessed on March 2, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/net/ipv4/tcp_output.c#L2467.

[49] `linux-kernel-5.19 | tcp_update_skb_after_send`. commit. Accessed on March 2, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/net/ipv4/tcp_output.c#L1198.

[50] `linux-kernel-5.19 | tcp_pace_kick`. commit. Accessed on March 2, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/net/ipv4/tcp_output.c#L1187.

[52] `linux-kernel-5.19 | tcp_small_queue_check`. commit. Accessed on March 2, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/net/ipv4/tcp_output.c#L2680.

[53] `linux-kernel-5.19 | sch_fq.c`. commit. Accessed on June 1, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/net/sched/sch_fq.c#L135.

[56] `linux-kernel-5.19 | enum hrtimer_mode`. commit. Accessed on March 10, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/include/linux/hrtimer.h#L27.

[58] `linux-kernel-5.19 | tcp_tso_autosize`. commit. Accessed on March 2, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/net/ipv4/tcp_output.c#L1968.

[59] `linux-kernel-5.19 | tcp_sendmsg_locked`. commit. Accessed on March 2, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/net/ipv4/tcp.c#L1192.

[60] `linux-kernel-5.19 | tcp_data_send_check`. commit. Accessed on March 2, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/net/ipv4/tcp_input.c#L5486.

[61] `linux-kernel-5.19 | tcp_write_xmit`. commit. Accessed on March 2, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/net/ipv4/tcp_output.c#L2599.

[62] `linux-kernel-5.19 | __tcp_transmit_skb`. commit. Accessed on March 2, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/net/ipv4/tcp_output.c#L1237.

[65] `linux-kernel-5.19 | x86 fpu api.h`. commit. Accessed on March 8, 2023. URL: https://github.com/torvalds/linux/blob/3d7cb6b04c3f3115719235cc6866b10326de34cd/arch/x86/include/asm/fpu/api.h#L18.

Article references

[1] David Wei et al. 'TCP pacing revisited'. In: *Proceedings of IEEE INFOCOM*. Vol. 2. Citeseer. 2006, p. 3.

[2] Elie F. Kfoury et al. 'Enabling TCP Pacing using Programmable Data Plane Switches'. In: *2019 42nd International Conference on Telecommunications and Signal Processing (TSP)*. 2019, pp. 273–277. DOI: 10.1109/TSP.2019.8768888.

[3] Eric Dumazet. *tcp: internal implementation for pacing*. commit. Accessed on February 24, 2023. May 2017. URL: https://github.com/torvalds/linux/commit/218af599fa635b107cfe10acf3249c4dfe5e4123.

[4] Jim Gettys. 'Bufferbloat: Dark Buffers in the Internet'. In: *IEEE Internet Computing* 15.3 (2011), pp. 96–96. DOI: 10.1109/MIC.2011.56.

[5] Wesley Eddy. *Transmission Control Protocol (TCP)*. RFC 9293. Aug. 2022. DOI: 10.17487/RFC9293. URL: https://www.rfc-editor.org/info/rfc9293.

[6] Ethan Blanton, Dr. Vern Paxson and Mark Allman. *TCP Congestion Control*. RFC 5681. Sept. 2009. DOI: 10.17487/RFC5681. URL: https://www.rfc-editor.org/info/rfc5681.

[7] Wikipedia contributors. *Berkeley sockets — Wikipedia, The Free Encyclopedia*. [Online; accessed 7-April-2023]. 2023. URL: https://en.wikipedia.org/w/index.php?title=Berkeley_sockets&oldid=1143831642.

[8] *socket*. Section 2 of the Linux man-pages project. Linux man-pages project. 2019. URL: https://man7.org/linux/man-pages/man2/socket.2.html.

[9] J. Nagle. *RFC 896 - Congestion Control in IP/TCP Internetworks*. Request for Comments. Status: INFORMATIONAL. Jan. 1984. URL: https://datatracker.ietf.org/doc/html/rfc896.

