Extending TCP for Low Round Trip Delay

Logarithmically Scaled Additive Increase Multiplicative Decrease (LS-AIMD)

Asad Sajjad Ahmed

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

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2019

Extending TCP for Low Round Trip Delay

Logarithmically Scaled Additive Increase Multiplicative Decrease (LS-AIMD)

Asad Sajjad Ahmed

Extending TCP for Low Round Trip Delay

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

Printed: Reprosentralen, University of Oslo

Abstract

Transmission Control Protocol (TCP) is a widespread protocol and has evolved since the very beginning of the Internet. TCP implements congestion control to hinder congestion collapse and to shareout the capacity evenly across the participating flows.

Recent studies have shown that the acknowledgement clock TCP uses to control its transmission rate has a remarkable weakness which makes TCP perform not so well under shallow Round Trip Time (RTT). The current deployment of DataCenreTCP (DCTCP) in data centres and Low Latency Low Loss Scalable throughput (L4S) over broadband has helped in cutting the generally deep queue of the Internet. However, Linux TCP has a weakness which causes it to override the AQM whenever the base RTT is also shallow because of its minimum transmission rate of two segments per RTT. Shallow base RTTs is the typical environment within data centres so by overriding the AQM TCP makes the job of the AQM undoubtedly harder. TCP by going too fast brings back the once removed queue and diminishes any benefits of the low queue AQM.

We propose Logarithmically Scaled Additive Increase Multiplicative Decrease (LS-AIMD) in place for AIMD to scale over shallow base RTTs. We evaluate LS-AIMD against AIMD under both Reno and DCTCP. Our initial screening shows that LS-AIMD performs under shallow RTT more than ten times better than AIMD. The gain of LS-AIMD continues to grow as base RTT continues to descend to even lower values.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisor Bob Briscoe for his trenmendous help. I would also like to thank familiy. I would also like appreciate the help I got from serval friends over at the UiO and Simula (Fornebu).

Part I

Introduction

Chapter 1

Introduction

We start by stating the motivation behind this thesis. Next, we describe the exact problem we will investigate. We then give out various limitations to scope down the length of our research. A summary follows of our main contributions toward better understanding and solving the problem in hand. We state our usage of typically conventional research techniques found in computer science. These guidelines set the structure of our thesis. Finally, we give out an outline of how the rest of our work will look alike.

1.1 Motivation

Low latency has become a never-ending quality of service requirement for today's applications. While the bitrate of the Internet has continued on its rapid growth worldwide, do we not see the same trend in the direction of latency. The enormous latency experienced on the Internet impacts the lives of almost everyone worldwide and is a cost associated with just being online. The low latency requirement does not equate to low capacity eating traffic, the need for low latency has become as an essential requirement for all greedy traffic run over the state of the art Transmission Control Protocol (TCP).

The development of modern applications has led to the need for a very low end-to-end delay, where the more shallow the delay is, the better and more attractive is the service to the end-user. The gaming and VoIP industry has long been one of those applications, but in today's world has the need shifted and the need for low latency has become as important, if not more, as a high bitrate for literally all application. A modern application which fulfils such need could be web, instant messaging, virtual/augmented reality, cloud gaming, video conferencing/streaming, emergency systems, remote assistance, drones, and many, many more. The need is, of course, just the tip of the iceberg, surely the need will continue to grow also in the future, and the benefits of lower latency remain ignored until they start to be part of one's everyday life.

The Round Trip Time (RTT) experienced by the application consists of mainly irreducible base RTT, which consists of large chunks such as transmission and propagation delays. Another part of the RTT exists of

parts which have the opportunity to be massively reduced. One such part is the precedence of an outstanding queuing delay within the routers and switches in the path between the end-hosts. The queueing delay usually stands for the most substantial reducible chunk of the RTT and has been a problem since the very beginning of the Internet.

Active Queue Management (AQM) schemes were introduced to take care of the enormous queueing delay problem and is a research field of its own. The bottleneck is made "smart" to solve the problem, so it either evict or mark packets which stay queued for more extended periods. The queue of the bottleneck then stays mostly short, and the remaining buffer absorbs the bursts from synchronised traffic. The AQM is then in the full control of the queue and may select a threshold just high enough to maintain full utilisation of the link. However, the co-existence of TCP traffic and an AQM optimised to keep a shallow queue do not mix well together. The problem that this thesis is investigating is a not so unusual problem where TCP traffic starts to override a shallow queue AQM in a network where the base RTT is already low (more explained in the next section).

DataCentreTCP (DCTCP) tries to overcome some of the flaws of TCP and scales well in-network with shallow queue AQMs because of its less aggressive reduction of the transmission rate. Data centre often use DCTCP because of this, but it comes with the same problem as TCP when it comes to scalability for shallow base RTT. For DCTCP, the problem is a lot worse since the processing nodes within a data centre are tightly packed together to achieve a shorter base RTT. TCP Prague aims to be the next generation of congestion control algorithms that follow a set of requirements given out by the Internet Engineering Task Force (IETF). One such requirement requires the sender to stay responsive in network path with very low end-to-end delay.

Low **L**atency **L**ow **L**oss **S**calable throughput (**L4S**) is an ongoing project to deliver low latency over the Internet. L4S is about making scalable congestion control algorithms, such as DCTCP and TCP Prague safely available to **all** applications over the Internet. The deployment of L4S gives a golden opportunity to scale for such networks because of the isolation delivered by this architecture in terms of an independent queue for new traffic. The problem that this thesis undertakes is, therefore, vital to solve in the earliest stage of deployment as possible, or else will the window to scale for lower base RTT be a problem which continues to be hard to solve also in the future.

The problem happens on Internet links where a low transmission rate per RTT is required. Note that a low transmission rate does not necessarily mean a low bitrate, but could instead mean a low base RTT. The likelihood for the problem to happen on the Internet is, surprisingly, assumed to be more common in a modern network topology. The problem is currently non-existent on the Internet due to a proportionally massive queue at the bottleneck compared to the bitrate delivered. However, this trend is about to end with the deployment of shallow queue AQMs.

The same problem exists widely in current data centres due to the more modern network infrastructure. Multiple groups are investigating

the problem and proposing their solution [19, 37, 38, 47], but the proposed solutions remain targeted to fully controlled environments.

1.2 Problem

A capacity seeking sender tries to consume all the available capacity in its path to the receiver. The sender continues to probe for more capacity until the first sign of congestion. The bottleneck is forced to drop the newly arrived packet as the buffer has run out. A new packet will first start to enqueue again when the head of the queue has made space by dequeuing a packet. Packet loss is an indication of congestion taking place in the network. The queue got in this condition as a result of the greedy behaviour of the sender. The sender must, therefore, reduces its transmission rate when eventually the loss event is detected to prevent buffer overflow. The queue will then have time to drain and reduce the possibility of successive losses at the bottleneck. Each sender typically does this through a congestion window made out of segments assembled from the application data. The congestion window is reduced, typically to half of its original size, upon the detection of packet loss. Congestion is not an uncommon phenomenon in computer networks. Instead, it is the usual outcome of the capacity-seeking process. Meaning, congestion does not necessarily imply excessive traffic neither or insufficient capacity at the bottleneck. The greedy sender will always be able to cause congestion no matter how high bitrate the network can deliver and neither is a deeper buffer of any help.

The sender uses acknowledgements to get reception of earlier transmitted segments and usually except one to arrive after one RTT. The acknowledgement usually tells the sender that some of the earlier transmitted segments have now most likely left the network, and the earlier consumed bitrate is now free to reuse. The sender can thus keep a constant "clock" of segments in flight per RTT by clocking out as many new segments as just got delivered to the receiver. The sender does not only keep constant of segments inflight per RTT but tries to probe for more capacity, and do so by updating its congestion window through additive increase schemes. The additive increase has linear growth of the congestion window per RTT. Meaning, the congestion window will continue to grow until the next epoch of congestion. The sender will then releases half of its transmission rate and try to regain the previous rate by increasing its congestion window by one segment at a time per RTT. The goal of using an Additive Increase Multiplicative Decrease (AIMD) scheme is to converge a set of senders to their appropriate share of the available capacity. However, the capacity seeking behaviour of the sender has kept the queue of the bottleneck mostly filled. Packets received at the bottleneck will, therefore, be delayed since they must wait through the enormous queue. The queuing delay further impacts the RTT, and the effective RTT between the sender and the receiver appear to be longer than it is. The induced queueing delay is, of course, the fault of the bottleneck who let the queue build-up and

silently absorbs the capacity-seeking traffic.

AQM schemes were introduced to battle the uprising problem with high latency caused by the capacity-seeking traffic. An AQM makes a proactive choice and evicts packets before the queue gets out of control to induce a fake signal of congestion to the sender. The AQM fools the sender and reinitiates the probing phase of the sender earlier. The AQM is now in full charge of the queue by penalising traffic, which tries to build an enormous queue. The bottleneck is now keeping a short queue, and therefore deliver a lower base RTT to its traffic. The benefit of having a bottleneck with a long buffer while maintaining a short queue through AQM schemes is to absorb the burst of traffic. While by just having a short buffer equates to a loss of utilisation whenever multiple senders arrive synchronised at the bottleneck [33]. However, an AQM which utilises the use of drop as a signal of congestion is cumbersome and wastes the limited resources of the network [24]. An AQM is, therefore, often set up to mark congestion through the use of Explicit Congestion Notification (ECN). AQM modifies particular bits of the IP header of the packet for packets which stays enqueued for an extended period. A receiver then echoes the congestion signal back to the sender. However, this only works if both hosts were willing to use ECN. An ECN-capable sender typically sees ECN at the same level of congestion as packet loss and back off as if the congestion were due to a tail loss episode. The real benefits of an AQM will only be possible if all traffic follows the recommendation, packet loss due to congestion will now be non-existent, and the buffer of the AQM will now be freed to absorb a possible burst. However, the AQM will be forced to drop packets of traffic which does not support ECN since this is the last option left to slow them down.

The problem that this thesis tries to address is the sender refusal to go slower than two segments, or 2*SMSS bytes, per RTT. The latest congestion control TCP standard[4] (Section 3.1) does not require the sender to go slower than this threshold. However, the standard mandates the sender to reprobe from a single segment on Retransmission Timeout (RTO) (see Section A.3.1). The sender also prefers to stay above two segments per RTT to interwork with the delayed acknowledgement mechanism found at the receiver (see Section A.1.2). The sender will, therefore, at this point, become unresponsive, and continue to send at least two segments per RTT. The AQM will emit even more signals to the sender to slow down, but the sender will merely ignore these signals. The sender now contributes to a long queue at the bottleneck than chosen by the AQM. The extra queuing delay added by the sender now propagates to all other traffic at the bottleneck. The AQM struggle to keep the RTT low for all traffic at the bottleneck just because TCP traffic refuses to cooperate. The culprit is TCP, and a better configuration of the AQM cannot solve this problem.

The problem happens in networks with low Bandwidth Delay Product (BDP), which can be due to a low bitrate or low RTT. Our primary motivation is a network path with low BDP due to a very high bitrate and shallow RTT.

The BDP could also be low due to other factors such as a heavily

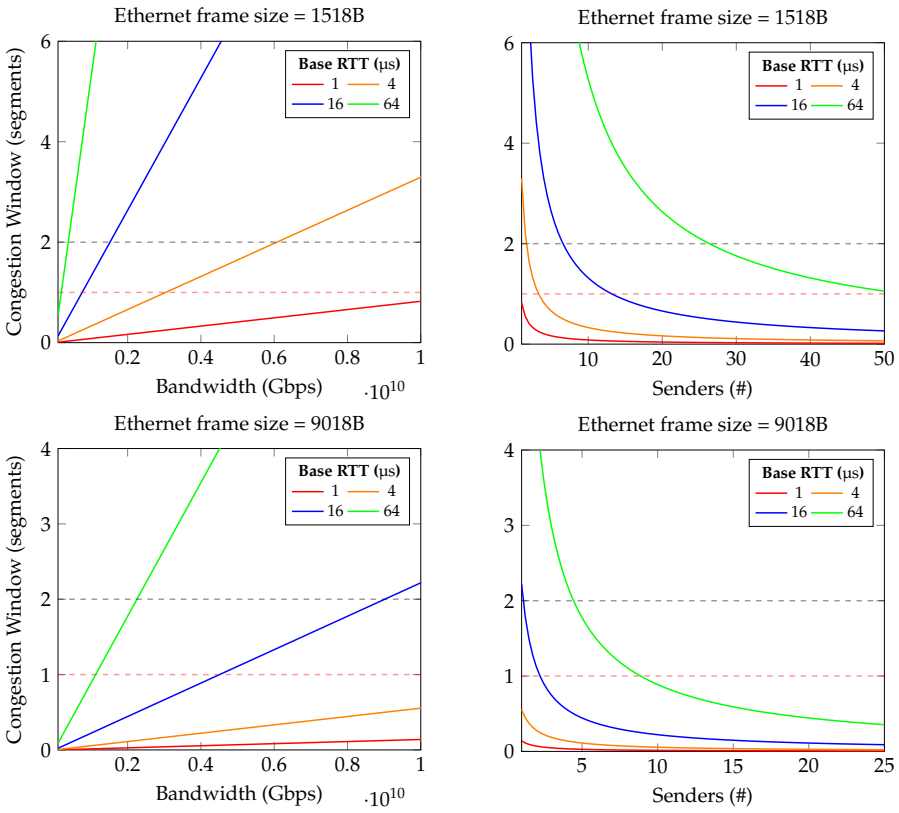


Figure 1.1: Plot of congestion window sizes for 10Gbps

congested path. A lower BDP equates to a lower congestion window (see Figure 1.1). A network path with very high bitrate does not avoid the small BDP problem. A network with responsive TCP traffic contra the conventional TCP traffic opens up for a lower base RTT.

Data Centre TCP (DCTCP) is a congestion control algorithm often used in data centres where processing nodes have a very high bitrate and low latency in the network. DCTCP aims to keep a very low queuing delay and do so by using moving average to make less drastically reduction of the congestion window. However, DCTCP struggles as the AQM tries to enforce a shallow RTT. DCTCP has the same problem as the conventional TCP to transmit at a minimum of two segments per RTT. Therefore, solving this flaw of TCP benefits data centres traffic. The RTT of the network becomes more stable and shallow. Short-lived flows in data centres will, therefore, not need to wait through an enormous queue caused by capacity eating flows.

The traditional way to clock out packet conflicts with the problem since the current way to clock out packet relies on the delayed acknowledgement mechanism found at the receiver. The sender will thus be forced to wait for a reception of earlier sent data to emit more bytes. The sender when in low BDP network is neither able to send two whole segments within an RTT to trigger the acknowledgement. Taking the commonly used delayed acknowledgement into consideration cause problems since the receiver will

not return the acknowledgement before 40-500ms has passed. The timeout of the delayed acknowledgement mechanism is a substantial penalty for the sender when the RTT becomes low. For instance, an RTT of 5 ms equals a penalty as significant as 8-100 RTTs.

1.2.1 Problem statement

We list problems in TCP which conflict with our objective to achieve shallow RTT. The order of this list decides the priority of the problem (the very first entry of this list has the highest priority). We will try to address as many of the problem given here.

- **Can we extend TCP for shallow base RTT?**

  The goal of this thesis is to research if the current specification of TCP can be extended to allow a low queuing delay in computer networks. The main problem this thesis tries to solve is the TCP ability to work with shallow RTT. A Low RTT can only be possible if a bottleneck on average has a small queue.

- **Can we preserve stretch acknowledgements for low congestion window values?**

  It is tempting to disable the delayed acknowledgement mechanism to mitigate the problem. Although, we can make the network more efficient if we can keep this mechanism untouched. Another goal is, therefore, to preserve the delayed acknowledgement mechanism. However, one must weight this up against drawbacks it sets on the solution.

- **Can we preserve a congestion window made out of segments in a network with a low transmission rate per RTT?**

  TCP standard let the implementer use a congestion window based on two different approaches, namely the byte-based approach and segment-based approach. The congestion window must go below SMSS bytes due to the requirement of the low BDP problem, but this is difficult for the segment-based approach since it cannot use a congestion window between 0 and SMSS bytes. Therefore, another goal is to design a solution which does not conflict with this requirement.

- **Are we able to create a minimal solution which is easy to implement?**

  A secondary goal is to solve the problem with minimal modification done to the already existing TCP. A generic solution to the problem is preferred. As it then allows a variety of other TCP congestion control algorithms to take in use the solution.

1.3 Limitations

- Our research is solely focused on the use of ECN as a signal of congestion. The loss recovery of TCP is already heavily researched, and our work will *not* ignore it, but we will not focus too much on it.
- Our experiments are limited to the Linux kernel version 5.0.
- We limit our experiments to the Reno and DCTCP congestion control algorithm. Reno is the most commonly understood congestion control algorithm of TCP. We do not need to add more complexity with other algorithms to show the prevalence of the problem. DCTCP, on the other hand, is commonly used inside data centres. DCTCP is mainly used to keep a short queue; we will conduct experiments to see how it reacts when trying low queuing delay over a shallow RTT.

1.4 Main Contributions

- **Can we extend TCP for shallow base RTT?**

  Yes, we have modified TCP such that the sender adds a local transmission delay to keep a lower transmission rate. The protocol is now capable of scaling in environments which mandates a shallow transmission rate. We describe a solution where the transmission rate has no lower bound unless the implementer sets an artificial limitation.

- **Can we preserve stretch acknowledgements for low congestion window values?**

  Yes, we have designed the solution around this requirement. The solution itself does not depend on when the receiver decides to send an acknowledgement. The sender transmits packets at the minimum of the stretch acknowledgement factor but adds delays between its packets. The sender later subtracts the added delay. The solution is meant to scale well with significant stretch acknowledgement factors.

- **Can we preserve a congestion window made out of segments in a network with a low transmission rate per RTT?**

  Yes, we propose the use of fractional congestion window. The congestion window extends to have an additional congestion window which allows finer transmission adjustments.

- **Are we able to create a minimal solution which is easy to implement?**

  Yes, the solution requires minimal modification to the sending host and is, therefore, out of the box partially deployable over the Internet. We not only made the solution simple but additionally made ease for other congestion control algorithms to modify the rate of the sender using a one confined congestion window.

1.5 Statement of Originality

The work of this thesis is solely the work of the author in cooperation with the supervisor. The supervisor, Bob Briscoe, has been of massive help and has helped make the correct design decisions. The contribution of this thesis to the problem would not have given such a great outcome if it had not been for the supervisor.

To our best knowledge, the work of thesis is original and does not exist in any other publications[1] except for work where we have given the explicit reference to the rightful person or group of persons. The work of this thesis has been a collaboration between the University of Oslo (UiO) and Simula Reseach Laboratory (Fornebu) and contributes to the Reducing Internet Transport Latency (RITE) project[2]. However, any opinions expressed in this thesis remains the word of the author and no one else.

1.6 Research Methods

This thesis base its research methods on the final report from ACM: "Computing as a Discipline" [21]. This report tries to address the long debate on whether computer science is real science. The ACM published the report on their 42nd anniversary in cooperation with the IEEE. This report gives out reasons why computer science conducts real science and gives out a guideline — our work bases on the principles of how an engineer goes at solving a problem: we first state requirements and specifications. We then characterise and quantify of the prevalence of the problem. We propose a design and derive an implementation in an attempt to solve the problem in an open-source manner into the Linux kernel. We then evaluate our solution against the existing current best solution. We do design, implementation and evaluation in one phase, and we repeat this phase until we cannot further improve the results. Finally, we conclude our work and gives way for others to continue our work.

1.7 Outline

This thesis is structured as follow:

- *Chapter 2: Background*

  The thesis begins with the technical background of the problem. We list as many details as needed to pinpoint the problem. The reader will fast understand the complexity of the problem. We must understand congestion control and loss recovery mechanism of TCP to at a fine granularity. We further need to understand the Linux kernel implementation of the TCP stack. A significant amount

[1] We wrote a paper and presented our main ideas in Netdev 0x13 [15], but the full project is first now made public

[2] https://riteproject.eu/dctth/

of work has been done first to explain TCP and how some of its mechanisms conflict with our goal at achieving a low RTT in the network. Another focus of this thesis directs toward data centres where the problem is more common due to the low base RTT between the processing nodes.

- *Chapter 3: Quantification*

  The thesis quantifies the prevalence of the problem. We look into how prevalent the problem is by investigating the perfect condition needed for the problem to exist. We emphasise the vital role TCP play in keeping long queue at the bottleneck by proactively overriding the recommendation of the AQM.

- *Chapter 4: Related Work*

  The thesis lists earlier work related to the problem. Earlier work mostly talk about low BDP in situations where there is a magnitude of senders at the bottleneck. This thesis addresses the same problem with low BDP, but our primary motivation is to extend TCP for low base RTT.

- *Chapter 5: Methodology*

  The methodology chapter gives our experiment plan, which is needed to conduct the proper experiments to confirm the prevalence of the problem. We also list the metrics we use for our evaluation of TCP. Finally, we make our configuration of the testbed transparent.

- *Chapter 6: Design*

  We propose a design. This chapter gives in detail the modifications needed for TCP to work in a network with shallow base RTT.

- *Chapter 7: Implementation*

  We implement our work from the design proposal into a stable version of the Linux kernel. We work out a Linux kernel module to fulfil our needs and also make the necessary modification to the input and output engine of TCP to support this module. Our prototype is made simple with the intended goal to test out the main ideas of the design proposal.

- *Chapter 8: Evaluation*

  We evaluate our implementation under the Linux kernel. We test the implementation for stability, durability and performance. We mainly test if our design and implementation work as predicted.

- *Chapter 9: Results*

  In the resulting chapter, we present data from our experiments. We use our strategy from the methodology chapter to conduct the proper experiments. We compare our results against the conventional TCP to see if we were able to succeed in scaling TCP for low base RTT.

- *Chapter 10: Conclusion*

  Finally, we express a summary of our work and list our contributions. We list future works which we did not manage to address or was out-of-scope for this thesis. Others can thus continue this research where we now end.

Chapter 2

Background

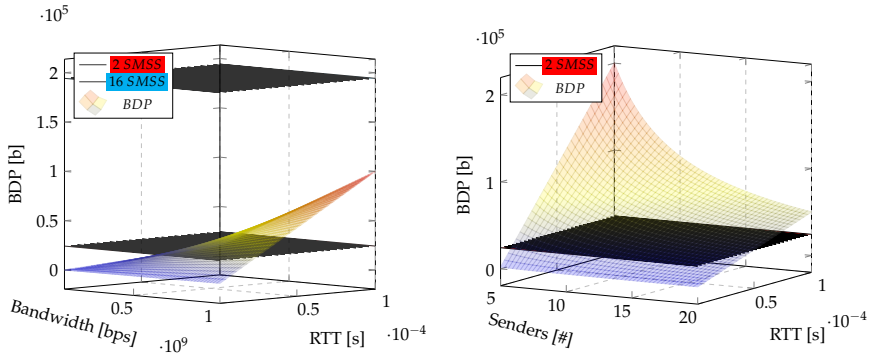
2.1 Bandwitdth Delay Product (BDP)

Bandwidth Delay Product (BDP) stands for the number of bits a sender must keep inflight to utilise the link capacity entirely[40]. In other words, BDP is the number of bits the sender must emit every RTT to keep the link utilised. The calculation of BDP is, therefore, just the product of the link speed (bits/second) and the RTT (seconds), in bits (see Equation (2.1)). The sender emits the application data in a quantum of octets/bytes (8-bits) which again assembles into whole segments.

$$BDP = Bandwidth * RTT \tag{2.1}$$

Figure 2.1a shows how BDP varies with variety of bitrates and RTTs. It is quite obvious from this figure that shallow RTTs yields a very low BDP compared to some higher RTTs. Higher bitrate appears to have almost no impact on how fast BDP increases for lower RTTs. The lower bound of TCP, which is two segments, is added to show shallow RTTs which a non-responsive sender cannot utilise. The sender has no other one to blame than himself.

A receiver usually acknowledges every other segment to lower the



(a) A single sender with upto 10 Gbps link (b) Mulitple senders sharing a 10 Gbps link

Figure 2.1: BDP for shallow RTTs

Flows (#)	**1**			**5**			**10**		
Bandwidth (Mbps)	5	10	20	5	10	20	5	10	20
RTT (ms)	Fair Rate (Kb)								
1	5	10	20	1	2	4	0.5	1	2
2	10	20	40	2	4	8	1	2	4
5	25	50	100	5	10	20	2.5	5	10
10	50	100	200	10	20	40	5	10	20
RTT (ms)	Congestion Window (segments) $\approx$								
1	0.41	0.82	1.65	0.08	0.16	0.33	0.04	0.08	0.16
2	0.82	1.65	3.29	0.16	0.33	0.66	0.08	0.16	0.33
5	2.06	4.12	8.23	0.41	0.82	1.65	0.21	0.41	0.82
10	4.11	8.23	16.46	0.82	1.64	3.29	0.41	0.82	1.64

Table 2.1: Table of BDP (multiple flows)

processing cost of the network. This technique is generally known as usage of stretch acknowledgement (see Section A.1.2). A receiver is said to have generated a stretch acknowledgement in this case. The benefits of lowering the processing cost are the desired goal of data centres to utilise higher bitrates with as low processing cost as possible. These benefits are not limited to a data centre type network as a lower processing cost is beneficial to a set of devices running on a portable power source. It further benefits the wireless network as a whole since fewer acknowledgements are inflight per RTT. A lower bound of 16 segments is also plotted to illustrate what limitation it opposes to an implementation which tries to keep a more substantial acknowledgement stretch factor.