[10] David Borman. *TCP Options and Maximum Segment Size (MSS)*. RFC 6691. July 2012. DOI: 10.17487/RFC6691. URL: https://www.rfc-editor.org/info/rfc6691.

[11] Dr. Steve E. Deering and Jeffrey Mogul. *Path MTU discovery*. RFC 1191. Nov. 1990. DOI: 10.17487/RFC1191. URL: https://www.rfc-editor.org/info/rfc1191.

[12] J. Postel. *RFC 792 - Internet Control Message Protocol*. https://datatracker.ietf.org/doc/html/rfc792. Sept. 1981.

[13] M. Mathis and J. Heffner. *RFC 4821 - Packetization Layer Path MTU Discovery*. https://datatracker.ietf.org/doc/html/rfc4821. Standards Track. Mar. 2007.

[14] R. Braden. *RFC 1122 - Requirements for Internet Hosts - Communication Layers*. https://datatracker.ietf.org/doc/html/rfc1122. Oct. 1989.

[15] Matt Sargent et al. *Computing TCP's Retransmission Timer*. RFC 6298. June 2011. DOI: 10.17487/RFC6298. URL: https://www.rfc-editor.org/info/rfc6298.

[17] H.K. Jerry Chu et al. *RFC6928 - Increasing TCP's Initial Window*. 2013. URL: http://www.rfc-editor.org/rfc/rfc6928.txt.

[18] Sally Floyd et al. *TCP Selective Acknowledgment Options*. RFC 2018. Oct. 1996. DOI: 10.17487/RFC2018. URL: https://www.rfc-editor.org/info/rfc2018.

[19] Ethan Blanton et al. *A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP*. RFC 6675. Aug. 2012. DOI: 10.17487/RFC6675. URL: https://www.rfc-editor.org/info/rfc6675.

[20] Matt Mathis, Nandita Dukkipati and Yuchung Cheng. *Proportional Rate Reduction for TCP*. RFC 6937. May 2013. DOI: 10.17487/RFC6937. URL: https://www.rfc-editor.org/info/rfc6937.

[21] Y. Cheng M. Kukojo and M. Mathis. *Use FlightSize instead of cwnd*. https://github.com/NTAP/rfc8312bis/issues/114. 2021.

[22] Injong Rhee et al. *CUBIC for Fast Long-Distance Networks*. RFC 8312. Feb. 2018. DOI: 10.17487/RFC8312. URL: https://www.rfc-editor.org/info/rfc8312.

[23] David Borman et al. *TCP Extensions for High Performance*. RFC 7323. Sept. 2014. DOI: 10.17487/RFC7323. URL: https://www.rfc-editor.org/info/rfc7323.

[25] Van Jacobson. 'Congestion avoidance and control'. In: *ACM SIGCOMM computer communication review* 18.4 (1988), pp. 314–329.

[26] Yuchung Cheng et al. *The RACK-TLP Loss Detection Algorithm for TCP*. RFC 8985. Feb. 2021. DOI: 10.17487/RFC8985. URL: https://www.rfc-editor.org/info/rfc8985.

[27] Dominik Scholz et al. 'Towards a deeper understanding of TCP BBR congestion control'. In: *2018 IFIP networking conference (IFIP networking) and workshops*. IEEE. 2018, pp. 1–9.

[28] Neal Cardwell et al. *BBR Congestion Control*. Internet-Draft draft-cardwell-iccrg-bbr-congestion-control-02. Work in Progress. Internet Engineering Task Force, Mar. 2022. 66 pp. URL: https://datatracker.ietf.org/doc/draft-cardwell-iccrg-bbr-congestion-control/02/.

[29] Sangtae Ha, Injong Rhee and Lisong Xu. 'CUBIC: a new TCP-friendly high-speed TCP variant'. In: *ACM SIGOPS operating systems review* 42.5 (2008), pp. 64–74.