Unfortunately, a typical scenario in computer networks is competition from other senders. This possibility may further require the sender to limit its transmission rate to share out the link speed evenly. The BDP formula (Equation (2.1)) can easily be extended to equally share the link speed among multiple competitors (see Equation (2.2)).

$$fair\ rate = Bandwidth\ /\ nr\_of\_senders * RTT \tag{2.2}$$

$$W_s = Bandwidth\ /\ (nr\_of\_senders * 1518B * 8) * RTT \tag{2.3}$$

Table 2.1 shows fair rates based on an increasing number of flows, link speed and base RTT, in Kb. The table also shows how large the TCP state variable congestion window must be of each sender, in segments (Equation (2.3)). We use the typical Ethernet frame size of 1518B for the calculation of the congestion window. BDP drastically reduces as more competition occurs in the network. The BDP significantly reduces as both lower base RTT, and more competition takes place in the network. Therefore, to keep a link fully utilised with a lower BDP must each sender emit fewer bytes per RTT. Figure 2.1b shows how fast the BDP drops below the threshold of two segments for bitrate as high as 10 Gbps when the

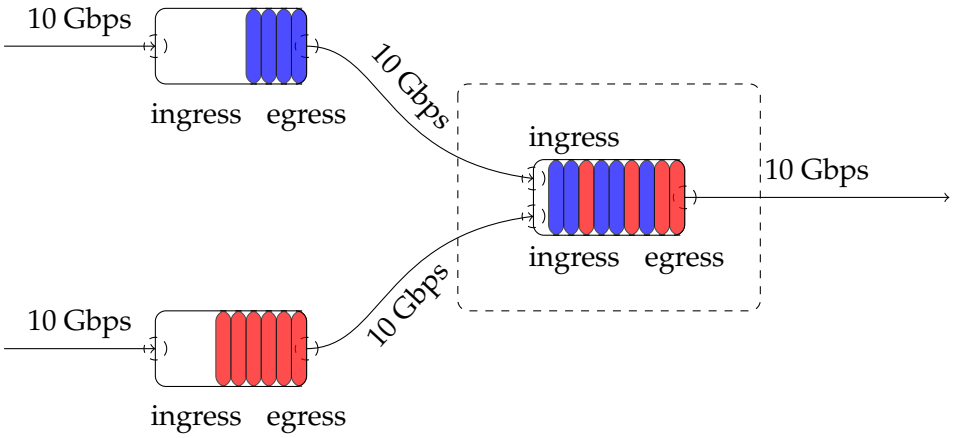


Figure 2.2: Network elements with a single queue

dynamics of both the number of senders and base RTT plays together against the higher bitrate.

2.1.1 Long, Fat Network (LFN)

Long, Fat Network (LFN)[11] are networks with an extensive BDP path. A sender must send a massive number of segments within a single RTT to utilise the link fully in this environment. Meaning, there is a high risk of reusing old segment numbers for new segments. Old segment numbers might still be inflight, and therefore unsafe to use for any new segment. Changes were made to TCP to utilise the link capacity on such network path efficiently. Timestamps (TS) were added to TCP for Protection Against Wrapped Sequences (PAWS) and Round-Trip Time Measurement (RTTM).

The flow windows (see Section A.2.2) was also extended with the ability to scale well beyond 16 bits with the TCP Window Scale (WS) option. A sender cannot have BDP path longer than 64 KiB in the absence of such scaling. WS option allows the receiver to announce a much larger receiving window. The sender now uses the scaling factor and the flow window to yield the permitted flow window.

LFN networks were all about solving a massive BDP path problem are we here trying to address the low BDP problem caused by shallow base RTT. The link speed of a low BDP path could be of the same order as a massive BDP path. We see it of importance to emphasise that the low BDP problem have very little to do with the bitrate when we push RTT to a significantly lower level. A sender who scales for low BDP paths has the benefit of utilising the same link speed with a lower base RTT.

2.2 Queuing Delay

A network element on the Internet uses a buffer as a storage space for packets [62]. The network element has a maximum rate for each of its links

and a link can either be an ingress or egress link. Each egress link requires a buffer to maintain link utilisation (see Figure 2.2).

The input rate of an ingress link is how fast packets arrive. Likewise, the output rate of an egress link is how quick packets departure to the next hop. An output rate of an egress link can have traffic from multiple ingress links. The moment the combined input rate of all ingress links exceeds the maximum output rate of an egress link will queue start to build of packets. The buffer is used to temporarily hold packets until the first possible transmission is possible to the next hop.

A simple First In First Out (FIFO) queue enqueues the newly arrived packet and dequeues the packet first when all previous packets have departed from a given egress link. The network element is forced to drop any new arriving packets when eventually the limited buffer runs out. The sender of the lost packet then gets to decide if retransmission is needed. Packets start to enqueues again when the head of the queue dequeues a packet. Queueing delay occurs when packets stay stored in the buffer waiting for their transmission. The combined input rate of ingress links must be reduced for a while to let the queue dry out of packets again. In order to fully utilise an egress link must the combined input rate of all ingress links be equal to the maximum output rate of the egress link. However, if there is already an outstanding queue on the egress link must the combined input rate of all ingress links be lower than the maximum output rate of the egress link. The queue will then get a chance to dry out of packets.

The desired goal is to have a short queue as it allows minimal queuing delay through the network element which happens to be the bottleneck, so new packets that may arrive at the bottleneck will then only wait through the shorter queue. All senders traversing the bottleneck then achieves a lower RTT as their packets spend less time waiting in the queues of the network.

2.3 TCP Background

The background of TCP is vast and commonly understood. An excellent understanding is needed of TCP to be able to realise how hard the problem of scaling to lower congestion window value is (see Appendix A).

2.4 TCP Congestion Control in Linux kernel

The Linux kernel implements the congestion control for TCP[63]. Many servers and network elements in the infrastructure of the Internet uses Linux, and the performance of the Linux kernel TCP implementation is therefore essential. The Linux kernel is open source; it is therefore significantly used in research and updates throughout its lifetime. While the problem persists in all the standard TCP implementations will this paper only explain the implementation found in the Linux kernel. Linux TCP implementation not only follows the guideline set by the IETF through

Request for comments (RFC) but also make additional modifications to ensure the best overall performance for all participants on the Internet.

Congestion window in the Linux TCP only use a concept of whole segments; the consequence is that Linux TCP considers every packet as one segment in the congestion window regardless of its size. Linux TCP is, therefore, more conservative than what is permitted by the TCP standard.

The Linux kernel also implements the TCP NewReno algorithm as a non-removable module and used as a fallback by other congestion control algorithms. The Linux kernel let the user load congestion algorithms dynamically. Alternatively, the application can decide, if permitted, to use another algorithm on a per-socket basis. A congestion control Linux kernel module controls the additive increase and multiplicative decrease factor of the connection. The Linux kernel has, otherwise, joint code for the other mechanism of TCP. Meaning, loss recovery is taken care of by the Linux kernel which decrements the congestion window toward the ssthresh value set by the module. The user controls the use of SACK information through a global setting. The host then negotiates the use of SACK information with the remote host.

2.4.1 State Machine

The Linux kernel uses states for each of its TCP connection, the connection by default starts in the *open* state. The TCP connection is in the normal flow in the *open* state which means it is probing for more capacity.

The sender enters the *disorder* state upon reception of a duplicate acknowledgement or an acknowledgement with SACK information. The Linux kernel preserves the congestion window in this state and triggers transmission of a new segment on each incoming acknowledgement.

The sender enters *Congestion Window Reduced (CWR)* state as an Explicit Congestion Notification (ECN) mark is received. The *CWR* state mandates the sender to decrease its congestion window and exits the state if the congestion window reaches the new value of ssthresh. This state is also declared over if the sender needs to retransmit a segment, which can happen if the sender enters *recovery* or *loss* state.

The reception of three successive duplicate acknowledgements triggers the *recovery* state. This state is mainly the fast retransmit and fast recovery algorithm. The sender also reduces its congestion window as in the *CWR* state. The sender transmits the first unacknowledged segment and stays in the state until all transmitted segments before initiating *recovery* has been acknowledged (NewReno/SACK). The sender goes over to the *open* state again and initiate the congestion avoidance algorithm once the recovery phase completes. Otherwise, the sender has no choice, but to enter the *loss* state.

Finally, the sender enters the *loss* state when the Retransmission Timeout (RTO) expires. All segments inflight are assumed lost, and the sender reinitiates the slow start algorithm with a congestion window of only one segment. The sender may only leave this state if the receiver acknowledges all segments sent before entering this state.

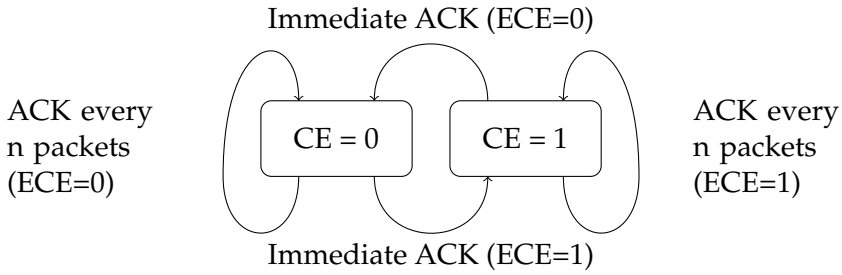


Figure 2.3: DCTCP ECN ACK State Machine (figure 10 of DCTCP paper)

The Linux kernel does not transmit any new segments in any of the states mentioned above. The approach is to instead update TCP state variables in each of the states and have a common place for the transmission of new segments. Linux TCP sender transmits a new segment if the amount of segments in flight is less than the updated congestion window. Additionally, the Linux TCP sender avoids transmission of more than three new segments for each acknowledgement received. This technique avoids excessive bursts in situations where the sender suddenly is allowed to transmit a bucket of segments at once.

2.5 Data Centre TCP (DCTCP)

Data Centre TCP (DCTCP)[7] is a congestion control algorithm often used inside data centres. The traffic of a data centre is often a mix of greedy and delay critical flows. A data centre, therefore, needs the best of both worlds, which means low latency and high throughput between hosts is essential. Traditional TCP does not scale well in networks with those requirements. The congestion window reduction phase of TCP effectively lose a significant part of the congestion window, and the process of regaining the previous window requires an enormous amount of RTTs for bitrates typically found in data centres. The inefficiency of TCP translates to the need of a massive queue at the bottleneck.

Meanwhile, a DCTCP sender reduces its congestion window more gradually, so the AQM can get away with a shorter queue. The AQM can then be easily configured to do a much more aggressively marking. The AQM, therefore, for DCTCP usually have a single marking threshold without any burst. The AQM is then set to mark as soon as the queue gets higher than the selected threshold.

2.5.1 Improved ECN Signaling

An ECN-Capable TCP sender detects congestion if one of the segments transmitted gets ECN marked. Recall from earlier the receiver repeats ECE-bit until the sender responds with CWR-bit. This feedback does, therefore, not tell the sender much more than to reduce its transmission rate. Although, an ECN mark confirms true congestion in the network

meanwhile packet loss is just merely an assumption of congestion based on a timeout.

A DCTCP implementation uses a modified version of the ECN to detect the number of bytes that caused the congestion. The sender can then update its congestion window precisely. The sender gets this information through improved ECN signalling done by the receiver. The receiver uses an acknowledgement state machinery to decide when to transmit an acknowledgement with the ECE-bit set (see Figure 2.3).

The DCTCP implementation has a boolean state variable to decide when to transmit an acknowledgement with the ECE-bit set. The state variable is updated based on the CE codepoint received. Reception of the first CE codepoint changes the state variable from false to true and triggers acknowledgement immediately with the ECE-bit set. The acknowledgement with the ECE-bit is then repeated and sent after every n segments received. The state variable changes back to false when the first segment comes without the CE codepoint and triggers another immediate acknowledgement. This time the acknowledgement does not have the ECE-bit set. The receiver also repeats this acknowledgement same as earlier, but now without the ECE-bit set and do so until a CE codepoint is once again received. The receiver is then able to forward the current level of congestion immediately to the sender. The DCTCP sender measures the duration of the congestion based on the number of segments marked. The DCTCP sender can thus pinpoint the exact amount of bytes that caused the congestion.

2.5.2 Congestion Window Reduction

The sender updates its congestion window more gradually (see Equation (2.4)).

$$S_s = max(inflight_s * (1 - \alpha/2), \quad 2_s) \tag{2.4}$$

DCTCP use alpha as a moving estimate of the number of segments marked within an RTT. The DCTCP sender updates this estimate for every acknowledgement received. The initial estimate of the fraction alpha is 1. Meaning, DCTCP reduces its congestion window by exactly half. The value of alpha updates according to Equation (2.5). The congestion window is reduced more gradually as the equation yields a smaller alpha.

$$\alpha = \alpha * (1 - g) + g * F \tag{2.5}$$

The DCTCP sender can now efficiently utilise high bitrates with a low queue at the bottleneck. However, a DCTCP sender does not let its congestion window fall below two whole segments in an RTT as the conventional TCP. DCTCP experience the same problem as the conventional TCP and does not scale well for very low RTTs.

2.5.3 Deployment Issues

In order to deploy DCTCP must full control of the network be necessary, that is control of each host and the infrastructure made out of switches. The benefits of using DCTCP is only relevant if the whole network relies on ECN. However, this is not the case for the Internet. DCTCP does neither rely on negotiation between hosts since it was designed to use in a controlled environment. A TCP endpoint cannot detect if the sender is using DCTCP.

Conventional TCP does not compete evenly with DCTCP over the Internet since the conventional TCP reduces its congestion window much more aggressively than DCTCP on a mark. The AQM on the internet is neither set up to mark when the instantaneous length of the queue exceeds a threshold, but rather on an average queue estimate. The DCTCP sender, therefore, also lack fast feedback from the receiver to detect changes in dynamic of the queue length.

DCTCP must, therefore, not be deployed over the Internet as is. However, DCTCP could work as L4S (see Section 2.6) traffic since the regular traffic segregates into a separate queue. The DCTCP sender must still fall back to standard TCP in the event of packet loss. Packet loss is an indication that there may be network elements which do only have a single queue. The fall back ensures DCTCP reacts same as TCP in scenarios where the network has legacy network elements which share queue for all its traffic.

2.6 Low Latency, Low Loss, Scalable Throughput (L4S)

L4S[16] is an ongoing project to build a clean state Internet service with the purpose of achieving Low latency, Low Loss, Scaleable throughput (L4S) in the network. The L4S architecture is designed to be deployed over the Internet incrementally with the use of dual queue AQMs (see Figure 2.4). The primary goal of the architecture is to deliver low latency to its traffic.

The L4S network has scalable senders who use a modified version of ECN. The idea of using a modified version of ECN in the network has the same goal as DCTCP has in data centres. The use of modified ECN let the sender make a more gradual reduction of its congestion window.

It is not possible to achieve this goal with the traditional TCP traffic without Dual Queue AQM as managing L4S scalable senders requires a more aggressively marking. The conventional TCP does neither respect L4S requirement of a shallow queue. Finally, the constant add in the additive phase of TCP is problematic when the AQM aims for shallow RTTs.

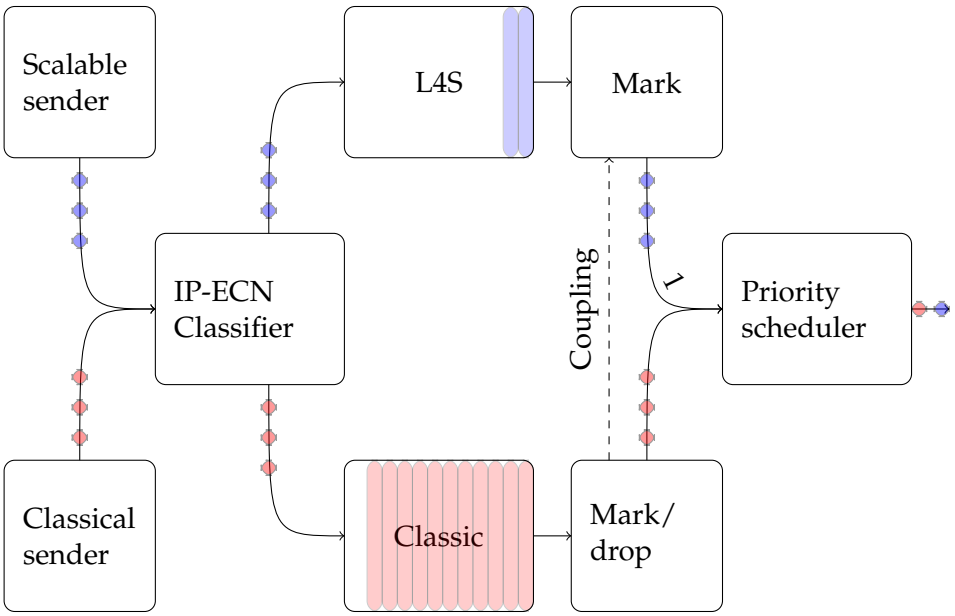


Figure 2.4: L4S Architecture [16]

2.6.1 Dual Queue AQM

A Dual Queue AQM segregates the per-flow or per-class scheduling algorithms. The Dual Queue AQM take care of queue management between these scheduling algorithms[6] which makes implementation of new scheduling algorithms more straightforward since each of the scheduling algorithms has their unique queue.

The new scheduling algorithm then does not need to compete with the legacy TCP traffic. The legacy TCP traffic does not always respect the Quality of Service (QoS) goals set by the new scheduling algorithm. The legacy TCP traffic either force a long queue and generally struggle to keep up with the excessive signals emitted by the AQM. The fix cannot be done to the conventional TCP yet as there will always be some sender who does not respond correctly to the requirements set by the scheduling algorithm. The problem, legacy TCP, is moved into its queue where it cannot disturb the balance of other queues. Although, the use Dual Queue AQM should not cause starvation of some scheduling algorithm. The AQM should also refrain from keeping packets of some algorithms for unnecessarily long periods.

2.6.2 L4S ECN

In the L4S architecture, the use of ECT(1) codepoint let the AQM detect L4S traffic[10]. The conventional TCP traffic must only use the ECT(0) codepoint. The AQM can then respond appropriately for the legacy TCP traffic. The Dual Queue AQM signal the L4S traffic much more aggressively to achieve low queuing delay since the L4S traffic uses an ECN mark to reduce its transmission rate less aggressively.

2.6.3 TCP Prague

TCP Prague will be one of the new generations of TCP congestion control algorithm which takes part as L4S traffic. However, there is a set of requirements which TCP Prague or any other modern congestion control must fulfil. One of the requirement is to operate in the sub-mss regime. A new way to clock out packet is needed to allow congestion window sizes needed to operate in this regime. When operating in the sub-mss regime, the congestion window must not increase as aggressively as in the conventional TCP. Increasing by one full segment per RTT is too aggressive when the congestion window may be less than one segment.

2.7 Summary

We have talked about BDP and how queueing delay has its role in networking. We also explained the vast background of TCP in order to pinpoint the problem. We then briefly explain how Linux implements TCP. DCTCP was introduced to explains its benefits. Finally, we talked about L4S and how it requires a solution to this problem.

Chapter 3

Quantification

We shall now quantify the prevalence of the problem by investigating how TCP performs in a set of network topologies typically found on the Internet. We address the question of how likely the problem is to occur within each of these topologies. With this work, we would like to give reasons for why the network is not in the fault for causing the problem, and why the limited scalability lies in the state-of-art TCP.

First, we test our hypothesis on a data centre like a network to emphasise the importance of how a low propagation delay serves in impacting the performance of the protocol. A Data centre type network is what motivated our work. We saw TCP becoming unresponsive on a shallow buffered AQM over an already tiny base RTT [14]. The unresponsiveness of TCP resulted in a steady growth in queueing as the number of competitors increased.

Secondly, we evaluate the likelihood of the problem to happen in the typical environment of TCP on the Internet, generally known as broadband. The propagation delay is not as low as in data centres due to the geographical placement of the sender and receiver. The lowest achievable base RTT grows as the distance between the endpoints increases because a signal can only travel approximately as fast as the speed of light. Our goal here is to investigate if a more significant base RTT is enough to hinder the problem.

Finally, we look at one last not so ideal environment for TCP, the now increasing use of TCP in place for Inter-Process Communication (IPC) over an unknown network topology. IPC is indeed a communication channel with a shallow base RTT. The evaluation of isolation over the same physical host, e.g. Virtual Machines (VMs) and containers, is in no way an execution environment for TCP. However, the now increasing use of migration of application closer to its peer makes it impossible to assume the location of the other endpoint. An application may be communicating to a local peer without realising it, so we see the need for a more scalable TCP to perform well over IPC topologies. By going from a data centre to broadband we were purely interested in knowing when the problem disappears (best case), but now by doing IPC, we want to see the other side of the spectrum and see how bad it can get (worst case).

3.1 Data Centres

Data centres typically operate at much higher bitrates and lower base RTT than the traditional broadband networks [8, 9]. A data centre tries to maximise throughput between processing nodes with an extensive bitrate in the network. The processing nodes are also tightly packed together to achieve a shallow base RTT. A data centre is often set up with a higher Maximum Transmission Unit (MTU) between its processing nodes. The advantage of using larger packets is to yield an even better efficiency of the network by reducing the overhead caused by network headers.

Cisco published a whitepaper which forecasts the evaluation of data centres in the period 2016-2021 [51]. This papers forecasts are almost doubling in hyperscale data centres and believe these type of data centre will stand for half of the total data centres. A hyperscale data centre is a public cloud with excessive yearly revenue and is backed up by some of the world-leading cooperations. Hyperscale data centres have excellent storage and networking capabilities. These type of data centres are used by cooperations to deliver a better quality of service to their massive user base. Another use case for such data centres is their significant processing capability in Big Data processing.

The bitrate typically found in a data centre range from 1 Gbps and may be as high as 40 Gbps. The base RTT in data centre usually lies in the sub-millisecond region and could be as low as some microseconds. Additionally, a data centre vendor might use frame size as significant as 9KB (jumbo frames) to yield better efficiency of the network [14, 57].

We now have the data we need to calculate the appropriate congestion window for **four** senders:

$$W_s = 40Gb \;/\; (4 * 9018B * 8) * 10\mu s \tag{3.1}$$

$$W_s \approx 1.39 \tag{3.2}$$

We try bitrate remarkably as high as 40 Gbps to illustrate the fundamental problem with TCP and its derivatives. We use DCTCP in our example as the congestion algorithm optimises for the data centre environment. Although, the problem also applies to a broad set of other commonly used congestion control algorithms. We only need to include four senders to show the problem in action. A 10 Gb throughput should be more than enough for DCTCP to keep a very short queue at the bottleneck. DCTCP in comparison to TCP only reduces its congestion window gradually, so DCTCP should stabilise to a very short queue selected by the AQM. However, the critical question remains how fast should the sender go to keep a ten µs RTT. The answer to that question might be shocking to some, but the answer is no larger than 1.39 segment per RTT (see Equation (3.2)). DCTCP will under no circumstance let its congestion window fall below two segments per RTT. Our handful of four DCTCP senders push the effective RTT to 15 µs.

One might think that is not bad at all, and it is quite easy to blame the network. One could argue that adding more bitrate prevails the problem.

Bandwidth (Gbps)	1	2	5	10	20	30	40
Base RTT (µs)							
10	0.03	0.07	0.17	0.35	0.69	1.04	1.39
20	0.07	0.14	0.35	0.69	1.39	2.08	2.77
50	0.17	0.35	0.87	1.73	3.47	5.20	6.93
100	0.35	0.69	1.73	3.47	6.93	10.40	13.86
200	0.69	1.39	3.47	6.93	13.86	20.79	27.72
300	1.04	2.08	5.20	10.40	20.79	31.19	41.58
400	1.39	2.77	6.93	13.86	27.72	41.58	55.44

Table 3.1: Table of congestion window sizes in data centres for 4 sender

The network must deliver roughly an additional 20 Gbps to satisfy our four DCTCP senders. Now let us imagine, a not uncommon scenario, four additional short-lived DCTCP flows joins and start probing for capacity. The congestion window is now barely one segments per RTT for 60 Gbps and, as we know, DCTCP cannot go this low. These DCTCP flows double the RTT, now 20 µs, of the network with unnecessarily queue at the bottleneck. Do we still blame the network? No, there is no reason why the network has to deliver a staggering 120 Gbps to keep these senders happy. This flaw is the fault of DCTCP and is genuinely how traditional and modern TCP congestion control algorithms works. This flaw has always been there, and for DCTCP, it is just more severe as the network is capable of delivering a shallow base RTT.

DCTCP has built-in support for the receiver to maintain a stretch acknowledgement factor of uncertain size to keep the processing cost of the network low as possible [7]. However, a DCTCP sender must then keep its congestion window at a minimum of this new stretch acknowledgement factor to overcome the delayed acknowledgement mechanism of the receiver. Up until now, we assumed a stretch factor of two segments per RTT, so let us see where this goes. For the sake of efficiency, DCTCP might be set up to acknowledge every sixteen segments. As earlier, a DCTCP sender cannot and will not hold a congestion window of 1.39 segments per RTT (see Equation (3.2)), so our four senders induce a base RTT as significant as 120 µs since they refrain from using a lower congestion window than sixteen segments (see Table 3.1). Now, how much bitrate is the network lacking? The network must provide our senders with a total of roughly 460 Gbps, so they can keep clocking out sixteen segments per RTT. As we see, a more significant stretch acknowledgement factor cause the sender to stabilise to a higher minimum congestion window, which results in higher RTTs in the network. A low congestion window is unavoidable for a shallow base RTT, and it gets worse when larger packets and more senders are present in the network. The bitrate *alone* is of little help and cannot keep the congestion window above the more sustainable stretch acknowledgement factor. The argument for applying more bitrate to fix a queueing problem does not hold as the counter-argument, which is to

scale the congestion window below the stretch acknowledgement factor, makes the achievable RTT less dependent on the bitrate of the network. An AQM would generally never try to keep less than two packets in the queue since the risk of running out of packets increases, and link underutilisation becomes unavoidable. By increasing the bitrate, the queueing delay of two packets declines proportionally.

For a data centre type network has a shallow RTT shown to be enough to break down some of the fundamental mechanisms of TCP. Even though, TCP shares the bitrate reasonably is it of little use when the AQM must keep a long outstanding queue at the bottleneck.

In other words, TCP does not scale for shallow RTTs or any significant stretch acknowledgement factors.

3.2 Broadband

In broadband, the situation is a bit different as the environment is not so controlled anymore. The physical distance between endpoints is variable and several times longer than in data centres. The distance alone makes it for a variety of cases impossible to achieve base RTT of less than one millisecond[1]. The bitrate also severely degrades compared to a data centre as it depends on the infrastructure of the specific country and is indeed no more than 1 Gbps. The achievable MTU is limited as well since all network elements in the path between the sender and receiver have to support the larger MTU.

The bitrate typically found in broadband networks ranges from as low as 2 Mb and may be as high as 400 Mb. On the other hand, the typical base RTT start as low as one millisecond and is at its worst at 60ms. The frame size is often no more than 1500 B [14].

We can as before calculate the appropriate congestion window, but now with **32** senders:

$$W_s = 320Mb \; / \; (32 * 1518B * 8) * 1\text{ms} \tag{3.3}$$

$$W_s \approx 0.82 \tag{3.4}$$

Broadband networks, at first glance, appear immune to the problem due to smaller packets and the higher RTT present in the network. We try a quick link as high as 320 Mb with base RTT of one millisecond. We imagine a scenario of 32 senders sharing this bitrate. A bitrate of 10 Mb should be more than enough for TCP to clock out the appropriate number of segments per RTT. For instance, a 10 Mb with a frame size of 1518 B translates to roughly 823 segments per second. However, recall from earlier that TCP has no interest in transmitting less than two segments per RTT. A base RTT of one millisecond translates to 2000 segments per second. The

[1]The now increasing use of Content Delivery Networks (CDNs) and Data centres helps in distributing the workload of a single system through the use of geographically nearby proxies. The use of nearby proxies helps in minimising the cost of serving adjacent peers across continents.

Bandwidth (Mbps)	10	20	50	100	200	300	400
Base RTT (ms)							
1	0.01	0.01	0.03	0.06	0.13	0.19	0.26
2	0.01	0.03	0.06	0.13	0.26	0.39	0.51
5	0.03	0.06	0.16	0.32	0.64	0.96	1.29
10	0.06	0.13	0.32	0.64	1.29	1.93	2.57
20	0.13	0.26	0.64	1.29	2.57	3.86	5.15
30	0.19	0.39	0.96	1.93	3.86	5.79	7.72
40	0.26	0.51	1.29	2.57	5.15	7.72	10.29

Table 3.2: Table of congestion window sizes in broadband for 128 senders

sender becomes unresponsive at this point and continues to clock out at a minimum of two segments per RTT no matter how congested the network path is. The sender has thus failed in its mission to keep a rate of 10 Mb and is instead holding a rate of roughly 24 Mb where the remeaning 14 Mb translate into a queue at the bottleneck. Meaning, 32 senders contributes to roughly 450 Mb of a queue at the bottleneck. As we see, TCPs contribution to the queue doubles as senders present at the bottleneck doubles.

Now back to the idea of applying more bitrate to fix a queueing delay problem. The network has to provide roughly 770 Mbps, bitrate of 320 Mbps plus 450 Mbps of a queue, to keep our 32 senders in check. Does not sound that bad? The network has now enough capacity to let TCP do its two segments in an RTT. As the network is not usually capable of achieving a lower RTT or larger packets in a broadband setting is our last resort to introduce more senders to test TCPs scalability. We can try to double the number of senders to see which implications TCPs floor sets to the network. As we go for 64 senders is the now required bitrate to overcome the machinery of TCP as significant as 1.56Gbps. Let us not stop there, a further doubling to 128 senders moves the requirement of the network to roughly 3 Gbps (see Table 3.2). Meaning, the network has to double its bitrate whenever there is a doubling in the number of senders present at the bottleneck.

In other words, TCP does not scale well for a magnitude of senders with a moderate base RTT.

3.3 Inter-Process Communication (IPC)

Inter-Process Communication (IPC) refers to the communication channel between processes on the same host. IPC offers much of the same end-to-end performance as one would find in a data centre, meaning excessive bitrate combined with a shallow base RTT. The RTT between two processes is usually no more than a few microseconds and may even be as low as a couple of hundred nanoseconds. The Operating System (OS) provides several efficient and low-cost communication channels between

two running applications [41], e.g. through pipes, shared memory, local sockets or mailboxes. However, the use of containers and virtual machines makes it nearly impossible for an application to detect its current execution environment. For instance, applications intended to run in separate physical hosts initially could eventually shift to isolated environments located just in one physical machine. Applications are now more likely to migrate closer geographically in terms of data centres to its peer to keep the obtainable latency as low as possible. Thus, we need TCP to perform well in doing IPC in place for other more fit solutions due to the limited knowledge application has of the current execution environment.

An application could then use TCP for IPC to achieve both portability and abstraction from OS-specific IPC implementations. During a migration process, the communication channel between two running applications can be kept as is without requiring another mean of communication. It is better to make one protocol fit a changing execution environment than to synchronise the use of other non-widely used protocols back and forth. A typical TCP/IP implementation remains a complex beast and one generally avoid it for IPC because of the computational footprint it has. However, we might not always be able to distinguish a local peer from a remote one, so the use of a transport protocol like TCP might be required.

Another problem with using TCP for IPC has not surprisingly also to do with a very tiny RTT between the sender and receiver. The OS may optimise aswell and build large segments since the application data never leaves the physical host, so no external limitations impose for what defines the maximum size of the segment. The segment size could, for example, be 65KiB.

We now have the data we need to calculate the appropriate congestion window for **one** sender:

$$W_s = 400Gb \;/\; (65KiB * 8) * 1\mu s \tag{3.5}$$

$$W_s \approx 0.76 \tag{3.6}$$

A single sender is enough to cause trouble when using TCP for IPC. The bitrate is incredibly 400 Gbps but is in no way enough to go around the problem (see Equation (3.6)). A good network consists of two simple fundamental properties; high bitrate and low propagation delay. So, as the bitrate goes sky high, a similar trend happens to the base RTT, thus negating the need for a high transmission rate or a large congestion window. It also does not help, for TCP, that the size of the segment is now much more significant. One could think that by refraining the use of large segments alleviates the problem to some degree, but this increases the overhead footprint of TCP. A better approach would be to make TCP hold a lower transmission rate and let it use the maximum segment size available.

The bitrate must keep doubling for when the base RTT decreases on a network with unresponsive senders. However, what if the bitrate is already high, like in IPC? For instance, continuing on the previous example, the

Bandwidth (Gbps)	10	20	50	100	200	300	400
Base RTT (µs)							
1	0.01	0.02	0.05	0.10	0.19	0.29	0.38
2	0.02	0.04	0.10	0.19	0.38	0.57	0.76
5	0.05	0.10	0.24	0.48	0.95	1.43	1.91
10	0.10	0.19	0.48	0.95	1.91	2.86	3.81
20	0.19	0.38	0.95	1.91	3.81	5.72	7.63
30	0.29	0.57	1.43	2.86	5.72	8.58	11.44
40	0.38	0.76	1.91	3.81	7.63	11.44	15.26

Table 3.3: Table of congestion window sizes in IPC for 2 sender

system has to deliver roughly 1 Tbps to satisfy one single malfunctioning sender. A doubling in the RTT may sound like a doable compromise, but this is a burden which contradicts with our intended goal, and no doubt is the motivating factor of our work. TCP should **never** be allowed to decide when to overrule the AQM and become unresponsive, at least not without any consequences. A proper penalty for an unresponsive sender will as always be an AQM performing an eviction policy; TCP should, therefore, cooperate with the AQM or risk losing packets. An argument which requires other more sophisticated solutions elsewhere to play cool with TCP no matter if its responsive or not is nonsense. TCP should adopt to the current execution environment and not the other way around.

We should as earlier raise the stretch acknowledgement factor of the receiver to reduce the computation cost of the host to say 16 segments. TCP will, thus, struggle to keep a short queue locally to maintain the typically RTT one would expect between two local peers.

In other words, TCP does not scale well for a single sender with an extremely tiny base RTT and a tremendous bitrate.

3.4 Summary

We have now made it clear why TCP needs to scale further to yield better performance in a variety of network topologies. The minimum transmission rate of TCP prevents an AQM from doing its job correctly; the fault is TCP and not the network. The use of higher-bitrate is of little help as the resource demand the protocol sets to the network is without any grounds; the requirement for bitrate to grow proportionally as either RTT shrinks, higher MTU, more competition at the bottleneck or less frequent response from the receiver makes up a bad recipe for a scalable congestion control algorithm. The AQM is only able to deliver a shallow queue to its traffic if **all** senders promises to scale well below the two segments in an RTT threshold. Otherwise, the existence of a single unresponsive sender at the bottleneck is enough to neglect any benefits a shallow queue AQM was designed to deliver. The AQM can, of course, defend against

such unresponsive senders by evicting packets from such flows, but if the majority of traffic stands for the unresponsive traffic is high loss rate unavoidable. We would then loose out all the benefits of ECN and instead be controlling TCP by just using the exponentially growing RTO backoff mechanism.

We will now take a step back and look at how TCPs congestion control has evolved in the past for other researches around the precedence of the problem. The earlier work should help us in better understanding the problem and know if the problem is already solved or not. We are also interested in knowing if we can reuse some of the earlier work to work out a scalable solution ourself if there is no appropriate solution available from the past.

Chapter 4

Related Work

We summarise some of the earlier work found in the literature about the problem and others contribution to solve the problem. This work mostly talks about the problem as a consequence of a magnitude of flow sharing at the bottleneck. Our quantification chapter has already shown why such a requirement is not necessarily needed for the problem to exist. A small propagation delay in the network achieves the same effect. The previous work is still relevant as TCP suffers from the same symptoms and breaks as quickly as the rate of the flow falls below the stretch acknowledgement factor. A solution to this problem serves as an opening for TCP to scale for a variety of network topologies.

We differentiate between the sub-packet, and sub-mss regime as follows; A flow is said to be in the sub-mss regime when the flow has to send fewer segments than the stretch acknowledgement factor, usually two segments, per RTT. However, a flow is first in the sub-packet regime when the capacity is less than one segment per RTT. There is no real reason for this segregation as both regimes severely degrade the performance of TCP. The term sub-packet frequently occurs in the literature to talk about the problem. We, on the other hand, actively use the term sub-mss regime to generalise the problem for a broader definition which also includes the penalty of the delayed acknowledgement mechanism.

The sub-packet regime is a common weakness of TCP implementations since there is no easy way to let a flow hold a congestion window of less than one segment per RTT. TCP needs to transmit at least one segment per RTT to maintain its acknowledgement clock.

The sub-mss regime, on the other hand, occurs due to the delayed acknowledgement mechanism found at the receiver. Stretch acknowledgements make better use of the limited resources of the network. This regime is not so different from the sub-packet regime, but in this case, the selfish sender has a mechanism to clock out fewer segments but refuses such that the acknowledgement clock gives accurate RTT measurements.

The related work introduces two less-than-best-effort congestion controls algorithms; we briefly explain their motivation and how they are related to our work. The IETF posted a survey for this type of congestion control [69].

4.1 TCP - Many flows

Morris investigates TCP behaviour on a heavily congested network path in 1997 [49]. He ran two identical simulations in Network Simulator (ns) to test the scalability of TCP. The network consisted of a simple topology, where a fixed number of senders shared a bottleneck with an equal number of receivers on the opposite side. The AQM used for these simulations was RED in an attempt to keep the queue short for the whole duration of the experiment.

The first simulation had a few flows to validate that the congestion control algorithm of TCP did work as intended, and as we expected each flow got their equal share of the bandwidth and stabilised on a reasonable congestion window. The flow observed almost no packet loss and was thus able to recover from loss episode efficiently by initiating the combination of fast retransmit and fast recovery algorithm.

Meanwhile, the second simulation was set up with an excessive number of flows over the very same configuration. The meaning of excessive here does not necessarily mean more than the usual number of flow observed over the Internet. Morris did analyse packet traces of three type of Internet links and observed that some of these links had this level of flow through the bottleneck. The second simulation had the same level of throughput through the link, but the achieved bitrate of each flow severely varied throughout the simulation. Each flow observed a higher packet loss rate resulting in frequent silent periods followed by RTO. An RTO event would push the flow back exponentially and eventually resume its transmission of one segment per RTT. A flow would either be in good standing and maintain a large congestion window or be unlucky and get repeating occurrence of timeout.

A temporary solution was proposed to increase the buffer size as more flows competed. The bottleneck would then have enough buffer storage to let each flow send at their minimum congestion window of one segment per RTT. This solution made a certain number of flows share the bitrate equitably, but this at the cost of extra queueing delay in the network. The extra queueing delay inflates the RTT of all traffic to higher levels. The short-sighted solution does not scale well either since the size the buffer has to grow proportionally as more flows go through the bottleneck and thus resulting in even higher RTTs. Morris next solution of using RED, or any other more sophisticated AQM for that matter, does not overcome the problem either as we have already shown in our quantification chapter. As the RTT of TCP becomes shallow will any effort of the AQM to keep a shallow queue be of little help, and the contribution of each flow would make the job of AQM undoubtedly impossible.

Morris did, however, suggest an optimal solution which was to make TCP less aggressive by not rapidly increasing the low congestion window. He also suggested taking advantage of rate control for low congestion window values over traditional acknowledgement clock approach. Morris also claimed that the exponential backoff mechanism of TCP yields a terrible receipt for a scalable congestion control algorithm.

4.2 TCP - Adaptive RED & SUBTCP

Feng et al. investigates the overall loss rate of TCP in not so different network topologies using the Network Simulator (ns) [24]. Their study takes place in the same year as Morris first discovered the problem. They benchmark the performance of TCP by doing an in-depth analysis in revealing some severe flaws in the current way of doing congestion control, which substantially contributes to a loss rate of extreme levels. Their finding shows that even with the use of RED, the suggestion of Morris does not overcome the high loss rate from happening during times of congestion. In their study, RED falls short in pushing back enough sources to hinder buffer overflow, while in some other setups it becomes too aggressive and thus periodically let the link go unused[23]. The other part of their work is more relevant for us as they look at how TCP sources contribute at keeping high loss rates in networks where a correctly equipped AQM malfunction.

4.2.1 Adaptive RED

They first look into what role the number of TCP sources plays in the early detection of RED and do so by disabling the max_{th} length parameter of RED, i.e. max_{th} equals the maximum buffer capacity. It is clear from their result that RED is highly sensitive to the parameter selected for the experiment. They are either able to optimise RED for a small or moderate number of flows, but not for both at the same time with one unique configuration. Feng et al. find that to keep a link utilised for a small set of senders the max_p parameter of RED has also to stay low while under heavy periods of congestion the max_p parameter has to be kept sustainable high to prevent packet loss due to a buffer overflow. They repeat the same experiments with max_{th} set lower than the buffer size and still see packet drops due to a buffer overflow. It is first when they add a very deep buffer that they can eliminate packet drops, but RED keeps malfunctioning as it is neither able to keep the average queue it was configured for and often let the link go underutilised. Meaning, that they saw the need for a more intelligent AQM which would react more aggressive in the presence of a lot of capacity-seeking sources and an otherwise more conservative response from the AQM.

They propose an adaptive AQM, which they call Adaptive RED[1], to circumvent issues related to changing network topology. Their approach was to adjust the max_p parameter dynamically such that the average queue stays bound within the min_{th} and max_{th} parameter of RED. Adaptive RED increases the max_p parameter once by α if the average queue was shorter than the min_{th} parameter, and if the average was higher than the max_th parameter the max_p parameter was also decreased once by β. The max_p

[1]Floyd et al., the primary author of RED, later continued this work, and he calls this version also Adaptive RED [31]. The Adaptive RED algorithm was further improved and eventually made its way to lots of researchers. The main idea stays the same, but we have left out details of the new algorithm as they are irrelevant for the oncoming explanation.

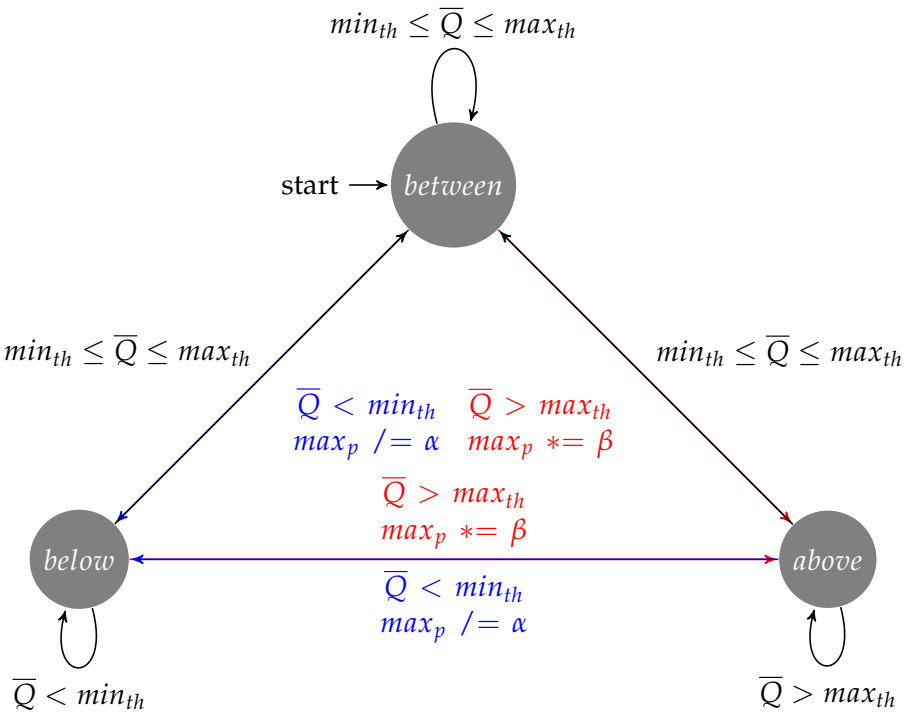


Figure 4.1: Adaptive RED

parameter was, otherwise, kept as is until the average queue crossed the other boundary (see state machine in Figure 4.1).

4.2.2 SUBTCP

They additionally investigate the performance of the end-to-end congestion control of TCP in scenarios where heavy congestion unfolds. Their work resumes the work of Morris and looks into the need of making TCP less aggressive in the sub-packet regime to maintain low loss rate through the bottleneck. Feng et al. tell that an AQM cannot alone guarantee a low loss rate without cooperating sending endpoints. They argue that for the typical Internet Service Provider (ISP) networks, this environment is quite common, and the current use of TCP may lead to high loss rates through their respective network.

They look at two major weakness of TCP, first the minimum transmission rate and then the linear increase probing. However, their study does not touch upon the impact of the delayed acknowledgement mechanism; they instead evaluate *against* TCP with a minimum congestion window of either one or two segments per RTT. They mention that the backoff mechanism of TCP falls short as RTO reinitiate upon one successful transmission and thus causing a very aggressive probing. The capacity is no longer shared evenly across the competitors of the bottleneck as some flow are stuck in repetitive silent periods, whereas other flows accelerate to inappropriate and unjustified transmission rates.

They propose a modified TCP, which they call SUBTCP, to overrule

the default action of TCP for when the congestion window is merely one segment and only let regular TCP to rule again when there is sufficient capacity available. First, they keep the backoff mechanism as is, that is whenever the sender detects a loss the congestion window cuts by half. Now instead of reprobing the network after successful transmission, they reduce the penalty of the sender by half. The results are indeed slightly improved, but they quickly understand that using Multiplicative Increase Multiplicative Decrease (MIMD) is not the way to go as it loses the convergence property of TCP and make it impossible to keep a stable queue. This problem motivates their next idea, which is to scale down the additive increase of TCP. They explain that the constant additive increase is troublesome when the congestion window is low as it causes the queue to fracture too much. So, they try to scale the additive increase down by a constant factor whenever TCP run out of congestion window and otherwise let TCP do its thing. They find that even by scaling down the additive increase; they are stuck with an additive increase phase which does not adapt well to change in congestion. As a last resort, they try to let the additive increase phase depend on the available capacity and probe a certain percentage over the offered bitrate. Meaning, the additive increase phase becomes less aggressive for when the congestion window is low and vice versa. They also use this method to probe for capacity in all environments of TCP.

By looking at their results, they are indeed able to scale TCP for low BDP paths with excellent results. They talk about continuing their work by making TCP even less aggressive to maintain a more stable queue and also look more into making TCP converge faster. At the moment, their proposed solution makes a flow with a low rate to use a significant time to back off other competitors keeping a higher rate, i.e. slow convergence.

4.3 TCP - The Initial Work on ECN

Floyd et al. wrote a short validation paper on ECN[30]. This paper was later used to justify the appropriate response from the sending host in the standardisation of ECN[59, 60]. They look at a variety of scenarios using ECN which they verify through experimentations.

The experiment of most interest talks about TCP equipped with ECN over a low BDP network path. They consider a sender holding a congestion window of only one segment per RTT. They explain that TCP with ECN must still back off when receiving marking when holding a congestion window of only one segment. They describe that instead of doing transmission based on the principle of acknowledgement clock, the sender must only use the retransmit timer for further transmission. The sender continues to grow the retransmit timer exponentially for any successive marking and do so until a segment arrives without the ECN-Echo (ECE) bit set. Their approach fakes a loss episode locally at the sender and effectively back off the sender indefinitely without needing to modify the congestion window any further. There is a minor flaw in depending on an exponential

growing transmission then reprobe upon a single successful transmission. Such sender grows its congestion window faster than a slow start probing and comes at a high cost of overshooting the capacity of the AQM for shallow thresholds.

They also briefly talk about another approach where instead of using the RTO to back off, the sender could shrink the size of the packet to achieve a similar effect. However, they seemed this to be unsafe because this would let TCP with ECN to accelerate faster than the standard TCP and would result in an uneven share of the capacity.

4.3.1 Linux TCP with ECN - Criticism

The Linux way of interpreting the work of Floyd et al. and the corresponding standard document remains questionable. The Linux sender always reset its retransmit timer on the reception of an acknowledgement and clock out the next segment without performing any backoff for when the congestion window is two segments [2]. Reason for it being two segments and not one has directly to do with the penalty of the delayed acknowledgement at the receiver. The Linux sender chooses to become unresponsive when the proposed standard tells otherwise is missing a justification. A solution where the sender clocks out two segments back-to-back using an exponential growing retransmit timer would surely have given a responsive sender indefinitely. Scalability of such solution might become a concern, but not reacting upon congestion is *always* wrong.

The choice of merely ignoring the concern yield degraded performance from the state-of-the-art Linux implementation, a misbehaving sender will struggle to yield any excellent performance. The poor performance of Linux could derive from the evolution of the standard ECN document of TCP. In the first version[59], which is now obsolete, the requirement to backoff while holding a low congestion window was optional but recommended and quantified. The specific part of the document which might light up their reasoning read as follow:

> *After the source TCP reduces its congestion window in response to a CE packet, incoming acknowledgements that continue to arrive can "clock out" outgoing packets as allowed by the reduced congestion window. If the congestion window consists of only one MSS (maximum segment size), and the sending TCP receives an ECN-Echo ACK packet, then the sending TCP should in principle still reduce its congestion window in half. However, the value of the congestion window is bounded below by a value of one MSS. If the sending TCP were to continue to send, using a congestion window of 1 MSS, this results in the transmission of one packet per round-trip time. We believe it is desirable to still reduce the sending rate of the TCP sender even further, on receipt of an ECN-Echo packet when the congestion window is one. We use the retransmit timer as a means to reduce*

[2] The Linux implementation while cleaning up its retransmission queue in acknowledgement processing sets a flag. The flag is later checked to rearm the retransmit timer.

the rate further in this circumstance. Therefore, the sending TCP should also reset the retransmit timer on receiving the ECN-Echo packet when the congestion window is one. The sending TCP will then be able to send a new packet when the retransmit timer expires.

The document is unambiguous in explaining the problem and why the sender should back off when receiving successive marked segments while holding a congestion window of merely one segment. The implementer is made aware of a possible scenario where the sender may become unresponsive, but the lack of a more accurate wording allows the decision to be up to the individual implementation. They turned this work later into a mandatory requirement for all sending host and gave the details in the current version of the document[60]. The previous paragraph now reads:

After the source TCP reduces its congestion window in response to a CE packet, incoming acknowledgements that continue to arrive can "clock out" outgoing packets as allowed by the reduced congestion window. If the congestion window consists of only one MSS (maximum segment size), and the sending TCP receives an ECN-Echo ACK packet, then the sending TCP should in principle still reduce its congestion window in half. However, the value of the congestion window is bounded below by a value of one MSS. If the sending TCP were to continue to send, using a congestion window of 1 MSS, this results in the transmission of one packet per round-trip time. It is necessary to still reduce the sending rate of the TCP sender even further, on receipt of an ECN-Echo packet when the congestion window is one. We use the retransmit timer as a means of reducing the rate further in this circumstance. Therefore, the sending TCP MUST reset the retransmit timer on receiving the ECN-Echo packet when the congestion window is one. The sending TCP will then be able to send a new packet only when the retransmit timer expires.

Note the use of "MUST". The last two sentences now mandate the transmission to happen **only** through the retransmit timer when holding a congestion window of one and receiving marked segments.

The latest congestion control standard of TCP does not either put any light on their reasoning. The congestion control standard reinforces the message of the new paragraph above by citing the up to date standard document.

4.4 TCP - Extensive Testing with Many Flows

Qiu et al. profile TCP under heavy load to see how well the protocol scale in a variety of network topologies[58]. Additionally, they investigate what effect a varying propagation delay has on the performance of TCP. In their analysis and simulations, they evaluate the aggregated throughput, goodput and the loss rate of TCP in a simple network topology consisting of just one bottleneck between the sending and receiving hosts. They test

the scalability of TCP under three set of bottleneck scenarios: a small, medium and massive buffered bottleneck. They evaluate the performance of TCP without the support of ECN.

They start by using an identical propagation delay for all the sending hosts through a simple drop-tail bottleneck; they quickly observe a long-lasting synchronisation among the competing flows. Flows repeatedly departure and arrive within a short period, which leads to periods with underutilisation of the link once multiple senders recover from loss simultaneously, and synchronised losses once the buffer occupancy reaches its maximum length. They try to either add random processing time to the sending host or introduce a RED gateway to minimise the risk of global synchronisation among the competitors.