[31] Amit Aggarwal, Stefan Savage and Thomas Anderson. 'Understanding the performance of TCP pacing'. In: *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No. 00CH37064)*. Vol. 3. IEEE. 2000, pp. 1157–1165.

[34] Sally Floyd, Dr. K. K. Ramakrishnan and David L. Black. *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC 3168. Sept. 2001. DOI: 10.17487/RFC3168. URL: https://www.rfc-editor.org/info/rfc3168.

[51] Carlo Augusto Grazia, Martin Klapez and Maurizio Casoni. 'The New TCP Modules on the Block: A Performance Evaluation of TCP Pacing and TCP Small Queues'. In: *IEEE Access* 9 (2021), pp. 129329–129336. DOI: 10.1109/ACCESS.2021.3113891.

[54] Neal Cardwell. *Re: How to comfirm BBRv1 works with HTB+FQ?* Google Groups: BBR Development. Available at: https://groups.google.com/g/bbr-dev/c/ADbl1zcpP0w/m/dIZGuuycBgAJ. 2021.

[55] Thomas Gleixner and Ingo Molnar. *hrtimers - subsystem for high-resolution kernel timers*. Accessed: 2023-03-10. URL: https://docs.kernel.org/timers/hrtimers.html.

[57] Yuchung Cheng and Neal Cardwell. 'Making linux TCP fast'. In: *Netdev conference*. 2016.

[63] Salvatore Pontarelli, Giuseppe Bianchi and Michael Welzl. 'A Programmable Hardware Calendar for High Resolution Pacing'. In: *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. 2018, pp. 1–6. DOI: 10.1109/HPSR.2018.8850731.

[64] Linus Torvalds. *Mail from Linus speaking on FPU enabling*. mail. Accessed on March 2, 2023. URL: https://lore.kernel.org/all/Pine.LNX.4.44.0303101203330.2722-100000@home.transmeta.com/.

[66] Romerio. *fastapprox: fast approximations of mathematical functions*. https://github.com/romeric/fastapprox. 2017. URL: https://github.com/romeric/fastapprox.

[67] Wikipedia contributors. *Methods of computing square roots*. https://en.wikipedia.org/wiki/Methods_of_computing_square_roots. Accessed: October 5, 2023. Wikipedia, The Free Encyclopedia, 2023. URL: https://en.wikipedia.org/wiki/Methods_of_computing_square_roots.

[68] Oracle. *Documentation for Solaris*. Accessed: 2023-10-06. 2023. URL: https://docs.oracle.com/cd/E18752_01/html/817-5477/epmpv.html#:~:text=SSE2%20instructions%20are%20an%20extension,precision%20floating%2Dpoint%20conversion%20instructions.

[69] *iperf*. Section 1 of the iperf Linux man page. die.net. 2023. URL: https://linux.die.net/man/1/iperf.

[70] *tc-netem(8) - Linux man page*. Accessed: 2023-11-07. 2023. URL: https://man7.org/linux/man-pages/man8/tc-netem.8.html.

[71] *tcpdump*. Section 8 of the tcpdump Linux man page. die.net. 2023. URL: https://linux.die.net/man/8/tcpdump.

[72] Sebastian Zander and Grenville Armitage. 'Minimally-intrusive frequent round trip time measurements using Synthetic Packet-Pairs'. In: *38th Annual IEEE Conference on Local Computer Networks*. 2013, pp. 264–267. DOI: 10.1109/LCN.2013.6761245.

[73] Kernel.org. *Documentation/sysctl/net.txt*. 2023. URL: https://www.kernel.org/doc/Documentation/sysctl/net.txt.

[74] Stephen Hemminger. *Re: [Netem] TBF burst size*. https://lists.linuxfoundation.org/pipermail/netem/2007-March/001094.html. Accessed: 2023-11-07. Mar. 2007.

[75] Ahmed Saeed et al. 'Carousel: Scalable Traffic Shaping at End-Hosts'. In: *ACM SIGCOMM 2017*. 2017.