The experiment of most interest runs on a small buffered bottleneck. Qiu et al. find that TCP yields terrible performance leading to an uneven share of the capacity and periodically silent periods among the competing flows. By adding random processing at the sending host, they can break up the synchronisation and tells that one need to add at least 10% of the RTT worth of processing to break up the synchronisation. The use of random processing delay ends up weakening the performance of TCP by introducing higher loss rates and less even share of the capacity.

Finally, in their last resort at improving the results they use RED without ECN to mitigate the issues regarding synchronisation. Under RED, they no longer see any synchronisation between the flows. However, the capacity sharing remains uneven when using a shallow queue AQM, and neither is the buffer occupancy of RED any stable. They can achieve better performance with RED on the other schemes, but this is not relevant for our work as we mainly focus on an overall shallow base RTT, which implies a shallow queue AQM too.

Lastly, their results indicate that RED can reduce an unfair bias between two greedy competitors with different base RTT. TCP performs poorly with different RTT; a flow with a lower base RTT gets an advantage over other competitors. A flows with lower base RTT achieve a much higher portion of the capacity.

Their work does indeed confirms the earlier work that the culprit is TCP which either stays stuck in an exponential growing retransmit timer or accelerate above its appropriate transmission rate. The network or the AQM has surely no blame to take in this matter. An indefinitely responsive sender which uses the retransmit timer to back off does not yield any significant results.

4.5 TCP - Delay Control

4.5.1 Receiver-based Delay Control (RDC)

Hsiao et al. investigate the performance of TCP over the use case of unicast layered video streaming[35, 36]. Applications doing video streaming are susceptible to variation in both delay and throughput. The standard TCP

is not well fit for these applications since the rate halving behaviour of TCP introduces massive jitter in both the achieved latency and bitrate over small periods. Nonetheless, this problem is precisely the motivation behind TCP Friendly Rate Control (TFRC) [27, 32]. Hsiao et al. propose a quite simple Receiver-based Delay Control (RDC) for smoother congestion control. They do not modify the existing congestion control mechanism of the sender, but instead, extend the receiver to help with congestion control in the reverse path of the origin of the greedy traffic (from the sender to the receiver).

Their approach is to make the receiver add a calculated penalty to the processing procedure of a stretch acknowledgement. The receiver makes the RTT seen by the sender longer in the event of congestion and effectively reduce the maintained rate of the sender and even when the congestion window is held constant by an unresponsive TCP. They look at two different methods to calculate the penalty. First, they employ a simple token bucket mechanism at the router to insert tokens as part of the queueing process of the router. They then drain the token bucket at a lower rate than the actual output rate of the link, the selection of the drain ratio decide the balanced equation between the link utilisation and buffer occupancy. The lower the drain ratio, the higher the buffer occupancy and vice versa, but now the link periodically underflows. The information, the token, obtained from the token bucket gets forwarded to the receiver, so the piggybacking acknowledgement gets delayed with the appropriate time unit. However, this requires additional information in the TCP header and more functionality at each network element, so they found a better solution which was to measure the extent of congestion from the marking level observed at the receiver. A low percentage of marking resulted in an additive **decrease** of the delay, and whenever the marking level became significant, above a certain threshold, the delayed doubled. The receiver measures the level of congestion for every RTT of the sender, but since the receiver lack this information it has to be approximated. Additionally, to additive decrease the delay, they rely on an approximation of the congestion window of the sender as well. The receiver is, thus, smoothing out the probing of TCP by working against the congestion control algorithm of the sender.

They were able to achieve great results with RDC; better fair-sharing of the capacity and overall reduced latency. The queue was kept low by keeping the sender responsive even during an extreme level of competition at the bottleneck. RDC has no issues in competing fair with standard TCP since the sender was not modified.

4.5.2 Sender-based Delay Control (SDC)

Hsiao et al. continue their work, but this time, they tackle the problem from the sending host[44, 52]. Their reasoning for not doing congestion control from the receiver is because both the congestion window and RTT has to be approximated, which makes the solution hard to scale. So, they propose a Sender-based Delay Control (SDC) algorithm to induce a delay during

transmission of segments.

The sender no longer uses congestion window purely to back off but additionally rely on SDC to become responsive in environments with extreme competition. They split up the congestion control of TCP into two modes. Regular TCP perform the congestion control when there is sufficient capacity available and otherwise let the SDC algorithm take over. They define a low congestion window to be when the sender has to send fewer than eight segments per RTT. Reason for them to choose such a high threshold is to maintain the efficiency of both the fast retransmit and fast recovery algorithm and avoid the cost of the RTO. As the success of an efficient loss recovery phase severely degrades when fewer segments are inflight per RTT. The sender needs enough segments inflight to trigger three duplicate acknowledgements from the receiver in addition to those segments which were lost by the network.

They explain that to bring down the packet rate, W_s/RTT, of the sender is only possible by expanding the RTT. Nonetheless, a packet rate of $8/(0.001 + 0.001)$ is the same as $4/0.001$. The former sender reduces its packet rate by inflating its RTT and keeping its congestion window constant. While making the sender add a local delay to its transmission may sound counter-intuitive do they argue that this is better for the network. A bottleneck which employs an AQM is then able to keep a minimal queue because all senders stay responsive. A sender should instead do a deferred transmission through a shallow queue and only add a small delay that corresponds to $1/N$ of the delay a deep buffered bottleneck would have needed, where N is the number of competitors at the bottleneck. They also find that by extending the RTT of the individual sender, one also minimise another common problem of TCP, which is an uneven share of capacity between senders with different base RTT. By, emitting more congestion signals to a sender with low base RTT, the AQM can help in levelling off the difference among the competing senders.

SDC approximately increases the transmission rate as fast as the traditional TCP congestion control to stay fair. They also evaluate the impact of reducing the added delay faster and slower than TCP. More importantly, SDC offers to go slower when performing delayed transmission. They evaluate SDC against regular TCP equipped with ECN and observe an overall performance boost from SDC. TCP is fragile and goes more often silence because of the RTO, while SDC needs to be less often rescued by the RTO.

Their work has the potential to scale on the Internet because it generally requires minimal modification to the sender. The receiver remains as is and the network is also just required to support ECN. However, they do not look into the challenges of the delayed acknowledgement mechanism. Stretch acknowledgements are much prefered as they keep the resource demand of the network contained.

4.6 TCP Nice - A Background Transfer Protocol

TCP Nice is a congestion control algorithm designed for background traffic. The goal of TCP Nice is to use the available spare capacity in the network. The congestion control algorithm base itself on TCP Vegas RTTs measurements approach and signal congestion for higher RTT measurements. TCP Nice has a more conservative approach and is more sensitive to congestion than TCP Vegas. TCP Nice does this by detecting the congestion early and reduce its transmission rate more aggressively. TCP Nice is also one of the few congestion control algorithms that permits a congestion window below one segment per RTT. The developers of TCP Nice, Venkataramani et al.[68], conducted experiments in Network Simulator (ns) to confirm that their congestion control algorithm was less aggressive than the standard TCP NewReno and TCP Vegas. TCP Nice was designed not to interfere with foreground traffic even if it meant that some of the available spare capacity did go unused.

The idea behind background traffic is to improve the overall Quality of Service (QoS) for non-critical tasks in order to give a smoother experience to the user. QoS goals for a service, which runs in the foreground, could be better availability, reliability or latency. The foreground flow then saves time when the service later needs data, which has already been prefetched by the background traffic.

The idea behind TCP Nice was to detect congestion early and avoid interfering with the foreground flows at all cost. TCP Nice uses RTT samples to detect the queue in the process of building at the bottleneck and thus back off early, so no foreground flow notice its appearance. TCP Nice maintains minRTT and $maxRTT$ over time just like TCP Vegas. The congestion algorithm uses $minRTT$ to detect an empty queue at the bottleneck. Meanwhile, the $maxRTT$ is used to find the maximum capacity of the queue at the bottleneck. Initially, the $maxRTT$ is the very first $minRTT$ times two. These measurements are used by TCP Nice to understand the queue dynamics and make intelligent decisions on when to emit more into the network and when to yield for other traffic.

TCP Nice back off when a fraction (f), usually a half window, of packets encounter congestion in an RTT. A packet is said to be causing congestion when the packet is measured to have RTT of more than the base $minRTT$ plus an offset. TCP Nice indicate congestion as follow (Pseudocode from the paper[68]):

//Per ACK operation
if $(curRTT > (1-t) * minRTT + t * maxRTT)$
    numCong++;

//Per RTT operation
if $(numCong > f * W_s)$
    $W_s = W_s/2$;
else
    //TCP Vegas congestion avoidance

The threshold (t) controls the sensitivity which determines if the current RTT measurement is an indication of congestion. The developers of TCP Nice selected a threshold of 0.2, which they argue gives a great queue dynamic at the bottleneck. TCP Nice achieve a good chunk of the spare capacity without being too aggressive against other critical flows. TCP Nice reduce its transmission rate similar to the conventional TCP, which is to reduce the transmission rate to half of the original congestion window.

TCP Nice use the same additive increase approach as TCP Vegas to probe for additional capacity at the bottleneck. TCP Nice uses the TCP Vegas congestion avoidance algorithm until enough segments in the current window have resulted in a congestion signal. The congestion avoidance phase of TCP Vegas goes as follow (Pseudocode from the paper [68]):

$E \leftarrow \frac{W_s}{minRTT}$ //Expected throughput

$A \leftarrow \frac{W_s}{curRTT}$ //Actual throughput

$Diff \leftarrow (E - A) * minRTT$

if $(Diff < \alpha)$
    $W_s \leftarrow W_s + 1$
else if $(Diff > \beta)$
    $W_s \leftarrow W_s - 1$

TCP Vegas use $\alpha = 1$ (minimum) and $\beta = 3$ (maximum) to indicate how many of its packets should remain queued at the bottleneck. TCP Vegas in comparison to other congestion control algorithms does not make

a multiplicative reduction of its congestion window. TCP Vegas uses linear increase and linear decrease to keep the number of packets enqueued between α and β for the currently observed RTT.

TCP Nice implements congestion window sizes below one segment per RTT by deferring transmission of new segments for a given amount of smoothed RTTs. TCP Nice use a timer on the socket to clock out two successive packets (burst) as the congestion window of the flow drops below two segments. TCP Nice uses transmission of two segments to counter a receiver which plans to delay acknowledgement of the first segment. The deferring process causes the flow to lose its acknowledgement clock temporarily, but the transmission is still in full control of the sender. TCP Nice behave in a more conservative behaviour than its counterpart of other congestion control algorithms and effectively maintain a fixed number of bytes per RTT. This behaviour should conceptually not be an issue as the flow goes slower than the acknowledgement clock permits. TCP Nice has an artificial floor for the congestion window of one segment every 48 RTT. Meaning, the scalability of TCP Nice has limitation to the transmission of two successive segments every 96 smoothed RTTs.

TCP Nice is impractical for our use case due to the reasons we list here. First, TCP Nice only look attractive for background traffic due to its less than best-effort mechanism. We, on the other hand, wants to solve the problem for a broader set of congestion control algorithms typically used over the Internet. Secondly, TCP Nice solution of transmitting at least two successive segments is not a great solution to the challenging delayed acknowledgement mechanism. A significant stretch acknowledgement factor makes this approach infeasible since TCP Nice would then have to transmit more successive segments at once. Although, the developer of TCP Nice was at that time aware of this weakness and planned to improve this by pacing packets out evenly. Their work, therefore, validates our problem statement as we are not the only one that requires a scalable solution to the delayed acknowledgement mechanism for lower congestion window values. Finally, the implementation of TCP Nice is outdated and requires a tremendous effort to work on a newer version of the Linux kernel.

4.7 TCP - The Sub-packet Regime

4.7.1 Sub-packet Regime

Chen et al. [17] describes and validates an analytical model to examine the stationary distribution for a given number of flows in the sub-packet regime. They explain that the sub-packet regime paths are common in developing countries. Their approach is to use a simple variant of the full Markov model[3] to understand the backoff behaviour of TCP in the sub-

[3] A Markov model capture states and transitions of a generally random system. Markov is a memoryless probability model, so it has no history of the previous states. This property

packet regime. The model constructs around the RTO of TCP which they explain is a widespread phenomenon during the sub-packet regime.

They validate our concerns that TCP breaks under the sub-packet regime and yield terrible performance as a result. TCP no longer share out the capacity evenly and suffer high packet loss. The regime causes TCP to suffer from repetitive occurrence of RTO and thus growing periods of silence. A sender at the bottleneck either probe faster than its competitors or is held back by the exponential growing RTO of TCP.

The model is meant to be used by network elements to optimise for the TCP traffic. The assumptions behind the model are that there will be lots of packet loss events per flow, and each flow operates in the sub-packet regime with small congestion window sizes. Finally, the last assumption they make is to model packet loss as one single loss parameter to the model. They validate the prediction of the model through ns2 simulations for a variety of bottleneck scenarios.

4.7.2 Time Aware Queueing (TAQ)

Chen et al. employ their previous work by introducing Timeout Aware Queuing (TAQ) [18]. TAQ is designed around the requirement to minimise the level of RTO observed by TCP in sub-packet regimes. They do not change the standard TCP and instead implement a smart network element which models the current state of the congestion control of TCP.

They explain that it is tough to make a model of TCP generally and is even harder to predict a sender in the sub-packet regime without the knowledge of all the previous states. TAQ, therefore, keeps per-flow state information of each flows at the bottleneck over a short period, approximated to be the RTT for a given flow. They use an idealised version of the model to predict the current state of each flow traversing the bottleneck. Using the model, they classify flows within a given state and makes the bottleneck aware of the congestion control of TCP. They use a multi-class priority queue schedule policy to perform a more aggressive action against flows that are probing faster than their fair share of the capacity and tries to keep timeout of each flow no more than its needed. TCP flows should then be able to observe an equal number of silence periods and thus shareout the capacity evenly.

They improve the sharing of capacity between flows in the sub-packet regime. TAQ allows a more significant number of senders through the bottleneck by inducing less often RTO to fragile flows. They, conclusively, say they will continue their investigation and try to find a more optimal solution from the end-to-end congestion control of TCP. A solution from the host itself is prefered as it let all the network elements of the Internet to stay stateless and not be required to predict the current state of each flow.

is crucial as it makes the model feasible to implement.

4.8 LEDBAT - Sub-packet Regime

Low Extra Delay Background Transport (LEDBAT)[64] is a less than best effort congestion control algorithm developed by BitTorrent. The goal of such scavenger transport is to take the spare capacity found in the network. A LEDBAT flow does not probe as aggressively as TCP, and neither does LEDBAT wants to disturb a TCP flow. LEDBAT actively measures the one-way delay and signals congestion after the flow has built a small queue at the bottleneck. A LEDBAT flow retreats as a TCP competitor probe for capacity. A scavenger transport method is well suited for an extensive background bulk-transfer, as in pair-to-pair file-sharing networks, since such transfer does not induce the same level of congestion as traditionally congestion control algorithms.

Komnios et al. [43] investigate the performance of the scavenger transport protocol LEDBAT/fLEDBAT vs the conventional TCP in an emerging region type network setting where the sub-packet regime is more likely to occur. The sub-packet regime, as we know, happens as the throughput of the flow goes below the one segment per RTT threshold. Their findings show that LEDBAT acts more aggressively than TCP in the sub-packet regime.

TCP, as we know, gets stuck in an exponential growth period of silence. TCP flows at the bottleneck, therefore, fails to contribute with enough queue in the sub-packet regime. LEDBAT flows are misled to think they can continue probing for more capacity when the situation is the opposite. They tell that AQM is not the answer to the problem. They plan to extend LEDBAT machinery with a new mechanism to preserve the goal of scavenger transport in the sub-packet regime. As we see, other protocols like LEDBAT has to be more complicated in the sub-packet regime just because of the scalability issues of TCP. By scaling TCP properly, other protocols can be kept simple and not be required to cooperate with a non-scalable TCP.

4.9 DCTCP - Packet Slicing

Huang et al. investigate the performance of DCTCP in a data centre environment in recent years, the period 2015-2018 [37, 38]. They argue that DCTCP starts to struggle in data centres because of the presence of a magnitude of concurrent flows and a low propagation delay between the processing nodes. Their focus is on application load which needs many concurrent flows traversing to one single processing node, e.g. the design pattern aggregate/partition. They find that merely reducing the congestion window is not enough to avoid the incast problem, which results in periodically buffer overflow and underutilisation of the link. They use ECN, but the shallow buffered switches typically found in data centres run quite fast out of the buffer. The outcome they see is frequent loss periods, which is severe enough to cause loss of the whole window of segments. The efficient algorithms such as fast retransmit and fast recovery

fall short, and cause uneven silent periods across the set of flows (RTO). They see that the solution has to come from DCTCP and not the network, as the typical switches in data centres use shared memory across all ports and the use of more expensive hardware is out of the question. Although, the use of expensive switches is in no way going to solve the problem if the concurrency origin from one port.

They propose dynamically adjusting the size of the segment, in addition to the congestion window, as a means to reduce the transmission rate of the sender. To choose the optimal size of the segment, they introduce packet slicing, which relies on Internet Control Message Protocol (ICMP) signalling to propagate the appropriate size to all the traversing flows. They argue that by using packet slicing, they can keep the packet overhead as low as possible in even the presence of extreme congestion levels, through the use hardware optimations to reduce the utilisation of the CPU. Another problem they face is bursty arrival of flows because of synchronised senders, the likelihood of synchronised sawtooth increases as more flows competes. They use the pacing facility of the Linux kernel to spread out the bursty arrival and also look into adding random delay to the transmission process of the sender.

In the end, they can achieve better results than pure DCTCP. They claim to have solved the incast problem of TCP by evaluating their results in both network simulator (ns) and a real small-testbed. In their evaluation, they mainly use the number of concurrent flows and the goodput achieved to determine the success of their approach.

The switch, in their system, needs to do the packet slicing periodically which add complexity since the switch must maintain the state of each flow and requires a firmware update.

4.10 DCTCP - ExpressPass

Cho et al. look at some scalability issues of DCTCP in data centre environment[19]. Their work is quite recent and dates only back to late 2017. They examine the performance of DCTCP under shallow RTT with a moderate level of traffic over link speed as high as 10/100 Gbps. This setup is indeed the same network topology, high bitrate and shallow RTT, as we would like to benchmark DCTCP over.

Not surprisingly, they stun upon the same problem as the one motivating our work, which proves that this is, in fact, a problem out in the wild and is starting to gain momentum among researchers in data centre environments. They show that a moderate level of greedy senders is enough to reach this limitation of DCTCP even with a tremendous link speed as high as 100 Gbps.

They propose a new architecture called ExpressPass which bases on credit packet going in the reverse path of the origin of traffic, so from the receiver to the sender. The receiver uses the credit packets to govern the transmission rate of the sender. A credit packet does the congestion control of the sender; the sender is only allowed a new transmission if

the receiver has permitted such transfer through non-accumulative credits. The switches in the data centres then regulate the rate of the credit packet to maintain shallow queue in the reverse direction. Their approach is different from standard congestion control as it hinders the transmission of new segments, even *before* congestion takes place. The sender, usually, reacts upon a request to go slower after an RTT passes. Their system yield excellent performance in even the extreme scenarios, e.g. the sub-packet regime, since the sender has minimal control of the transmission and instead rely on the network on an individual packet basis. The scalability problem of DCTCP, or any other modern congestion control algorithm, is then avoided because of the passive congestion control at the sending node.

However, their promises of the architecture do not come free as it introduces some significant drawbacks. ExpressPass depends on asymmetric routing path between the two endpoints. The credit packets need to flow the same path as the traffic or else may the bottleneck in the reverse direction by a different node. ExpressPass is also not easy to deploy outside of data centres since they assume a minimal presence of other traffic. Simply segregating the new traffic from the legacy traffic is not enough to share out the capacity evenly with the existing traffic.

4.11 Summary

To summarise, there are lots of closely related work out there describing, quantifying and solving the problem we have in front of us. The sub-packet regime was located early on by Morris and was all about the presence of a high number of incast flows. The problem more or less lost its momentum for a decade because of the significant increase in bitrate around the globe, the rapid evolvement of the Internet. At the end of this period, the network elements of the Internet started to become bloated by the excessive buffer occupancy and masked away the negative impact of higher loss rate by ramping up the latency of the Internet. The problem is now making a comeback and becoming unavoidable because more factors are now pushing the higher bitrate to lower levels of fair share among competing flows.

The earlier work tends to either go around the problem using various techniques, involve assumptions which are not necessarily applicable to the Internet, degree of higher complexity or happens to be a not so straightforward solution. Nonetheless, there exist only a few earlier work which goes in the right direction and tries to solve the problem from the root. Feng et al. and Hsiao et al. do precisely that, but they only solve the problem partly as they lack a proper solution to the delayed acknowledgement mechanism.

The need for a solution is, of course, now starting to be taken seriously by the data centre community, but there is no scalable solution out there. A solution worked out by data centre vendors usually scope the fix to be constrained within a controlled environment, which is not the only problem we are solving here. Packet slicing tradeoffs the achieved goodput

to deliver a shallow latency to the application, which may not be an acceptable solution if the need for bitrate is also high. ExpressPass, on the other hand, does not solve the problem from the root and therefore add unwanted complexity and constraints.

Now that we have fully understood, quantified its precedence and seen others earlier attempt at solving the problem, we will give our methodology. In our methodology chapter, we ought to explain in detail how we plan to conduct our experiments.

Part II

The Project

Chapter 5

Methodology

We define our experiment plan in the first section of the methodology chapter. We plan to run a set of experiments in order to understand how TCP operates in networks with very thin RTTs. The primary goal of these experiments is to investigate how well TCP cooperate with a proper fine-tuned shallow queue AQM over an already low base RTT. We assume that TCP will become unresponsive by not going slower than two segments per RTT and keep overriding the AQM. We also believe that one single source of unresponsive flow is enough to cause serious trouble for an AQM which do not perform eviction policy.

We list all the metrics which we will use in the process of evaluating our experiments. One of the most vital metrics we look into is the queueing delay of the bottleneck. The queueing delay stands for the currently imposed delay through the AQM for all traffic. Another metric which closely relates to the queuing delay is the respective RTT of a flow that is currently traversing the bottleneck. We hope to see a higher queuing delay and RTT in our experiments when TCP flows in our network misbehaves.

The methodology chapter ends with the configuration of the testbed. We go for a testbed topology to test the scalability of TCP on a network path with a tremendous bitrate and a shallow RTT. We limit our testing to the Linux kernel implementation of two commonly understood congestion control algorithms, namely TCP Reno and DCTCP.

5.1 Experiment plan

We plan to run several experiments to get a deep insight into TCP's behaviour for our problem statement. We base our experiments on the calculation of the Bandwidth Delay Product (BDP).

BDP is, as we already know, the number of bits a set of sender combined needs to emit per RTT to fully utilise the link. Going lower than the BDP leaves the link underutilised when eventually the queue dries up. However, going any faster than the BDP builds a queue at the bottleneck. We place an AQM equipped with ECN in our network to keep the link fully utilised for the whole duration of the experiment with a shallow queue.

Our focus is on experiments with high bitrates and shallow base RTTs.

We know that we can still get a low BDP path with tremendous bitrates when the network is capable of delivering shallow base RTTs. We want to show that the conventional TCP refuses to slow down when dealing with such paths and builds an enormous queue in the process.

Our main goal is to show that the real issue lies in TCP's congestion control, which prevents the congestion window from falling below two segments per RTT. No matter how much effort an AQM makes to keep, a shallow queue is it of no use and becomes an impossible task when the transport protocol proactive refuses to cooperate. TCP pushes the overall RTT of the network to higher grounds and keep overriding the AQM in the process. The consequence is not only limited to other competing TCP flows but is a massive problem for any other delay critical traffic as the queue is a shared resource. TCP is greedy by the pure definition of the protocol and is eager to maximise the achievable throughput over a network path as long as the application permits. A single TCP flow is, therefore, enough to cause serious issues when it starts to malfunction and result in a higher constant contribution of queuing delay at the bottleneck. The delay critical flows, not a competitor of TCP, is caught up in all of this just because of the appearance of TCP traffic.

5.1.1 Traffic

The objective of our experiments is to show TCP refusal to cooperate with the AQM for low base RTTs. The problem we are trying to address is specific to TCP, and our experiment plan is, therefore, to run a set of TCP flows. In other words, we assume that the only traffic present in our experiment, and therefore in our network, is made of just TCP flows.

We also assume that TCP flows which run in our experiments will be SACK-capable and makes intelligent decisions in the loss recovery phase. Further, the TCP implementation will use the standard[11] which cover the extension for a high-performance network. The idea is to give TCP perfect environment with a decent network and all the latest improvements.

We may now select our congestion control algorithm to evaluate in our experiments. Modern TCP congestion control algorithms will not permit their ssthresh to fall below two segments, which then limits the congestion window to two segments per RTT. The reason for rounding up as we now are first due to the delayed acknowledgements mechanisms at the receiver, but we also now that the latest congestion control standard[4] do not demand these algorithms to fall below two segments per RTT. A congestion control algorithm will risk being outcompeted by other TCP flows if it allows its congestion window to fall below two segments per RTT. We are therefore not so strict in our selection of the congestion algorithm since every modern congestion control algorithm will quickly fail this criterion.

- **TCP Reno**. We do not need to choose a more complicated congestion control algorithm than the standard TCP Reno as if it refuses to slow down then neither will any other algorithm do. Going slower than

the standard TCP is surely not a requirement for any sane modern algorithm which intends to fight with TCP Reno.

- **DCTCP**. We also try DCTCP to see if a more modern congestion control algorithm is up for the challenge as it optimises for a low queuing delay AQM.

5.1.2 Active Queue Management (AQM)

We mainly use RED to enforce a low queuing delay in our network. There is no specific reason to use RED other than that it is well understood and heavily used over the Internet and in data centres. We do, however, need to use distinct configurations of RED for each of our congestion control algorithm. DCTCP only need a straightforward configuration of RED, whereas TCP Reno needs a more advanced configuration to smooth out the queue.

- **TCP Reno** requires a complicated AQM scheme to hold a low queueing delay. RED must work on average queue length and also have an allowance for bursts before making any marking. We also use RED in adaptive mode to not deal with the same problem as Feng et al. had. We follow the guideline given here [29] for our configurations.

- **DCTCP**, on the other hand, only requires a straightforward AQM scheme to produce a stable shallow queue. The AQM can be very simple since the complexity shifts to the congestion control algorithm. The AQM only needs one single threshold parameter and mark on instantaneous queue sizes without any form of smoothing or bursts [1].

$$K = max(Bandwidth/8 * min, 2 * MTU)$$

Where K is the capacity of the queue in bytes, and min is the desired queue length in seconds. We need to use a floor of 2*MTU bytes in queueing capacity to overcome the time needed to serialise a packet on slow links [1]. We do, however, want to test how a ramp setup of RED work out on DCTCP, but without any smoothing or bursts (RED instantaneous).

5.1.3 Concrete Plan

We list all our experiments in Table 6.1. This overview helps in understanding where we focus on with our experiments.

[1] https://tools.ietf.org/html/draft-ietf-tsvwg-aqm-dualq-coupled-08 (Work in progress)

Experiment (#)	1	2	3A	3B	4A	4B	4C	4D	4E	4F	4G	4H
Bitrate (Mbps)	200	200	200	200	600	600	600	600	600	600	600	600
Base RTT (us)	300	300	100	100	250	500	750	1000	250	500	750	1000
MTU (B)	1500	1500	1500	1500	1500	1500	1500	1500	9000	9000	9000	9000
SMSS (B)	1448	1448	1448	1448	1448	1448	1448	1448	8948	8948	8948	8948
Stretch ACKs (bool)	false	false	false	true	false	false	false	false	false	false	false	false
Flows (#)	1	2	8	8	16	16	16	16	16	16	16	16
Flow time (sec)	10	20	40	40	40	40	40	40	40	40	40	40
Flow pace (sec)	-	1	1	1	1	1	1	1	1	1	1	1
Target queue (us)	500	500	500	500	250	250	250	250	250	250	250	250
ECN (bool)	true	true	true	true	true	true	true	true	true	true	true	true
DCTCP g (weight)	1/16	1/16	1/16	1/16	1/16	1/16	1/16	1/16	1/16	1/16	1/16	1/16
Buffer capacity (ms)	200	200	200	200	200	200	200	200	200	200	200	200
RED Ramp Marking												
Min_{us}	500	500	500	500	250	250	250	250	250	250	250	250
Max_{us}	1500	1500	1500	1500	750	750	750	750	750	750	750	750
Max probability (%)	10	10	10	10	10	10	10	10	10	10	10	10
Burst (packets)	20	20	20	20	31	31	31	31	31	31	31	31
Min W_s	13.18	6.59	1.24	1.24	1.54	2.32	3.09	3.86	0.26	0.39	0.52	0.65
Max W_s	29.64	14.82	3.29	3.29	3.09	3.86	4.63	5.40	0.52	0.65	0.78	0.91
Adaptive	true	true	true	true	true	true	true	true	true	true	true	true
RED Step Marking												
K_{us}	500	500	500	500	250	250	250	250	250	250	250	250
W_s	13.18	6.59	1.24	1.24	1.54	2.32	3.09	3.86	0.26	0.39	0.52	0.65
RED Instantaneous Marking												
Min_{us}	500	500	500	500	250	250	250	250	250	250	250	250
Max_{us}	1000	1000	1000	1000	500	500	500	500	500	500	500	500
Min W_s	13.18	6.59	1.24	1.24	1.54	2.32	3.09	3.86	0.26	0.39	0.52	0.65
Max W_s	21.41	10.70	2.26	2.26	2.32	3.09	3.86	4.63	0.39	0.52	0.65	0.78

Table 5.1: Experiments

5.2 Metrics for the evaluation of TCP

We base our evaluation of TCP on previous works found in the literature [5, 26]. We select metrics to measure by looking at how frequent the metric appears in previous work and what value it brings in our evaluation. We log these measurements for the whole duration of the experiment. We can measure a variety of metrics since we have full control of every node in the network. We also measure metrics for every packet processed at each node and can, therefore, get a very accurate reading of each metric at any given time in our experiments.

We do, however, not address flow rate fairness in our work as it is a meaningless metric. The flow rate alone has shown to give a false premise of fairness. The flow rate fairness metric has repeatedly been used in the literature to claim fairness among congestion control algorithms. This metric alone has been used to measure the quality of congestion control algorithms by looking at how good one such algorithm share out the rate with another participating algorithm. The dynamics of the first standardised TCP is commonly used as a baseline to defend any new proposals. Briscoe wrote about this dilemma in: "Flow Rate Fairness: Dismantling a Religion" [13]. He said that the internet communities kept on using this broken fairness requirement to address resource allocation and accountability, which base on no inheritance in any previous work found in philosophy or social science. Briscoe also mentioned that flow rate fairness does not even address what it is originally designed to tackle. Flow rate fairness only looks at the current rate of one flow compared to all other competing flows and does, therefore, not look at one single entity connected to the Internet. A malicious user or a cheating program could easily create multiple TCP connections to get a more significant quantum of the capacity and thus setting other users in the network at a significant disadvantage. Briscoe meant that we should base the fairness mechanism on the cost we impose other users in the network and that the flow rate would then be the outcome from such a fairness metric.

In other words, we do not base our work on this broken ideology and neither claim that our proposal is compatible with one such metric. Our work tries to scale TCP for lower RTTs and is therefore not compatible with most modern congestion control algorithm in this region. Our goal is to achieve low queuing delays in the submss regime and not look at how good other algorithms competes with our proposal. Fairness is, therefore, not a priority in our work as we are building a new baseline for a new set of modern congestion algorithms. The definition of what fairness means for TCP, or in general networking, has yet to be fully defined and agreed on and requires far more work other than what we intend to address here.

5.2.1 Queuing Length & Link Utilisation

One of the most critical metrics measured in our experiments is the queuing delay and link utilisation at the bottleneck. Queuing delay is the time taken from a packet takes to enqueues at the egress interface of the host and

until it departures to the next hop in the network. A lack in queue means underutilisation of the link.

We measure the queueing delay at the AQM. An AQM is often placed between the endpoints in the network to shorten the length of the queue. The queue length is often easily configured, and the main goal of the AQM is to keep a stable queue of this length. The length of this queue decides how fast the TCP flow should go. A shorter queue means the TCP flow must send fewer bytes per RTT. We want to measure the length of the queue when TCP is asked to send less than two segments per RTT. Therefore, the goal is to test if TCP obeys the recommendation given by the AQM. As we already know, a shorter queue means lower delays in the network.

5.2.2 Smoothed Round Trip Time (SRTT)

We will, besides, measure the RTT for every TCP flow in our experiment. We should be able to see the impact of the queuing delay to the RTT. We also want to see how the delayed acknowledgement mechanism impacts this reading when losing up the acknowledgement clock.

5.2.3 Packet marking rate

We also intend to measure the effectiveness of the AQM by checking the impact of a higher queue has on the marking rate of RED.

5.2.4 Throughput

We also measure the throughput of the individual sender to see whether or not the competitors of the bottleneck shareout the capacity evenly.

5.3 The Framework

We use a testbed topology to conduct our experiments. The reason we go for a testbed setup is to better understand TCPs behaviour with a variety of real network delays. We want to demonstrate that the problem we are addressing still exist with actual propagation delays in the network. We ought to emphasise the importance of queuing delay contra all other delays, which may impact the RTT of TCP. Our primary goal would be to show that TCP refuses to comply with the AQM in our testbed for lower RTTs. A testbed also let us examine the real TCP implementation of the Linux kernel. There are also other pitfalls in using simulation-based or emulation-based testing as outlined in "On the effective evaluation of TCP" [5].

We have set up the testbed as illustrated in Figure 5.1 totalling of three hosts connected in a private network. This setup consists of two dedicated hosts directly connected which the sole purpose of conducting experiments. The client node receives the traffic of importance from the server node as a downstream. The bottleneck is, therefore, placed on the upstream of the server host. The AQM is not a standalone host in

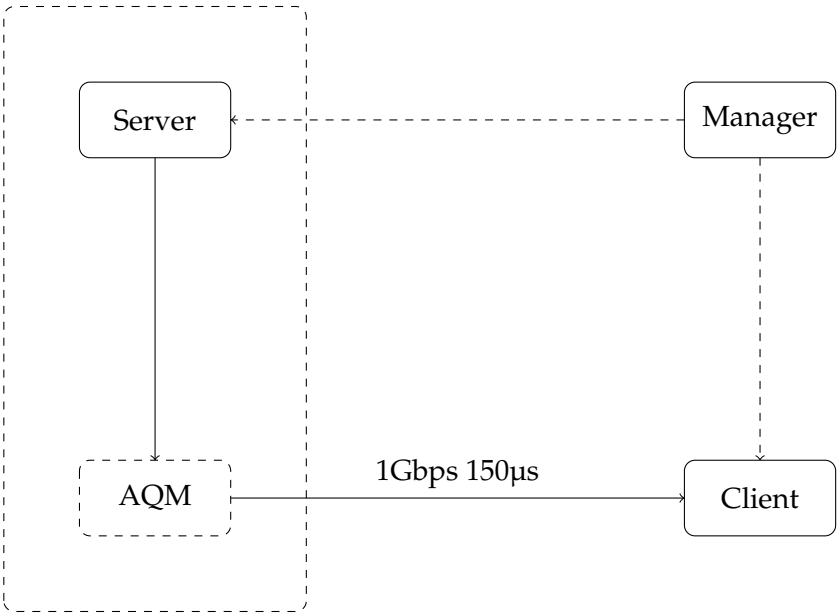


Figure 5.1: Experiment Setup

our network as the functionality of the AQM can easily be configured on the link between the server and client node as a queuing discipline. Our testbed replicates a typical network topology and is essentially how servers over the Internet, or a data centre, competes to deliver services to their clients. It is essential to keep in mind that we mainly care about the performance of the transport protocol over a data centre type network since the existence of the problem is more likely to occur there due to shallow RTTs. We believe our findings will assist researchers in better understanding the difficulties and limitations of the transport protocol in networks with very low RTTs.

The managing host is used as an administration host in our testbed to delegate out work to the server and client host. The managing host does not participate in the experiment as a processing node but is instead responsible for the experiment to complete within the given test parameters. The use of a managing node eases our need to reconfigure the processing nodes for each experiment manually. We believe this allows us to test the transport protocols behaviour with greater complexity and produce more reliable results. The manager issue commands on the respective host through WebSocket as a mean of communication. The manager then waits for confirmation before proceeding to the next command to keep our processing nodes syncronised throughout the experiment. The processing nodes in our testbed do not initiate experiments by themselves but instead wait for the administrative host. The processing nodes receive test parameters as soon as the manager chose to schedule a new experiment and then reconfigure themselves based on this configuration.

Server instances run inside of the server host and listen for their client. The server is, in other words, not multiplexed and close the "listen" socket

as soon as accepting the client. The server host is instead set up to run several instances of the server to accept an equal pair of clients. We can then use the unique port of the server as a unique identifier for a given flow in our analysis.

5.3.1 The Configuration of the Testbed

We have set up the testbed with one-gigabit link speed and configured each node to use ECN. The bitrate does not replicate the optimal conditions of data centres but is much higher than what most network paths typically offers in a broadband setting. We do this because of some resource constraints we currently have. We chose not to limit our scope of the problem statement as we know from the quantification chapter that more bitrate will not be able to solve queuing problems for data centres. We believe that our current chosen approach will be enough to reveal some otherwise ignored concerns and help cast some light on the problem in the research community.

We evaluate our experiments under the Linux kernel version 5.0.0. We place several levels of queueing disciplines on the egress interface of the server host to regulate the outbound traffic. We first let the propagation delay be a configurable test parameter in our experiments through Netem. Next, we add a Hierarchy Token Bucket (HTB) to limit the bitrate of the link artificially. It is crucial to limit the bitrate such that we measure the pure performance of the congestion control mechanism of TCP and not evaluate the quality of the driver of the NIC currently being used. The bitrate is a configurable parameter in our framework and is needed to test TCP under a variety of bitrates. Finally, we place the AQM as one final queueing discipline on the link to regulate the length of the queue in our network.

We launch a measuring tool at the server host to log metrics of senders as soon as a new experiment starts. We use trace events to logs congestion window. Trace events record metrics of our senders as new packets arrive from their receivers. Meaning, each acknowledgement from a client adds a new entry in our log for the corresponding server instance. We also run a similar packet monitoring tool[2] on the client node. This tool lets us extract various metrics that were added to the packet by the AQM. Meaning, we have measuring tool at both hosts to log a variety of metrics for all packets transmitted throughout our experiments.

We have designed our framework to permit flow with different congestion control algorithms within the same experiment. Although, our primary goal is not to make a detailed comparison of a variety of congestion control algorithms but to test some commonly used algorithms in a network with low RTTs.

The Maximum Transmission Unit (MTU) of the link is also a configurable parameter in our framework. A higher MTU enables us to test TCP in

[2]Written in C using PCAP API, this application uses techniques from here: https://github.com/henrist/aqmt

a data centre like a network where larger segment sizes are typically used to reduce the overhead caused by network packet headers. We want to compare the performance of the transport protocol with the typical frame size of 1500 bytes against the jumbo frame size (9KB). We expect the transport protocol to perform worse with jumbo frame sizes as the appropriate congestion window, in segments, for the same bitrate is significantly smaller (by a factor of 6). TCP would then be more vulnerable to the submss-regime for even higher RTTs and bitrates.

Hardware optimisation is often used by the operating system to offload as much work as possible to the Network Interface Card (NIC). The operating system can thus keep a more efficient system by saving CPU cycles. Higher gigabit-ethernet speeds become harder to achieve as the processing time in the operating system is too costly and the timeslot to process a single packet decline when moving to higher bitrates since a magnitude more packets has to be processed within the same period. Offloading work to the NIC usually give better results, but the success of such offloading may vary between different vendors and driver versions. Such optimisations are, therefore, often disabled by researchers before conducting experiments to not evaluate the effectiveness of their NIC at hand [45]. There also exist other non-hardware optimisation done by the operating system which might shew our results. The operating system also makes these optimisations with good intentions, but we must disable these when benchmarking TCP to reveal the actual performance of the transport protocol in an isolated setting. We have quite decent equipment at hand, so limitations due to turning off optimisation should not be a problem in our setup. Although, we keep a close eye on the utilisation of the CPU during our experiment to not get shewed results. We list each optimisation and explain what purpose they serve, and why we choose to disable them:

- **TCP metric saving** is used by the transport protocol to reinitiate new flows with information obtained from previous flows to the same destination. Information about a variety of state variables is reused to achieve this goal. The metric saving is essentially a cache mechanism which becomes invalidated after a long enough duration since the last update took place. We, therefore, disable metrics saving to not bias our new experiments from earlier conducted experiments. We also flush this cache periodically, so we get rid off any existing cache.

- **Segment offloading** is used by the operating system to postpone the process of splitting the user data into appropriately sized segments and vice versa. The NIC usually assists in making this process more efficient by reducing the number of packets that the operating system has to handle and thus saving CPU cycles. Segment offloading is typically disabled by researchers due to the reason is given above (unpredictable behaviour). We make no difference and turn off all form for segment offloading in both software and hardware. We as, other researchers, want to benchmark TCP as is and without any surprises that might come from buffering of segments. However, we

let TCP use Nagle's algorithm to reduce small transmission from non-greedy applications.

TCP Segment Offload (TSO), or Large Segment Offload (LSO), is a feature commonly found on NICs. TSO helps the operating system by offloading the procedure of partitioning data into appropriately sized segments. The segmentation process attaches the network header passed by the operating system to each now smaller sized segment. TSO plays a vital role for servers which aim to fully utilise higher bitrates with as little CPU utilisation as possible.

Generic Segmentation Offload (GSO) is a software-based segmentation which does not need support from the NIC and is also available to other protocols other than TCP in contrast to TSO. GSO tries to achieve the same goal as TSO by postponing the process of segmentation as much as possible to save CPU cycles.

Large Receive Offload (LRO) is the opposite of TSO and can be used to reassemble broken segments into a whole chunk of data and produce a single copy of the network headers. The operating system can thus reduce the number of packets that need attention and lower the overall load on the system at the client's side of the conversation. However, LRO is slightly aggressive in its merge process and could end up losing some vital information from the network headers. The merge process is, therefore, not lossless and could break the end-to-end principle if it were to be used by a network element in the path between the endpoints.

Generic Receive Offload (GRO) is as the counterpart GSO designed to improve efficiency in cases where the NIC does not support offloading and just like GSO it generalises to benefit all traffic. GRO effectively reassemble segments into a stream of data but has a stricter restriction for which segments to combine. GRO is, therefore, lossless and produce one single header for a set of joined segments and usually prefered over LRO. GRO is essentially the reverse process of what GSO accomplish.

- **Scatter-gather I/O** is an optimisation done by the NIC to read data from adjacent buffers located in memory efficient. A NIC that does not support scatter-gather I/O requires that the buffer passed to be in contiguous memory. Usually, the operating system has to assemble a linear buffer in the physical memory of the data passed from userspace and first, then instruct the NIC to read from this now duplicated memory region. Scatter-gather I/O, therefore, helps to reduce the need for otherwise expensive memory duplications. However, the efficiency of such vectored I/O may vary between different vendors and may not be available in all NICs. We, therefore, see the need to turn off this feature.

- **Interrupt coalescing** is a technique used by the operating system to defer interrupts from NICs with the hope that multiple events from

hardware are only seen as one single interrupt periodically. The operating system can thus avoid expensive computations and context switches, which comes part of the procedure in handling interrupts. However, moderation of interrupts does not come free per se as the hardware must refrain from generating an interrupt for an extended period and thus cause delays in responding. The hardware generates an interrupt first when a certain number of events enqueues or when a timer expires from the first event took place. These parameters are usually configurable by the operating system. Combining interrupts from devices might yield different behaviour across different vendors of NICs. We chose, therefore, to react upon each event from hardware as a single interrupt in our framework to produce easily reproducible results.

- **Pause frames** are part of the ethernet flow control which let an overwhelmed node in the network request halt of transmission of a sending node. The pause frame tells the sender to pause transmission for a given interval before resuming to avoid further congestion. However, the problem, oversubscription of a link, pause frames are designed to solve in our case resolved at a higher level in the hierarchy of the OSI-model, namely by the transport layer. Ethernet pause frames are also vulnerable to head-of-line blocking it can cause and are, therefore, most often not sent by an overwhelmed node but instead circumvented by implementing a virtual output queue. On the other hand, a sending node usually respects pause frames received in order not to overwhelm the receiver. We disable ethernet flow control in our framework to let TCP's congestion control algorithm deal with those issues. We can then measure the quality of TCP's congestion control algorithm and not rely on the link layer to provide such a mechanism for us. We are aware that priority-based flow control, a successor of pause frames for Data Centre Bridging (DCB) over lossless ethernet, exists and its use case. We chose to put this way to control congestion out-of-scope from our work due to its applicability to only a particular class of traffic and being unavailable to the majority of the traffic over the Internet.

5.4 Summary

We have listed our experiment plan and the various metrics we are going to evaluate. We have also made our simple setup transparent and the corresponding configuration.

We are now ready to design a better version of TCP which cooperates with the AQM to keep shallow queue under shallow base RTT.

Chapter 6

Design Proposal[1]

The main contribution of this thesis is given here as a design proposal which defines a scalable solution to the submss regime. Our goal with this design is to extend TCP's machinery to work more efficiently in networks with shallow RTTs. We are trying to solve the now uprising problem with long queueing delays caused by TCP because it refuses to lose up its acknowledgement clock. TCP has no grounds in keeping a long outstanding queue at the bottleneck when operating in low BDP networks.

We are, however, dealing with a hard problem as we need to change some of the fundamentals of how TCP currently operates. The protocol is poorly designed to scale in networks with low congestion window values [4, 60]. TCP also struggles to maintain a small congestion window due to its congestion avoidance phase of constant add of one segment per RTT. The constant additive increase of one segment per RTT is also the culprit, which makes the conventional TCP inefficient in enormous BDP network paths. We do not address fairness in our work, and neither has fairness been the basis behind this design document. Our main objective is to relook at how much better TCP can perform given the freedom of minimal backward compatibility. Our design introduces a new variant of a scalable congestion control algorithm in respect to TCP, and we are also exploring an otherwise ignored region that often the network gets the blame for lacking bitrate. We restrict to only propose minimalism modification in this design such that the current TCP implementations only require minor modification. The end goal of this design is, of course, to make up a scheme which helps to solve some of the current scalability issues of TCP.

[1] As mentioned in the introductory chapter, the work given here is with considerable help from the supervisor. The final design document is a product of continuous development cycles. The design proposal started as a rough idea and discovery of new problems and ideas updated it throughout its lifetime. The initial work was taken directly from the submss paper explaining and proposing a solution to the problem [14]. The supervisor has also helped in working out the design and contributed with ideas; this proposal reflects this work.

6.1 Fractional Congestion Window

We propose adding a fractional congestion window to TCP so it can operate efficiently in networks where the requested transmission rate must be low values on average due to shallow RTTs. We are primarily looking for a way to hold congestion window values in bytes so that TCP can keep less than one segment inflight per RTT. This requirement forces a significant modification to the protocol since the congestion window consists of whole segments, and most congestion control mechanism relies on this window to make decisions, e.g. the acknowledgement clock. We should, therefore, still keep a congestion window in segments and instead find a solution that works around these criteria.

We propose to add a separate state variable for the fractional part of the window. By doing this, we can keep the congestion window as is and maintain the fraction, values from 0 to SMSS bytes, in another variable.

$$W_B = W_s * SMSS_B + W_{frac}$$
$$S_B = S_s * SMSS_B + S_{frac}$$

Of course, this adds complexity to the protocol as we now have to update W_s whenever the fractional part surpasses the SMSS threshold and also update the fractional part with the reminder. However, this should only be a minor annoyance, whereas doing the opposite is more complicated. We would then have to convert the congestion window all the time into a unit of segments. This approach is cleaner and helps in keeping the congestion window segregated into two distinct part, one in segments and another one in bytes. No form for conversation is done, except when we want to know the exact value of the congestion window [2].

6.2 Packet Conservation Clock

We now need to look at how we can use the new congestion window, given in bytes, to transmit packets. We need to extend TCP to be able to use a congestion window of less than SMSS bytes per RTT.

A naive solution to this problem could be to send a packet of size W_B per RTT. The sender then goes slower, but this is not a suitable solution since smaller packet means a larger portion of the packet now consists of control data (network headers). The submss regime will then result in very small packets from all senders traversing the bottleneck, and overhead from such packets must not be ignored as the overhead will grow proportionally as more competitors enter the battle. The network shifts over in a useless state where each sender emit almost no useful application data per packet and the available bandwidth of the bottleneck is wasted on

[2] We have also explored the idea of using a mirrored (negative) version of the congestion window when in the submss regime. This modification sounded initially like a good idea, but it was hard to work with and became a source of confusion. We also had to deal with other mechanisms of TCP, so we dropped this approach over the one just described.

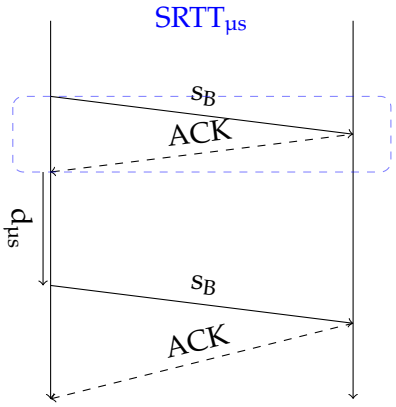


Figure 6.1: Packet Conservation Clock, no delayed acknowledgements

such packets. Additionally, this solution requires that at least one packet must be processed per sender for every RTT, and for shallow RTTs this result in lots of packets. This solution requires a significant increase in processing cost at all network elements in the path between the sender and receiver. Additionally, this also results in more processing done at both the sender and receiver, i.e. there is nothing good about this solution except for being simple. This naive solution results in identical behaviour as if the sender were not using the Nagle's algorithm and had application who did small writes all the time. This approach is, therefore, not scaleable and bring us to a dead-end.

$$d_{\mu s} + SRTT_{\mu s} = \frac{s_B * SRTT_{\mu s}}{W_B}$$
$$d_{\mu s} = (\frac{s_B}{W_B} - 1) * SRTT_{\mu s}$$

A scaleable solution could be to add a locally computed delay, $d_{\mu s}$, between each packet and simulate a congestion window of less than SMSS bytes per RTT. All senders continue to transmit full packets containing SMSS bytes, but now at a slower rate. This means that the gap between each packet grows as congestion window descends toward zero (can not be zero). For instance, if the sender where to send a packet of size SMSS bytes and waits for two RTT, this is indeed the same as sending two half-full packets per RTT. We can say that the sender has effectively sent the packet while holding a congestion window of a half, but notice that we this time never actually sent any less than one full packet containing SMSS bytes. A question that may arise; Why would the sender want to add delay to its own transmission when the same delay could be added later by the network as queueing delay? The difference lies in where such delay occurs, a localised delay within the host is always prefered over a shared delay across the network. All senders should prefer to add a short duration of the delay to their transmission such that whenever a sender does its transmission it happens through a shallow queue AQM. The self-motivated

goal of the sender is thus if all sender obeys by this principle it is more benefit able for everyone at the bottleneck.

This is indeed the same idea as used by TCP Nice, but our solution scales better and has some advantages. We never send more than one packet in a single transmission, so our solution is scalable as we do not cause any noticeable burst at the bottleneck. We know that this becomes a problem when our receiver starts to defer transmission of acknowledgements, but we solve this problem separately (see Section 6.3). Next, we use the whole range of the congestion window, while TCP Nice only let the sender defer transmission in an exact number of smoothed-RTTs. Finally, our approach makes senders competing at the bottleneck responsive for very low BDPs, whereas TCP Nice does not scale beyond 96 smoothed-RTTs. The scalability of our preservation clock depends on the sender's maximum segment size and the lowest permitted value of the congestion window.

6.3 Stretch Acknowledgement

We will now look at some of the challenges that come with having a receiver who merge acknowledgements. The receiver defers transmission of the acknowledgement until either δ, usually 2, segments have arrived resulting in a stretch acknowledgement, or a delayed acknowledgement returns because of an expired timer. This invariant makes the sender committed to keeping at least δ segments inflight per RTT, the receiver will then never get a chance to postpone the transmission of the acknowledgement and always return a stretch acknowledgement. The sender surely does not want to come in a situation where the acknowledgement clock block further transmission and at the same time have a receiver who is holding back the acknowledgement in expectation for more data to arrive.

Our solution to defer transmissions in the submss regime explicit this problem, so unless we address this issue will all our effort yield degraded performance from such receivers. Actually, it might sound like we are solving a non-existent problem or rather an easy problem to overcome. The use of stretch acknowledgement in the submss regime might not be required since we will always clock out less than δ segments per RTT, so acknowledgements which return back to back per RTT will also be less than δ. We will, therefore, generally perform better off than the conventional TCP in keeping the stress level of the network low, so we would be more than happy to just toggle off this mechanism at the receiver. The idea is to toggle the use of stretch acknowledgements whenever the sender enters or leave the submss regime. However, this information needs to travel to the receiver as a TCP option and would require negotiation between endpoints[3]. This will limit the scalability of the solution to only work in networks where the receiver supports this extension to the TCP header. We

[3] https://www.ietf.org/proceedings/97/slides/slides-97-tcpm-tcp-options-for-low-latency-00.pdf,https://tools.ietf.org/html/draft-wang-tcpm-low-latency-opt-00 (Work in progress)

will also struggle to keep the processing cost of the network low for when the receiver uses any significant δ values.

We should, therefore, keep this mechanism enabled all the time and instead try to look for other solutions which do not compromise the benefits of this mechanism. The first problem we need to overcome is to not hinder the sender from clocking out more packets in the event of no feedback from the receiver. This might sound crazy, but as long as the sending host keeps its packets evenly distributed with the packet conservation clock it should not be a problem (see Section 6.6). The penalty of the timer is thus avoided when enough segments arrive at the receiver and a stretch acknowledgement returns. It should also not be a problem if the timer of the receiver expires before enough segments have arrived at the receiver, so there is no hard requirement for the sender to reach transmission of δ segments. This is not the same as sending a window of segments and waiting for the feedback to return since this approach do not cause the link to go underutilised for when the receiver goes temporarily silent.

We need to solve another problem before this solution has any possibility of working. The sender is now no longer able to measure the RTT to the receiver since we have broken up the acknowledgement clock. The acknowledgement may arrive after multiple of RTTs later since we delayed the transmission of the sender. The measured RTT will appear to be longer than it really is since it includes all the time the acknowledgement was delayed at the receiver. We could maybe use the addition of Round-Trip Time Measurement (RTTM) to get precise RTT information through the use of timestamps [11]. However, even though this is a proposed standard it is not this easy; The receiver is required to echo the earliest timestamp received rather than the latest (see section 4.3 of RFC7323), so this does not bring us any further. The reason why the receiver has to echo the earliest timestamp is to give the sender a better picture of how long into the future the loss recovery (e.g. RTO) has to be set, this prevents unnecessarily retransmissions. We surely still want to keep this behaviour intact so the loss recovery adjusts to reflect the delay we add to our transmissions. We also need to know the RTT of the network so we are able to calculate the length of the gap.

The sender calculates the RTT with a timestamp as:

$$SRTT_{\mu s}^{loss} = Tc_{j} - ETs_{i}$$

where Tc_j is the current timestamp of the sender for j-th RTT, and ETs_i is sender's first timestamp echoed back from a set of segments. Therefore, the RTT measurement which includes the penalty of the receiver can be calculated from the delta between the echoed and current timestamp. The details of the smoothing process are not relevant to us, so we leave out these details. This method of measuring the RTT only becomes a problem when we are not bound by the acknowledgement clocked.

$$SRTT_{\mu s}^{pace} = SRTT_{\mu s}^{loss} - (Ts_{i+\delta-1} - ETs_{i})$$
$$SRTT_{\mu s}^{pace} = Tc_{j} - Ts_{i+\delta-1}$$

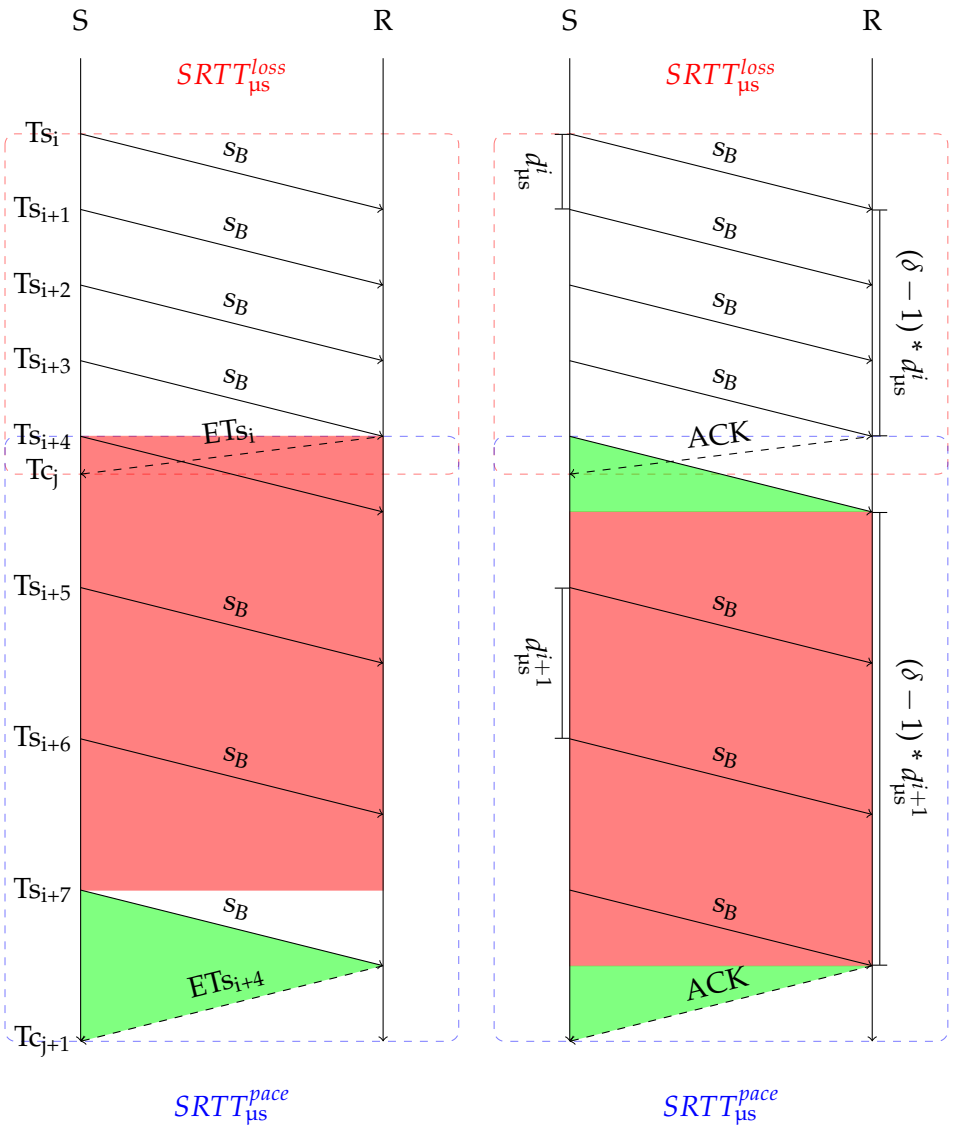
Our first idea was to store N most recent timestamps entries of the sender, so eventually, when a new acknowledgement arrives we could simply calculate the delta as before, but now with the corresponding entry in the list. We could then evict all previous entries which just got acknowledged from the list and start storing the next batch of timestamps. We can not simply store the lastest timestamp since we do not know the exact number of segments which gets acknowledgement by the receiver, so we may end up transmitting more segments than what the receiver acknowledges. However, this is impractical in practice since this requires the sender to store a potential very long list of timestamps in memory. This limits the scalability of this approach since we would be required to keep an upper bound for what δ could be to prevent this list from taking up too much memory.

$$SRTT_{\mu s}^{pace} = SRTT_{\mu s}^{loss} - (\delta - 1) * d_{\mu s}^{i}$$

Our next idea was to take advantage of the fact that the gap, $d_{\mu s}$, between all transmitted packets stays constant until the arrival of the next acknowledgement. The delay accumulated at the receiver is indeed the same delay as the sum of all gaps we have appended to the transmission of the sender.

6.4 Logarithmically Scaled Additive Increase

The constant additive increase phase of TCP does not scale well for either small or large congestion window values. We surely do not want to add one whole segment to the congestion window per RTT when dealing with small window values. The reason could be that the sender may have less than one segment inflight per RTT and adding one additional segment to the network gives faster than slow-start probing. Likewise, this also become an issue in networks where the sender has to maintain a large congestion window per RTT. The problem is now that the sender has to use a massive number of RTTs to regain lost capacity post a reduction phase [25, 61]. A possible scenario could be that the sender emits one million segments per RTT and reduces its congestion window to half of its original size. The process to regain the million takes the sender half a million RTTs. This process is, therefore, slow and inefficient and is the main reason why other congestion control algorithms were deployed over the standard TCP, e.g. Cubic [61]. Most modern congestion control algorithm tries to optimise TCP better in such networks, whereas no significant work has been done to do the same for low BDP path networks. The reason is simply that modern



(a) First idea

(b) Second idea

Figure 6.2: Packet Conservation Clock with $\delta = 4$

congestion control algorithms often have to design around the legacy TCP. The conventional TCP is also a headache when a new congestion control algorithm tries to probe slower since they risk being outcompeted.

The problem lies in how fast we grow the congestion window per RTT [61]. Currently, we always add one segment per RTT no matter the current value of the window, this results in a linear probe of the window per RTT. This behaviour needs to change such that we add a variable number of bytes to the congestion window. We must work in bytes since the addition of one segment does not work well in the submss regime.

We now need to select a slow-growing function for our window in bytes. A good candidate for this environment is the logarithmic growth function. The number of segments added to the congestion window per RTT should then only change when the input value of the function changes by a two order of magnitude. This property is essential when competing with the legacy TCP since we still need to behave properly through bottlenecks which have yet to segregate L4S traffic (scalable senders) from the legacy TCP (classical senders), so the freedom we thought we had turned out to be not so free after all. We intend to replace the current constant additive increase phase of TCP with a logarithmic increase counterpart. Although, the use of logarithmic increase has already been attempted in the literature [42] before, but not in the direction of the low BDP path problem.

We define a function, namely the *add* function, which is essentially a logarithmic growth function (see Equation (6.1)). We use this function to add a variable number of bytes to the congestion window per RTT (see Equation (6.2)).

$$add_{\mathrm{B}} = k0_{\mathrm{B}} * lg(S_{\mathrm{B}}/k1_{\mathrm{B}} + 1) \tag{6.1}$$

$$W_{\mathrm{B}} \mathrel{+}= add_{\mathrm{B}} \tag{6.2}$$

The *add* function is a logarithmic increase function with two growth constants, k0 and k1. We add by 1 before calculating the logarithmic of base 2 to ensure that the *add* function scale well from higher values of ssthresh to preferably all the way down the minimum value of 1_{B}. We use ssthresh rather than the current value of the congestion window to keep the processing cost to compute this function as low as possible per acknowledgement. The *add* function is then only recalculated whenever ssthresh changes, which happens either upon congestion (a sawtooth) or in the absence of congestion for longer periods [22]. We should, therefore, not be required to recalculate the *add* function more frequent than this interval and the worst scenario to encounter either of the condition is no more than once in an RTT.

It is important to notice that the additive increase phase is now varying and not constant for all values of ssthresh. It might be easier to imagine the growth as a linear increase function per RTT, where the angle of the function changes whenever ssthresh gets a new value. A low value of ssthresh equates to a slow additive increase phase, whereas the opposite

cause a steeper angle and therefore a more aggressive probing. The *add* function should now scale well for both small and large congestion window values, but we must first select appropriate growth constants to know whether this scales (see the proof in Section 6.4.1).

We need to update the congestion window differently between the normal operation of TCP and submss regime. This needs to be done such that we always accumulate add_B per RTT.

$$W_B \mathrel{+}= add_B / W_s, \qquad W_s >= \delta \tag{6.3}$$

$$W_B \mathrel{+}= add_B * SMSS_B / W_B, \qquad W_s < \delta \tag{6.4}$$

A sender who use the packet conservation technique to clock out packet needs to adjust the window such that the increase represents the period which has been missed out due to the lack of acknowledgement clock (see Equation (6.4)). While a sender who uses the acknowledgement clock can just spread out the additive increase phase over the number of packets in flight per RTT (see Equation (6.3)).

However, we discovered that unless the intention is to build a large sawtooth this will not work. We were not able to find any good solution to this problem so we instead simply scaled down the additive increase phase of LS-AIMD in some experiments using a scaling factor to see which benefits we were missing out.

6.4.1 Choosing values for the growth constants

Before we go loose and choose what would likely be good values for growth constants which would ensure scalability of the add function, we should look more deeper into what consequenses it bring to the protocol and existing traffic over the Internet. We list some contstraints which we believe needs to be taken into consideration before we make our choise:

1. We should try to add roughly one segment to the congestion window per RTT around the average of ssthresh currently experienced in the typical environments of TCP. We could then be able to share out the capacity evenly with the legacy TCP through both L4S and non-L4S bottlenecks. However, some cautions needs to be taken so we do not end up using a wrong estimate of the average of sshthresh. We may end up hurting the existing traffic by going too fast or risk being outcompeted by the legacy TCP if we try to go slower. On the other hand, the add function is an extremely slow growing function so we should be rather safe and not be required to use the exact average estimate. As long as we keep our estimate of ssthresh in the same ballmark as the real estimate by a two order of magnitude it should not be a problem, we will just then induce a minor disturbance at its worst. Ideally, we should also be able to choose the constants such that the protocol could coexist in a data centre setting with for instance DCTCP. The idea here is that we want our protocol to be easy

to fit a changing network topology, and should allow a more smooth deployment across private and isolated networks. However, we do not need the protocol to be compatible with both broadband and data centre environment at the same time. Therefore, to summarise we are looking for two set of growth constants, one for broadband and the other one for data centres.

2. We should try to keep the retransmission rate of TCP below the goodput it is able to utilise through the link. We should then be able to keep TCP efficient for the whole range of ssthresh. This contraint essentially limits how often TCP is able to hit the marking threshold after a reduction phase has taken place.

3. The constants should ideally hold numbers which represents a power of two such that both multiplications and divisions result in one single bit-shift operation. This is an important detail to keep in mind when doing the design phase since implementations which may derivere from this design should try to be as efficient as possible when it comes to CPU utilisation.

Starting with the first constraint (#1), we need to show that the add function from Equation (6.1) approxmiately equals to SMSS bytes, given the following condition:

$$add_B = k0_B * lg(\overline{S_B}/k1_B + 1) \approx= SMSS_B \tag{6.5}$$

$$\tag{6.6}$$

where $\overline{S_s}$ denotes the ssthresh experienced either over a broadband or data centre environment in segments of 1448_B or 8948_B respectively. The exact length of the segment may vary depending on which TCP options are in use (upto 40 bytes), but for our use case it is enough to prove that the approximation technique falls just within the ballmark of the TCP option space.

We now need to look at which restrictions the second constraint (#2) impose for our selection of the constants. The second contraint can generally be expressed for TCP as:

$$g = r(1-p)/p, \quad r < g \iff p < 0.5$$

where g is the goodput for a given sender with a certain retransmission rate (r) and loss probability (p). We see that in order to keep the constraint satisfied, $r < g$, the loss probability has to be less than 50%. This means that TCP should not induce a loss episode more often than every second RTT for all values in the congestion window. We should then be able to keep the retransmission rate of TCP below the achieved goodput. An additive increase from the add function should, therefore, be less than or equal to one decrease of the congestion window. We are happy as long as we are able to keep the inefficieny of TCP below 50% of the congestion window.

$$add_{\mathrm{B}} <= S_{\mathrm{B}}$$

We should thus always add no more than the value of ssthresh, which can be at its worst be upto half way to the congestion window (rate halving). Meaning, we do not discriminate between the graduall decend of DCTCP over the classic Reno backoff mechanism in our additive increase phase.

The slope of lg(x) monotonically decreases as x increases, so if we can show that the lowest value of S_{B} (1_{B}) has a slope of less than 1. We then know that the slope of the add function will always be constrained for all other values of S_{B}. We must, therefore, prove the following condition:

$$k0_{\mathrm{B}} * lg(1_{\mathrm{B}}/k1_{\mathrm{B}} + 1) <= 1_{\mathrm{B}} \tag{6.7}$$

where the add function of $S_{\mathrm{B}} = 1_{\mathrm{B}}$ has to yield an answer which is also no higher than 1_{B}. We can solve lg(x) by inserting the Taylor series expansion of ln(1 + x):

$$\begin{aligned} lg(1_{\mathrm{B}}/k1_{\mathrm{B}} + 1) &\approx= 1/(k1_{\mathrm{B}} * ln(2)) \\ k0_{\mathrm{B}}/(k1_{\mathrm{B}} * ln(2)) &<= 1_{\mathrm{B}} \end{aligned}$$

Then, $k0_{\mathrm{B}}$ can be given as:

$$\begin{aligned} k0_{\mathrm{B}} &<= ln(2) * k1_{\mathrm{B}} \\ &<= 0.7 * k1_{\mathrm{B}} \end{aligned}$$

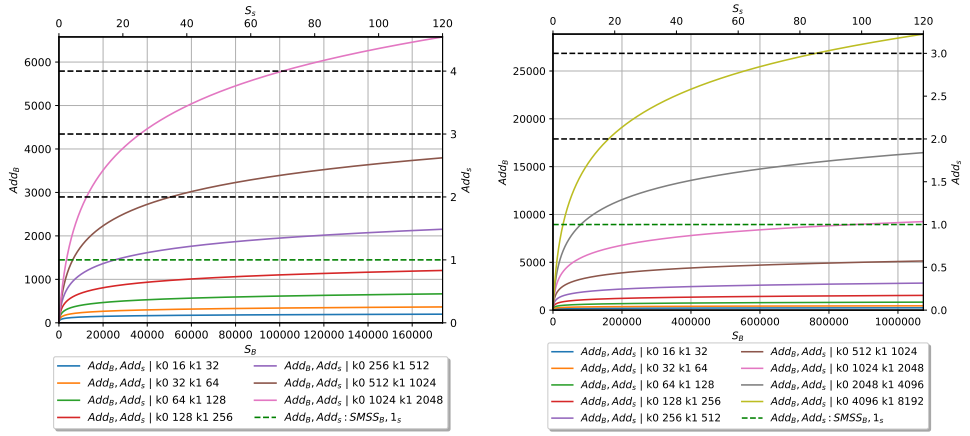
However, the last constraint (#3) require that we keep both constants as a power of two nuumber. This means, that we should try $k0_{\mathrm{B}}$ to be half way to $k1_{\mathrm{B}}$ rather than a factor of 0.7:

$$k0_{\mathrm{B}} = k1_{\mathrm{B}}/2$$

Now updating the first constraint (Equation (6.5) & **??**) whith the information we have gathered so far (a subsitution of $k1_{\mathrm{B}}$), we now have:

$$add_B = k0_B * lg(\overline{S_B}/(2 * k0_B) + 1) \approx= SMSS_B$$

$$\begin{aligned} \frac{\mathrm{d}add_B}{\mathrm{d}S_B} &< 1, \quad S_B = 1_B \\ &= k0_B/(ln(2) * k1_B * (1_B/k1_B + 1)) \\ &= k0_B/(ln(2) * (1_B + k1_B)) \\ &= 0.72 \end{aligned}$$

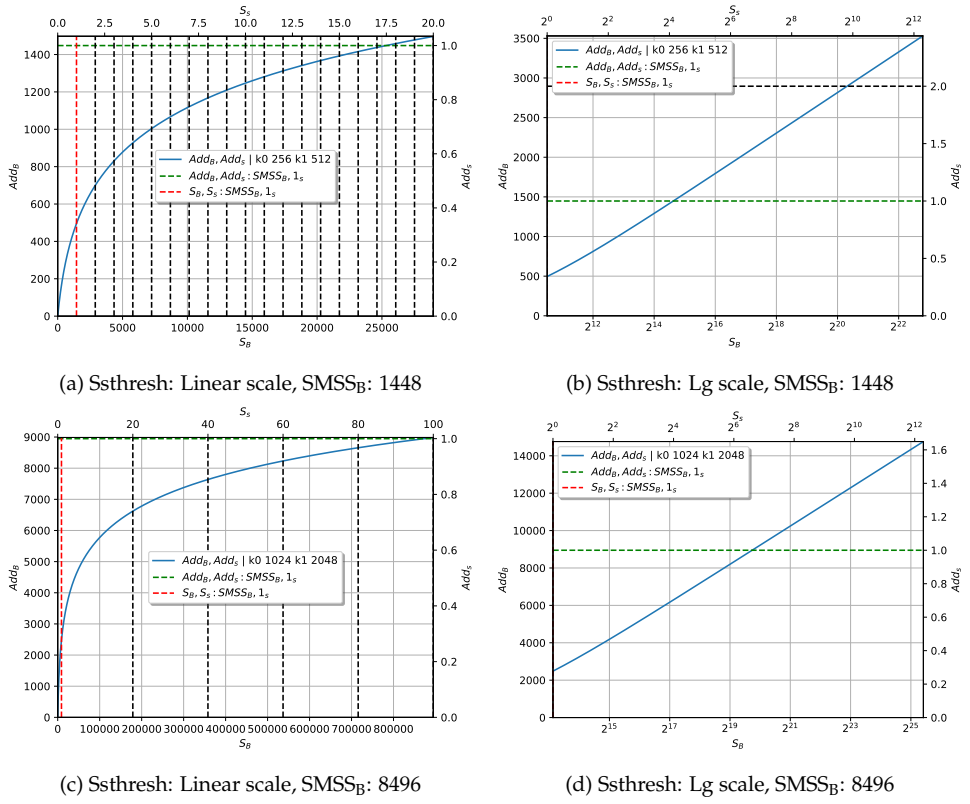


(a) $SMSS_B$: 1448 (b) $SMSS_B$: 8948

Figure 6.3: The Add function

MTU_B	**$SMSS_B$**	$k0_B$	$k1_B$	K0	K1
1500	1448	256	512	8	9
9000	8948	1024	2048	10	11

Table 6.1: Values of constants



(a) Ssthresh: Linear scale, $SMSS_B$: 1448 (b) Ssthresh: Lg scale, $SMSS_B$: 1448

(c) Ssthresh: Linear scale, $SMSS_B$: 8496 (d) Ssthresh: Lg scale, $SMSS_B$: 8496

Figure 6.4: The Add function

We can, therefore, say that the following formulas satisfies all our constraints:

$$add_B = 256_B * lg(S_B/512_B + 1)$$

6.5 Multiplicative Decrease

Now that we have a scalable way to increase our congestion window we should next be looking at how we can preserve the multiplicative decrease of TCP. It turns out, that we only need to make a minor modification to the protocol to achieve this. First, we must also work in bytes here since the window could be less than one segment per RTT. We, at the same time, also moves the lower bound of the ssthresh, from being a minimum of δ segment per RTT, to a floor of two bytes per RTT. This should give us enough room to scale for lower RTT, and a floor has to be chosen anyways to prevent the congestion window from falling to zero so why not two bytes. A congestion window of two bytes should give sufficient scalability to the protocol for the time being, but this could be the recipe for the next scalability issue in the future. However, this should let us test the scalability of the design and also let us know if the scalability it brings to the protocol is worth the extra complexity added to the protocol. The artificial limitation put on the design can then be shifted to lower threshold, this should not require any major modification to the design as most work given in this design should scale fine for an even lower threshold.

6.5.1 Loss Recovery

In order to make LS-AIMD resistant against losses, we follow the ideas of Hsiao et al. to insert enough segment paced to counter multiple losses. Then when some segments are lost, we still get enough duplicate acknowledgements to trigger retransmission.

6.5.2 LS-AIMD Reno

LS-AIMD Reno has a very simple backoff mechanism, and this design does not change this. We simply extend the rate halving of Reno into the submss regime:

$$S_B = max(inflight_B/2, \quad 2_B)$$

LS-AIMD Reno simply set ssthresh to half of the current congestion window. This reduction makes a set of sender competing at the bottleneck eventually converge. A sender who reduces its congestion window by half will do an exponential decay, it does not matter if the sender is in submss regime or not it still cause the appropriate reduction.

6.5.3 LS-AIMD DCTCP

Surprisingly, LS-AIMD DCTCP keeps a very identical reduction phase as earlier. We also extend the reduction phase of DCTCP such that:

$$S_{\mathrm{B}} = max(inflight_{\mathrm{B}} * (1 - \alpha/2), \quad 2_{\mathrm{B}})$$

Initially, we thought maybe we would need to do a more complex reduction phase as a sender in submss regime grows its RTT. However, we then understood that by simply extending the RTT the congestion signal weakens, but this is the same as two senders with different base RTT. Just that one sender only has one signal for each RTT.

6.6 Security Concerns & Deployment Challenges

We should now look at how existing traffic over the Internet may be affected by the proposed modifications. Our modifications only change the server-side of the conversation, so we only look at the possibility for our senders to be lured into malfunctioning. We have to think of all scenarios where the feedback returned by the network has some form of evil intentions. We have yet to confirm that the proposed modifications suggested in this design document actually works, but for the time being, we assume that our modifications work as given out in this design. Our focus here is thus to only discuss in the direction where the incorrect information provided to the sending host is now causing stability issues. We first look at how one malicious peer could accomplish such an attack against a scalable sender and then we give out details of what the possible outcome would be.

Our modification to TCP tries to preserve the acknowledgement clock of TCP for all values in the congestion window with the information available so far. The acknowledgement clock ensures that we are able to insert a fixed segment count into the network by assuming that the acknowledgement clock provides a form of guarantee that the previously sent data leaves as new data enters the network. We could say that the safety of the protocol remains as is since we trust our changes to accomplish this to be working, i.e. if we are wrong here then there is no point in breaking the protocol since it is already broken.

We also refrain from probing faster in scenarios where the receiver keeps returning partial acknowledgements so we do not reintroduce ACK Division Attack. We wait instead until at least one new segment has been acknowledged before we make modifications to the congestion window.

We could still be a victim one attack vector which is caused by the scalability promise of the design, so some caution should be taken when using some of the ideas given here.

An attack vector could be executed by a third-party in an attempt to reduce the performance of the sender and thus cause a significantly downgraded performance to the receiver. The sender could be lured into

thinking that the network path to the receiver is heavily congested and thus be pushed back resulting in a non-optimal performance to the receiver. If such an attack is possible it would also impact the conventional TCP, but our choice to scale for submss regime gives a much worse worst case. The scalable sender would keep backing off toward the new floor of 2 bytes per RTT vs. the previous floor of two segments per RTT. This would result in terrible performance in networks where either the bitrate or RTT is extremely high. The motivation behind such an attack could be to take out service if done in large-scale or just to gain an advantage over another sender at the bottleneck.

Such an attack could be prevented if ECN nonce was used to detect tampering of ECE endpoints, but ECN nonce was never fully deployed over the Internet and has now become obsoleted [10, 65]. Instead, there exist several other viable methods which can be used by the sender to detect cheating receivers [60, 66][4].

6.7 Summary

We have in this chapter given out a detailed plan on how we intend to scale TCP for the submss regime. We introduced a fractional congestion window to keep congestion window values between segment boundaries and also explained how TCP will now be able to hold values which are less than one segment. We have also explained how TCP could use the packet conservation clock to transmit segments in the submss regime. Additionally, we have come up with a scheme which allows the sender to clock out segments even in the event of the challenging stretch acknowledgement mechanism at the receiver. Our work contributes with the born of a new scalable congestion control algorithm that is meant to rework the additive increase phase of TCP which scales for both low and large BDP paths.

We are next going to look at how convenient an implementation could be by employing this work into action.

[4] https://tools.ietf.org/html/draft-ietf-tsvwg-ecn-l4s-id-06 Appendix C.1 (Work in progress)

Chapter 7

Implementation

We are now ready to produce our first implementation from the design document as an initial prototype. We first decide what needs to be included in our prototype; we certainly needs to limit ourself from trying to do a full implementation this time. Our first goal should be to test the main ideas of the design and only when we have tested them fully should we continue our work, then complete the remaning parts of the implementation in as efficient manner as possible.

We started intially with Linux kernel version 4.13.16, but we were later in the project forced to upgrade to the latest stable version 5.0.0 at that time. A lot of work went down to track down scalability issues which made 4.13.16 inconvient and hard to use for our intentions. These scalability issues were solved in the latest stable version of the kernel, but it then meant that we had to reinitate our project. We thought we now were in a better position, but it did not went long before we started to have issues. We, eventually, found a possible regression bug in the kernel. We intend to report this bug as soon as possible, but we need more time to verify that this is not due to some fault in our setup. A lot of work went down the road at track the bug and find a possible fix so we could continue our developement, but this did not happend before we had already spent a lot of time blaming ourself for introducing the bug. An important lesson was learned that day; always retest the vanilla implementation of the kernel whenever you perform an upgrade.

The implementation started as a rather complex beast and has more or less become a quite simple prototype. This is indeed the result of a continous cycle of trail and error.

We make an implementation of the design proposal as a pluggable Linux kernel congestion control module. The module introduce both algorithms, LS-AIMD Reno/DCTCP, in a joint module. Our work also modifies other part of the Linux TCP stack to support our module. These changes are made such that other congestion control algorithms modules of the kernel may eventually take the benifits of our design.

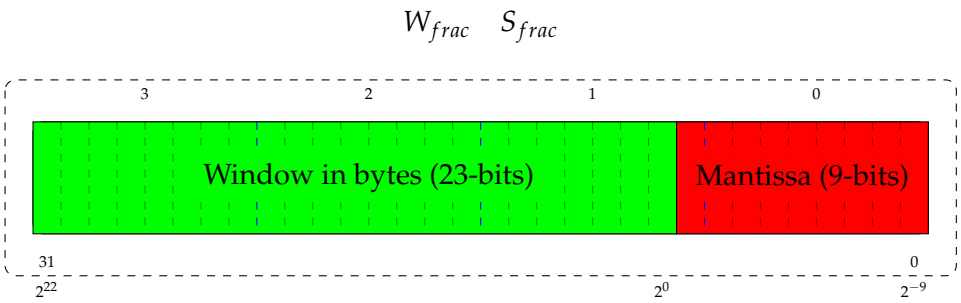


Figure 7.1: Fractional Congestion Window (4 bytes)

7.1 Design Decisions

We must include all modifications needed to extend TCP for the submss regime, since this requirement is the problem statement we are trying to solve. Unfortunaely, this means that we have to implement all consept of the design to a certain degree. We are first satisfied with our implementation when we are able to produce a prototype which scales for the submss regime without any compromises.

We are, however, not making any modifications to the receiving host of the conversation. The Linux kernel is unfortunaely not made such that we can easily change δ of the receiver (yet), so our prototype will only scale for the default value of $\delta = 2$. An alternative approach could be to let the receiver be a Windows endpoint, and then change $TcpAckFrequency = \delta$ through registry [1]. Our prototype will work with receiver who try to keep a more substainal δ, but this require the tester to change a global constant which we have defined to the new value of δ.

In the future, our plan is to be able to dynamically adjust the global constant of the sender. This will enable the soluton to work seamless to a variety of receiver with different stretch acknowledgement factor. We will also turn off loss recovery of TCP when evaluating LS-AIMD since loss should not occur very often in our simple topology. This should enable us to benchmark the performance of pure ECN. Since we have turned off fast retransmit and fast recovery, we simply set congestion window to ssthresh whenever we receiver the first mark of the RTT.

7.2 Fractional Congestion Window

We implement the fractional congestion window as an unsigned 4-byte value in the control block of TCP. Our reasoning for placing the fractional congestion window in the control block of TCP is to let it be available to all other congestion control modules. We constrain the value of the fraction congestion window to be within the range of 0 and SMSS bytes.

We scale up the fractional congestion window and ssthresh such that we can keep a window of fractional bytes (see Figure 7.3) [2]. We use the 9

[1] https://support.microsoft.com/en-us/help/328890

[2] this is required for what follows for the rest of this chapter.

rightmost bits to store the mantissa part of the window; the window can then hold byte values with a precision as low as 2^{-9}. However, this restrict the maximum segment size we can support. The maximum segment size we support is then only 2^{23} bytes rather than the full size of 2^{32} bytes, but this is still way more than what we would ever need. We now have to deal with scaled integer arithmetic, but this should only be a minor annoyance. We are also opening for the possibility for fraction congestion window to scale well below a single byte per RTT some time in the future.

7.3 Packet Conservation Clock

Our design gives out an idea of how packet conservation clock may be used to clock out segments in the submss regime efficiently. There is no point in reinventing the wheel, so we use the pacing facility found in the Linux kernel to clock out segments in the submss regime. We are not using the pacing mechanism to maintain a rate, but merely to defer transmission of the sender, i.e break up the acknowledgement clock.

The acknowledgement clock helps to spread out the bursts of a sender and also play a vital role in reducing the risk of synchronised senders. The transmission of the sender is bound by the acknowledgement clock, so the queue dynamics of the bottleneck keeps the sender in check. A queue reduce the risk of repeative burst to occur since the time an acknowledgements takes to return shifts according to its position in the queue. The queue is therefore useful in spreading out the sender's next round of transmission.

The problem with using pure pacing to keep a clock is that it schedule next depature from the time the last segment was transmitted and use congestion window as an upper limit. The sender using pacing generally keeps a rate which reflects the averaged rate reported back by the acknowledgements of that round. A sender whom decides to pace react, therefore, more slowly to changes in the queue and end up repeating the previous round of burst. Another problem that occurs with the use of pacing is that multiple senders can more easily become synchronised at the bottleneck. The AQM will then have hard time in desyncronising the flows since the propability to be marked will periodically switch between 0 and 100% for all senders.

We, therefore, extend the pacing facility of the Linux kernel to act more like the acknowledgement clock. Our approach is to make adjustements to the timeout whenever the network report back with more recent information. We still use the pacing timer to schedule a transmission some time in the future, but we addionally validate the timeout as soon as new intel arrives.

7.3.1 Reschedule an Earlier Depature

The pacing timer is set some time in the future and the incoming acknowledgement tells us about a queue which is in the process of

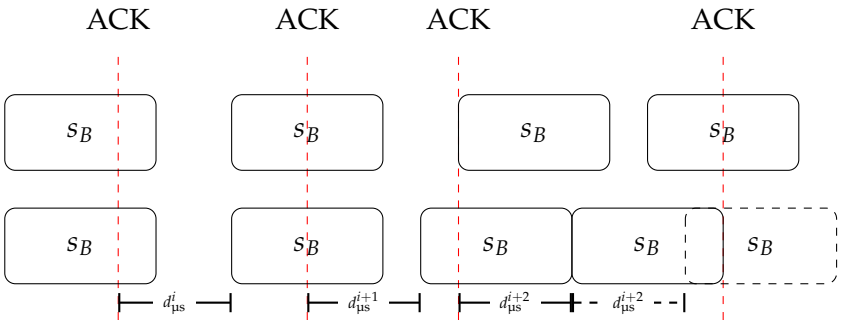


Figure 7.2: Packet conservation clock: early transmission

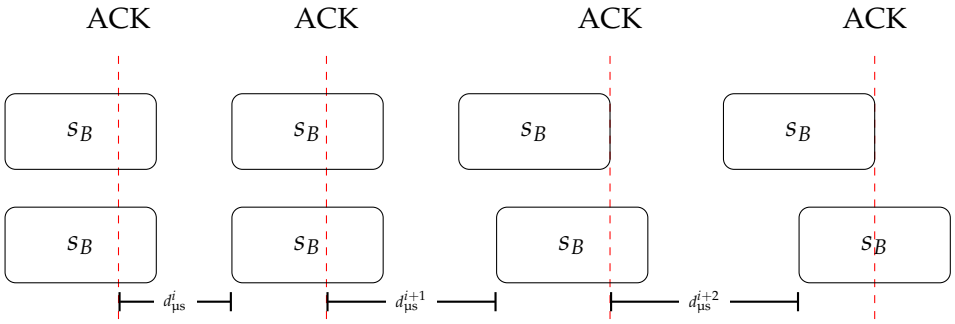


Figure 7.3: Packet conservation clock: postpone transmission

draining. We should make a proactive action and reschedule the pacing timer so the the segment departs earlier. We, by doing this, minimise the gap between the previous sent segment and next segment. We are not going any faster than what the congestion window permits, rather we are just fixing the gap to be the exact minimum distance allowed by the packet conservation clock.

7.3.2 Reschedule an Postponed Depature

Simmarlirarly, we should also reschedule the pacing timer when we detect that a queue is in the process of building. We could found ourself in a situation where we our previous RTT information is now no longer valid. We should postpone the next transmission of the sender so we do not end up reinforcing the previous round of burst. Note, that our approach is simmilar here too and we do not cause the sender to go slower than what the packer conservation clock permits.

7.4 Logarithmic Increase

We replace the constant additive increase phase of TCP with a logarithmic phase. The logaritmic increase phase is done in two stages; We first accumulate the fractional congestion window with either *add* bytes after one full window of segments has been acknowledged (one RTT) or if the sender is in submss regime we advance on each acknowledgement enough

to give the same effect. We then check if the fractional congestion window now represents at least one whole segment, if so update increase the congestion window by one and then keep the remainder in the fractional congestion window.

7.4.1 Add function in integer arithmetic

$$add_B = k0_B * lg(S_B/k1_B + 1)$$
$$add_B = lg((S_B \gg K1) + 1) \ll K0$$

We must be able to implement add function using only integer arithmetic as Linux does not allow floating point calculation in the kernel. The add function cannot be as is since it loses precision whenever ssthresh is lower than $k1_B$. The bit-shift of K1 will result in 0 whenever ssthresh is less than $k1_B$.

$$\begin{aligned} add_B &= k0_B * lg((S_B + k1_B)/k1_B) \\ &= k0_B * (lg(S_B + k1_B) - lg(k1_B)) \\ &= (lg(S_B + k1_B) - K1) \ll K0 \end{aligned}$$

We can imporve the precision by adding $k1_B$ inside lg and then subtract K1 afterward. It is the same calculation, but now we do not need to add by one and neither do we have do the division. However, we still lose precision when we calculate the logarithmic. The addition needs to yield at least 1024, and sstresh needs to be higher than 512 to yield the first integer. It gets worse as we try out next integer due to the property of the logarithmic function. The logarithmic function grows very slowly, and the input of the function must double to get the next successive integer. We need to calculate the logarithmic function with floating point precision using only integer arithmetic. We need to select a precision and yield the answer with the given precision. One must see the accuracy in comparison with the performance of the function. The lg needs to be very fast such that when RTT is very shallow, we can still apply the add function once every RTT. The whole point of operating in submss regime is to achieve very shallow RTT. It can also be other factors which may play a role such as battery consumption of the device.

We implement the logrithmically function based on earlier work since the effiency of this function was out of scope[67][3].

7.5 Packet Processing

We use a state variable in the control block of TCP to cache the current value of the add function to make it easy access to the whole TCP stack. We ask the current congestion control module to update the value whenever

[3] https://github.com/dmoulding/log2fix

ssthresh changes. By default, if the hook is not implemented we simply use the value of a single segment. However, the additive increase is still in the hand of the module and the use of the state variable is thus optional.

We also implement hooks which allows the current congestion control module to know about event such as when the TCP stack is about to enter or leave the submss regime.

7.6 Modes of operation

We need to distungish between the normal operation of TCP and when we need to use fractional window. Therefore, we need to extend TCP to operate in two modes, namely submss and non-submss. By default, TCP starts in the non-submss mode and goes into the submss mode when the congestion window is less than two segments. The reason for it being two is to interwork with delayed acknowledgement, we do not want to use acknowledgement clock if we risk being delayed by the receiver. Likewise, we leave the submss mode when the congestion window goes above two segments again. We add a new state variable, snd_submss, to TCP which functions as a boolean to detect if the flow is currently in submss regime or not. When TCP changes modes we need to either upscale or downscale cwnd, ssthresh and add. Any previous saved value from cwnd or ssthresh must also be updated. We also need to make sure that we transfer these variables correctly such the only difference is the scalling.

7.6.1 Non-submss

This mode is mostly unchanged for the typical TCP, but we use logarithmic increase instead of the constant additive increase in the congestion avoidance phase. We keep using acknowledgement clock to clock out more segments. The additive increase is spread over the whole RTT, each ACK adds "add" bytes to a counter. When this counter reaches above SMSS*W bytes, will we add counter/(SMSS*W) segment to the congestion window and finally subtract the counter with the number of segment added. We need to do this way so that we do not lose any precision.

In other word, TCP will continue to have information about both number of whole segments and fraction of a segment. In situation where an ACK does not advance the congestion window to a new segment is the congestion window left untouched and the fraction saved to the next update.

7.6.2 Submss

The congestion window is now less than 2 segments and we must therefore starts using the fractional congestion window. The inital value of the fractional window must be 1 segment, since we only enter this mode when congestion window is less than 2 and above 0. We do not send new segments upon receiption of old segments, we instead delay our transmission and pace out new segments based the penalty we calculate.

7.7 Linux Kernel Module

We implement our congestion control algorithms in a joint Linux kernel module. Our module is inter-connected with our modified DCTCP module, such that we can reuse existing code which measures the extent of the congestion.

We implemented the mandatory additive increase phase of TCP. The logarithmic increase phase is common between TCP Prague LIMD and TCP Reno LIMD, where as ssthresh is set seperately by the algorithm. We kept the standard slow start as is, but we do not double the congestion window directly when in submss regime since we already have an advantage through the pacing timer.

Our module also implement serveral new hooks which get called by the kernel during packet processing.

7.8 Summary

Ironically, the only reuqirement we set to a congestion control module is to implement the hook which demands a additive increase constant (*add*) and to move the floor of congestion window from 2 segments to 2 bytes. Our modification done to the input and output engine of TCP will kick in as soon as the congestion window falls below 2, or δ, segments.

Part III

Results

Chapter 8

Evaluation

We now have an implementation that should satisfy all our design goals and scales well for lower RTTs in the network. It is tempting to right away test the implementation against the existing Linux TCP. However, evaluating in such a manner might lose vital aspects of the solution and risk hiding away some of the added benefits or flaws which are otherwise not explored by such a direct comparison. Our approach is, therefore, to construct a baseline with our best current solution and first later check if some of the existing congestion control algorithms can match our results. We did initially construct a set of experiments to look for the need for a solution. This process was essential to confirm earlier research, and it is first later that we are on the path to scale TCP for the submss regime. We are also testing the motivation behind the need for the DCTCPs ECN feedback loop to control senders; the question boils down to how much gain do we get from using the extent of congestion over merely the existence of congestion. A Reno like backoff mechanism is preferred as it is much simpler. The benefits of DCTCP like congestion detection technique may out-beat rate halving by a large margin, but the dynamics of TCP in the submss regime are not well known and not certainly for our design proposal. We are, therefore, retesting prior research to see if our algorithms have the same remarks in terms of scalability. We confine our best current solution to be the one congestion control algorithm which scales the most. Our choice is thus limited to be either LS-AIMD DCTCP or Reno. We also evaluate two configurations of RED for DCTCP to see which scales best.

We plan to let each congestion control algorithms undergo a set of extensive testing to nominate a winner. We construct each test to have their own independent set of requirements. We initially test simple configurations and slowly lean toward more and more complex configurations. As soon as we reach the scalability of our solution, we halt our testing and summerise our conclusion. In other words, when none of our algorithms fulfils the requirements we have set for the test, we stop testing. We pick a random 100ms time slot of the experiment to evaluate, but we are strict in our selection, so we do not end up benchmarking the slow start phase of TCP. A 100ms time slot is enough to see the dynamics of TCP as we are mainly dealing with low base RTTs. However, we look at

more extended periods when performing a final scalability test.

8.1 Functional Test

First out, we do a systematic functional test of our implementation against the design proposal. We start by testing each of the components in the design separately and continue until we are satisfied with the results. We then see the need to test the interaction between these isolated features. If we find any significant flaws during testing, we see it as an indication of a bug in the implementation or likely a faulty assumption we made in the design period. In this case, we abort testing and look for which part of the design is to blame, and then find an appropriate fix to the fault and only then resume testing. We repeat all previous tests to look for other bugs produced by the fix. We have to go this route as changing a significant part of the TCP machinery can be both challenging and time-consuming process. On the other hand, we have attempted to split up the design into as small and independent chunks as possible, but we still have significant components. We must, therefore, evaluate each of these components extensively to reveal any flaws before testing the whole algorithm.

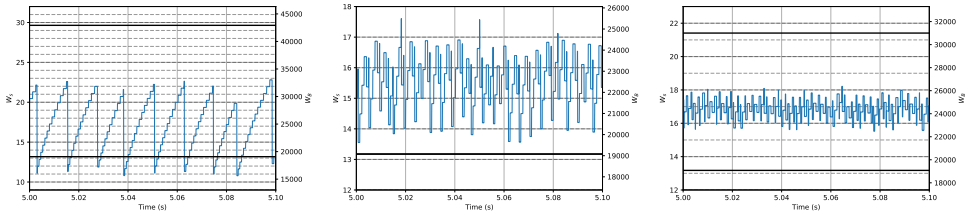
8.1.1 Logarithmically Scaled Additive Increase Multiplicative Decrease (LS-AIMD)

One of the requirements we had to solve when scaling for submss regime was to take in use a fraction window. The fractional window holds a value between zero and SMSS bytes. The fractional window is also scaled up to hold a mantissa part to increase precision in our calculations. The congestion window holds the original congestion window and the fractional window. Our goal with this test is to verify the correctness of the fractional window and fractional ssthresh, but this is not so easy to do without inspecting the whole congestion window and ssthresh. We plot the current value of the congestion window and ssthresh as follow:

$$W_{\mathrm{B}} = W_{\mathrm{s}} * SMSS + (W_{\mathrm{frac}} \gg 9)$$
$$S_{\mathrm{B}} = S_{\mathrm{s}} * SMSS + (S_{\mathrm{frac}} \gg 9)$$

We have no interest in looking at the mantissa part of the window, so we scale it down before plotting. We are then able to look at both window values in bytes over an RTT. Additionally, we add dashed lines to mark segment boundaries to make our plots easier to compare against other conventional congestion control algorithms. We also add indicators to the plot to reveal the min and max threshold of RED; this includes the base RTT, so we get the real picture as seen from the senders' perspective. We can then look at the current values of the window vs the experiment parameters.

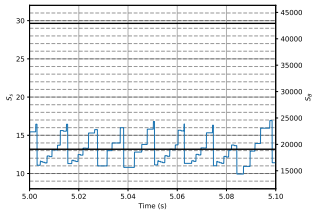
We conduct a simple experiment to validate the congestion window and ssthresh, the very first experiment (see Table 6.1 #1) is sufficient. A



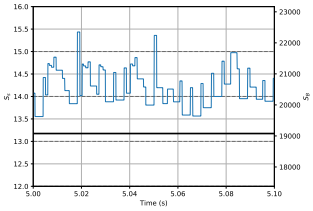
(a) W: LS-AIMD Reno Ramp

(b) W: LS-AIMD DCTCP Step

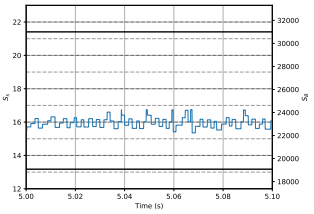
(c) W: LS-AIMD DCTCP Instantaneous



(d) S: LS-AIMD Reno Ramp



(e) S: LS-AIMD DCTCP Step



(f) S: LS-AIMD DCTCP Instantaneous

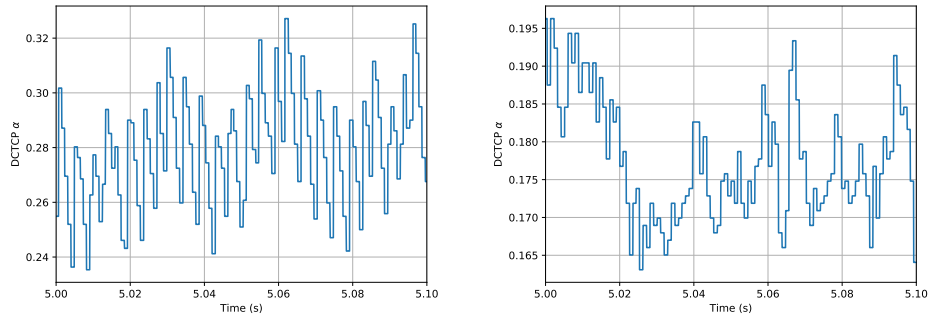
Figure 8.1: Experiment #1: W & S

single flow is enough to see if the TCP machinery is working as given out in the design proposal. We are not testing scalability here, but merely the interaction between our algorithms and the AQM.

Figure 8.1 shows the congestion window and ssthresh over the period: 5-5.1 seconds. The Figure 8.26b, Figure 8.1b and Figure 8.1c shows the congestion window for LS-AIMD Reno and both variants of LS-AIMD DCTCP respectively. Likewise, Figure 8.1d, Figure 8.1e and Figure 8.1f gives the ssthresh in the same order.

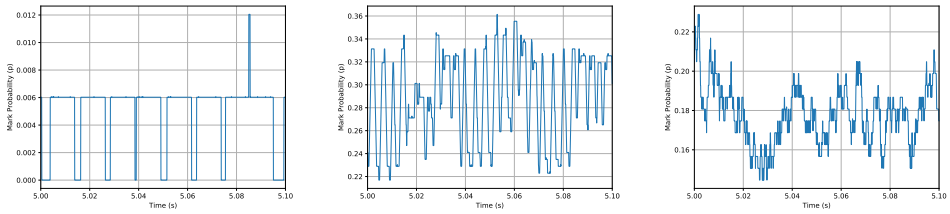
We see that all algorithms perform well by looking at their ssthresh; the value is indeed not very far from the minimum threshold of RED. This is a good indication as it clearly shows the willingness of TCP to cooperate with the AQM. Note that the additive increase phase is accumulating approximately a segment to the congestion window per RTT. We now know that the logarithmic aspect of the algorithm is working correctly since Add of one segment approximately translates to $20 * SMSS$ bytes of ssthresh. Next, we see that the congestion window is kept very stable for all configurations except LS-AIMD Reno which does rate halving. The congestion avoidance phase of LS-AIMD Reno is slow and inefficient. Remarkably, LS-AIMD DCTCP instantaneous has the most stable probing. One question that may arise, LS-AIMD Reno configuration does rate halving, but why is congestion window sometimes a fraction of a segment over the new value of ssthresh? We have done this on purpose as we apply the additive phase on all rounds, regardless of whether this round results in a mark or not. The congestion window is, therefore, reduced to half and additionally increased by the appropriate bytes per RTT.

In Figure 8.2, we look at the observed marking level of our configurations. Starting with LS-AIMD Reno, we see that the algorithm has on average a low level of marking. Meanwhile, we see that both DCTCP configuration induces a higher level of marking. The step configuration keeps



(a) α: LS-AIMD DCTCP Step

(b) α: LS-AIMD DCTCP Instantaneous



(c) p: LS-AIMD Reno Ramp

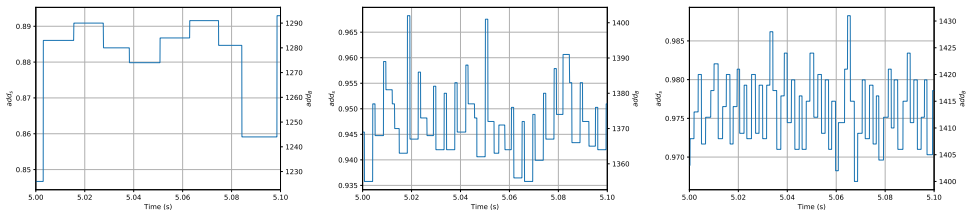
(d) p: LS-AIMD DCTCP Step

(e) p: LS-AIMD DCTCP Instantaneous

Figure 8.2: Experiment #1: Loss rates

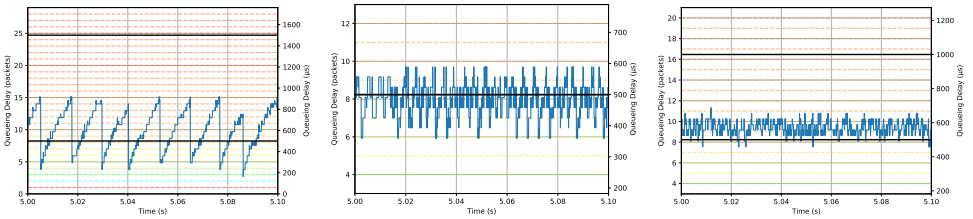
the highest level of marking on average. Nonetheless, step marking was expected to behave like this as the scheme has a simple on and off marking policy based on the instantaneous queue length whereas the other two configurations depend on a probability to produce a mark for the same queue length.

We look at the respective alpha of both DCTCP configurations to investigate our earlier observation of why LS-AIMD DCTCP instantaneous yield a more stable congestion window dynamics than the others. In Figure 8.2, we see that the step configuration is not well smoothed as LS-AIMD DCTCP instantaneous. A ramp-up in alpha is a consequence of multiple marks in a short period, and otherwise, a descend in alpha comes from a period of silence. To keep a stable congestion window, we except the alpha of DCTCP to have an even balance of marks over a short period. Such that both up and down transitions occur with evenly spaced interval. The step configuration relies on fast feedback from the receiver about the queue dynamics to produce a smooth alpha, but from the time the first mark takes place and gets delivered to the sender several successive marks follow because the queue length is still above the single instantaneous threshold. Therefore, the step configuration produces many successive marks, followed by a period of silence when eventually the queue length of the bottleneck goes below the marking threshold of RED. The instantaneous configuration differs here as each packet that arrives for enqueueing in RED has only a certain probability of being marked based on the current queue length where a higher queue length is more likely to produce a mark. These figures form a strong belief that RED is working,

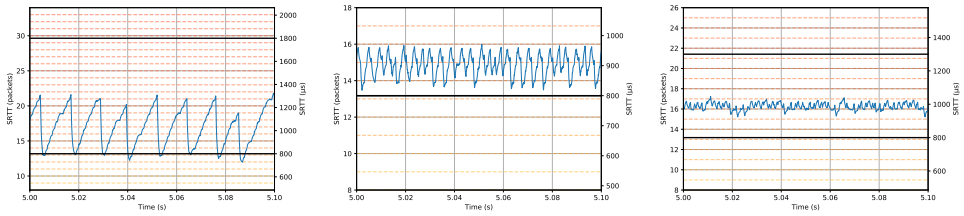


(a) LS-AIMD Reno Ramp (b) LS-AIMD DCTCP Step (c) LS-AIMD DCTCP Instantaneous

Figure 8.3: Experiment #1: Add



(a) Q: LS-AIMD Reno Ramp (b) Q: LS-AIMD DCTCP Step (c) Q: LS-AIMD DCTCP Instantaneous



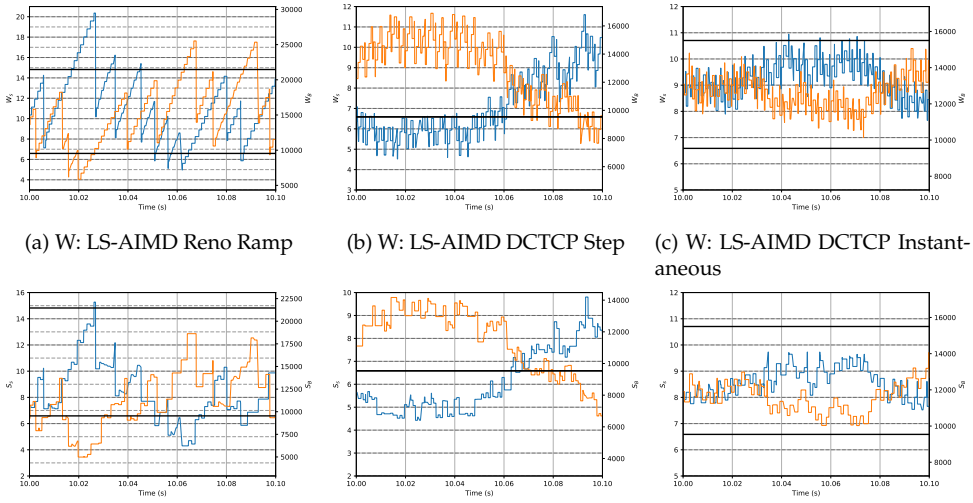
(d) SRTT: LS-AIMD Reno Ramp (e) SRTT: LS-AIMD DCTCP Step (f) SRTT: LS-AIMD DCTCP Instantaneous

Figure 8.4: Experiment #1: Queueing delay & SRTT

but our experiment result tells that RED with the default configuration of DCTCP is at a disadvantage. Even though the smoothness of the alpha may come as a possible problem later are we still satisfied with how accurate the sender can replicate the congestion level of RED.

In order to verify our finding so far, we take a look at the exact number of bytes that get added to the congestion window per RTT. In Figure 8.3, we confirm our earlier assertion that the add function adds approximately one segment per RTT. Additionally, we see that the add function calculates at most once per RTT. The Reno configuration is less resource-hungry than the other configurations because it has fewer reduction phases. However, as earlier, we see that the Reno variant is struggling in term of stability because of rate halving. Both DCTCP configurations calculate the add function more often, but unlike Reno, they perform well in keeping the number of bytes to add stable.

We should now look at the dynamics of the instantaneous queue and understand what impact it causes on the SRTT of the sender. In Figure 8.4, we see the queueing delay on the top and the SRTT at the bottom for



(a) W: LS-AIMD Reno Ramp

(b) W: LS-AIMD DCTCP Step

(c) W: LS-AIMD DCTCP Instantaneous

(d) S: LS-AIMD Reno Ramp

(e) S: LS-AIMD DCTCP Step

(f) S: LS-AIMD DCTCP Instantaneous

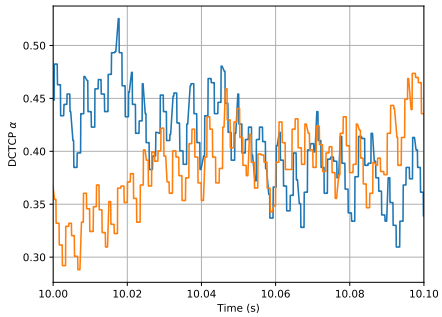
Figure 8.5: Experiment #2: W & S

all configurations in the same respective order. First, we take note at how the Reno variant has large sawteeth in both metrics. Rate halving is cumbersome as it results in large sawteeth, which makes it impossible to keep a stable RTT because of large fluctuations in queueing delay. Meanwhile, both our DCTCP configurations were capable of keeping a low queueing delay and delivering a stable end-to-end RTT over the same experiment.

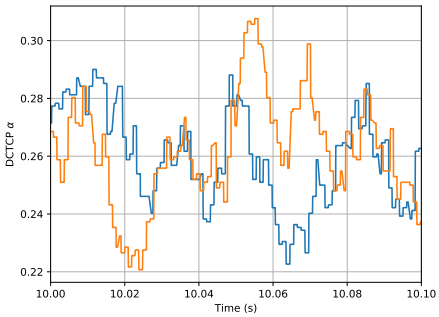
A take away from this experiment indicate that a proper smoothing of the congestion level is beneficial to produce stable results, and LS-AIMD Reno has no benefits except low marking level and lots of disadvantages. We now move over to our next experiment, now experiment #2, and introduce a competing flow to see how well the dynamics of LS-AIMD work out between two competitors.

We, as earlier, look at congestion control of the sender by investigating the congestion window and ssthresh. From Figure 8.5, we notice that all algorithms converge to their appropriate share of the window and well within the min and max parameter of RED. However, all LS-AIMD algorithms appear to be diverging periodically to unite slowly again. This uneven share of capacity could indicate an error in our solution if both competitors observe the same level of loss. We, therefore, have to look into the marking level of RED and additionally see if our DCTCP configurations are holding a proper value of alpha to understand the reasoning behind this shareout.

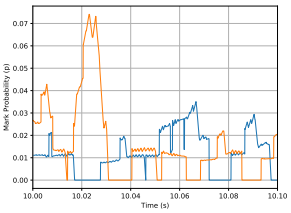
In Figure 8.6, we see the marking level of the network in an identical layout as in our first experiment. On top, we see the respective alpha of both DCTCP configurations, and upon inspecting them against the shareout of the congestion window, we now understand why our senders were probing slightly uneven. The capacity shareout between competitors mirrors their knowledge about the congestion level, and if alpha is not



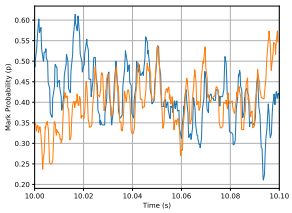
(a) α: LS-AIMD DCTCP Step



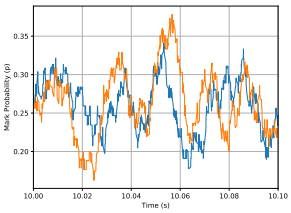
(b) α: LS-AIMD DCTCP Instantaneous



(c) p: LS-AIMD Reno Ramp



(d) p: LS-AIMD DCTCP Step



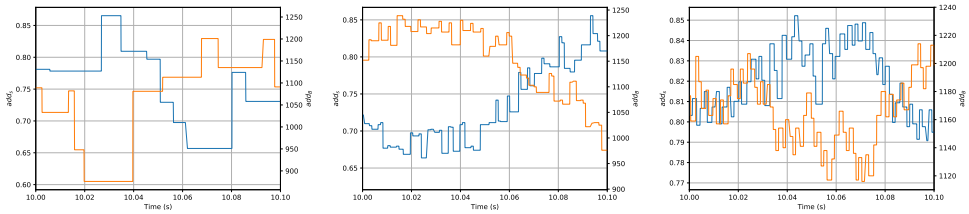
(e) p: LS-AIMD DCTCP Instantaneous

Figure 8.6: Experiment #2: Loss rates

equal, then it does not make any sense for the congestion window to be of equal size. DCTCP's adjustment to the congestion window bases itself on the current level of congestion so a lower alpha will always result in a more significant congestion window. However, we see that both of our DCTCP configurations calculate alpha accurately as the marking level of RED is no different. Nonetheless, if stability is a concern, it may be a good idea to smooth out alpha with a different set of weights such that alpha becomes stable, which again would make the congestion window of the sender and marking level of RED stable. However, some caution should be taken as a slower adapting alpha impacts the rate at which two competitors converge to an even share.

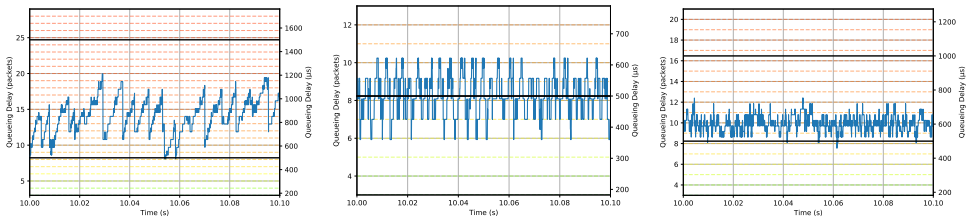
We also see that the instantaneous configuration of DCTCP has the most stable alpha, while the alpha of the step configuration periodically results in spikes. Another vital remark to notice about the step configuration is its now higher marking rates. On the other hand, the remaining configurations also slightly increased marking rate from the previous experiment do we not the same degree of rapid increase. In other words, our earlier observation that a stable alpha produces better results stands.

As earlier, in Figure 8.7, we see the same trend in the calculation of the add function. The number of bytes added strictly follow the dynamic of the other metrics. However, notice that the number of bytes to add is now slightly less because of a lower ssthresh. We can also now with certainty say that our logarithmically aspect of the algorithm is working correctly since a ssthresh of 10 KB equals to approximately 1100 bytes of add. Taking another look back at the congestion window, we see that the congestion

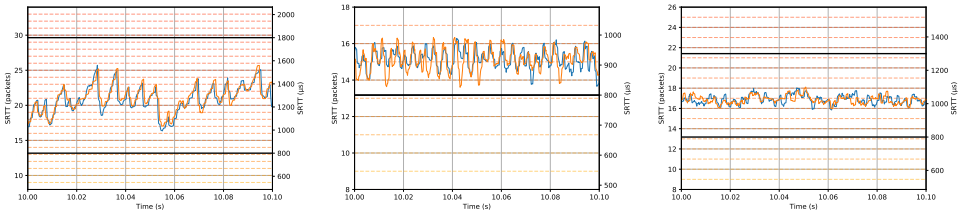


(a) LS-AIMD Reno Ramp (b) LS-AIMD DCTCP Step (c) LS-AIMD DCTCP Instantaneous

Figure 8.7: Experiment #2: Add



(a) Q: LS-AIMD Reno Ramp (b) Q: LS-AIMD DCTCP Step (c) Q: LS-AIMD DCTCP Instantaneous



(d) SRTT: LS-AIMD Reno Ramp (e) SRTT: LS-AIMD DCTCP Step (f) SRTT: LS-AIMD DCTCP Instantaneous

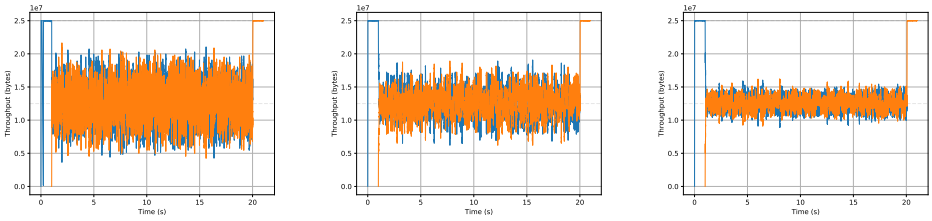
Figure 8.8: Experiment #2: Queueing delay & SRTT

window is indeed increasing by slightly fewer bytes now than earlier.

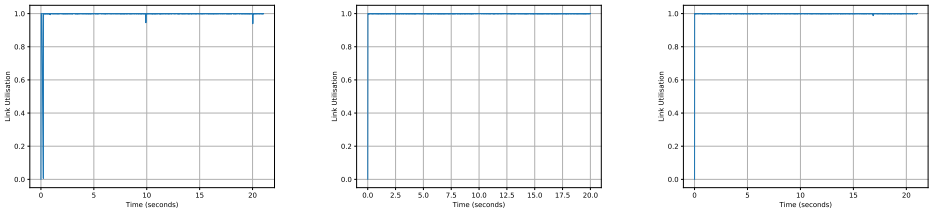
Going back to the network delay aspect of our evaluation in Figure 8.8, we surprisingly see no significant change in this experiment vs the previous one. Although, in Reno configuration, our readings now tells that both queueing delay and SRTT is more at the centre of the ramp of RED. Meaning, RED works better at stabilising the queueing delay with a higher number of senders since, unlike DCTCP configurations, RED marks on average queueing delay.

Finally, one crucial detail left to investigate would be the link utilisation and the capacity shareout between the two competitors. Staying with the same layout for our algorithms, we notice that all of our configurations maintain full link utilisation throughout this experiment in Figure 8.9. Our use of shallow base RTT in our experiments allows a very rapid startup phase, which results in full utilisation immediately, but since we are not evaluating slow-start can we ignore it for now.

However, when moving forward with the evaluation of the respective capacity shareout among the two senders, we see massive fluctuations



(a) Throughput: LS-AIMD Reno Ramp (b) Throughput: LS-AIMD DCTCP step (c) Throughput: LS-AIMD DCTCP Instantaneous



(d) Utilisation: LS-AIMD Reno Ramp (e) Utilisation: LS-AIMD DCTCP step (f) Utilisation: LS-AIMD DCTCP Instantaneous

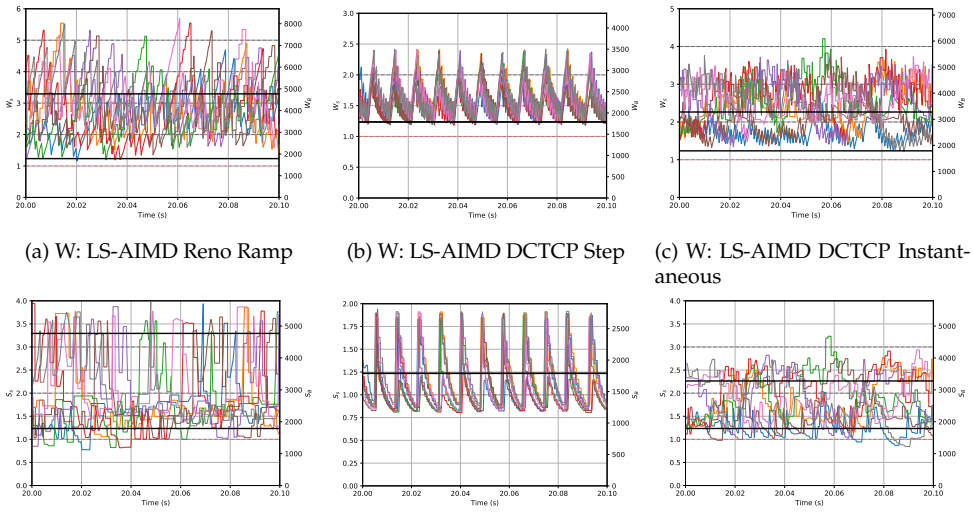
Figure 8.9: Experiment #2: Throughput & link utilisation

over short periods in both the Reno and DCTCP step configuration. The Reno configuration performs worst as it does rate halving. The second place goes to the step configuration because of the lack of a smooth alpha. Not surprisingly, LS-AIMD DCTCP instantaneous beats the other configurations with large margins. A smoother alpha translates into better stability because of less drastically modifications happens to the congestion window.

Our conclusion for this experiment undermines our observations from the first experiment. There is a particular advantage in using the extent of congestion over merely the existence of congestion for smooth congestion control. We have additionally proven that LS-AIMD does indeed converges, and when it does the algorithm distribute the capacity evenly among the competitors while maintaining low queueing delay at the bottleneck.

8.1.2 The Submss Regime

We are now ready to move over to the more exciting aspect of our work. We have, up until now, evaluated the probing requirement of TCP to scale for the submss regime, but it is first now that we test out our hypothesis about whether or not keeping a lower congestion window gives results of similar quality. To achieve the environment of submss regime, we introduce a few more flows, now 8, and lower the base RTT from 300 to 100 microseconds. We want to show that by only running a small number of flows in parallel over a low base RTT enforces a low congestion window on average for a bitrate as significant as 200 Mbps. LS-AIMD should have no problem in sharing the capacity among the competitors evenly and additionally maintain the shallow queue of RED.



(a) W: LS-AIMD Reno Ramp

(b) W: LS-AIMD DCTCP Step

(c) W: LS-AIMD DCTCP Instantaneous

(d) S: LS-AIMD Reno Ramp

(e) S: LS-AIMD DCTCP Step

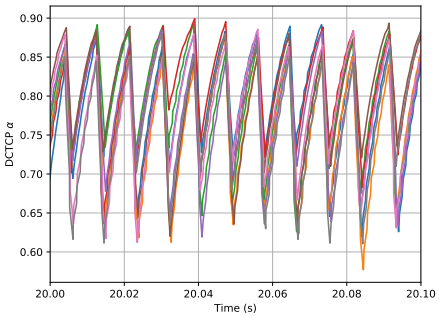
(f) S: LS-AIMD DCTCP Instantaneous

Figure 8.10: Experiment #3A: W & S

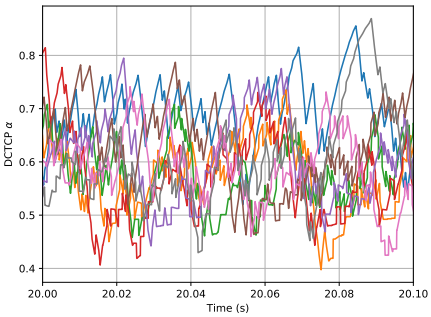
We continue our evaluation in the same fashion as earlier. Starting with the congestion window and ssthresh in Figure 8.10, we notice that there is always some flows which periodically keeps a transmission rate low enough to be considered inside the submss regime. However, all senders within the DCTCP step experiment appears to be synchronised. DCTCP step configuration is not able to keep a stable congestion window as all flows increase and reduce their congestion window together. Additionally, we observe that the other configurations appear to be also having some issues of their own. More specifically, we see that, unlike the previous experiment, LS-AIMD probe a little bit faster than the desired rate. Although we do not see any severe problems in them as they continue to converge, we would have dig into the marking level of RED to understand why LS-AIMD seems to be going faster than it should.

Investigating the marking rate of the network in Figure 8.11, we immediately notice a remarkably high loss rate in the DCTCP step configuration. Surprisingly, all senders appear to be arriving simultaneously at the bottleneck within a short period. On the other hand, when we look at the DCTCP instantaneous configuration, we see that there is no apparent synchronisation between the senders, but the DCTCP alpha does not seem to be as smooth as earlier. However, when we compare the alpha of DCTCP to the marking rate of RED, it is clear that the approximation of DCTCP is not to blame. We see identical dynamics in both metrics for all senders. Additionally, we observe the same problem in the marking rate of RED in the ramp configuration.

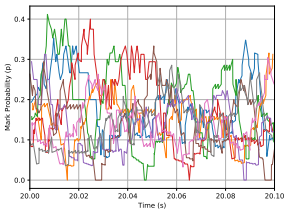
Now by comparing the loss rate against the congestion window, we see that the respective flow within each experiment does whats indeed it is told to do and adequately react on the higher marking rate by lowering its congestion window to yield for other traffic. However, we learn that the uneven balance happens precisely at the convergence point between the



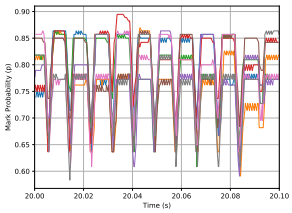
(a) α: LS-AIMD DCTCP Step



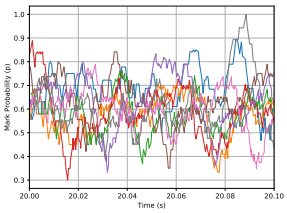
(b) α: LS-AIMD DCTCP Instantaneous



(c) p: LS-AIMD Reno Ramp



(d) p: LS-AIMD DCTCP Step



(e) p: LS-AIMD DCTCP Instantaneous

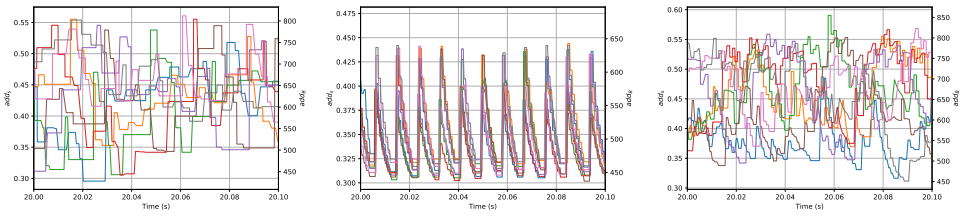
Figure 8.11: Experiment #3A: Loss rates

group of senders that probe within the submss regime, against the other group of senders not currently in the submss regime.

Our concluding remark for the possible cause for this problem to under folds lies in the way the sender transmits its segments. A sender within the submss regime uses the pacing timer of the Linux TCP implementation to schedule a segment sometime in the future. However, there is no guarantee that the pacing timer work with the same accuracy as a sender maintaining the acknowledgement clock, so divergence occurs whenever either of the mechanism probes faster than the other and since we are working with shallow RTTs a small mismatch turns into a fatal outcome. This problem could be minimised by manually adjusting the ratio of the pacing rate via an already existing global option in the Linux implementation. However, we found it hard to equalise them and improve the result any further. Therefore, another possible cause could be that the pacing timer repeats previous rounds of congestion, which is not the case for an acknowledge clocked sender since a queue help in spreading synchronisation of the next round.

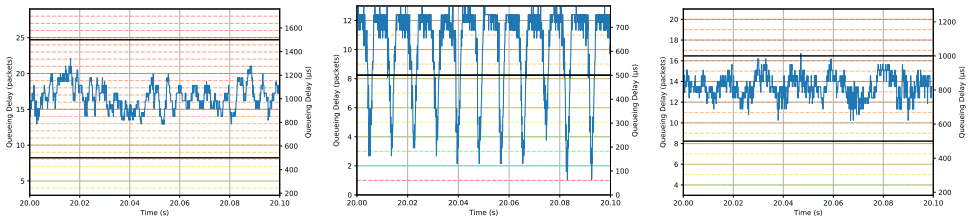
In Figure 8.12, we see that the number of bytes added to the congestion window per RTT shares a similar trend because ssthresh is neither stable.

Moving on to the queueing delay and SRTT in Figure 8.13, we see a small increase in both metrics for all configuration. The queueing delay now resides closer to the centre of the thresholds of RED. However, LS-AIMD DCTCP step turns into periods with high variations in a span of a short period because of continuous synchronisation among the competitors.

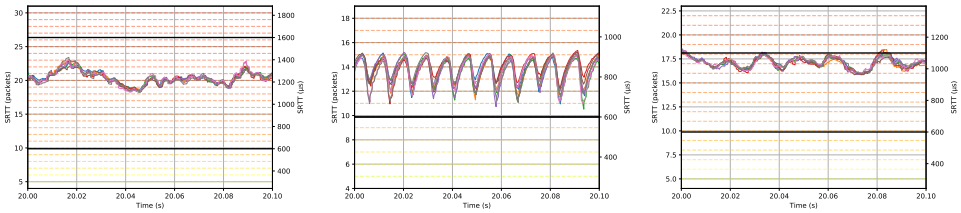


(a) LS-AIMD Reno Ramp (b) LS-AIMD DCTCP Step (c) LS-AIMD DCTCP Instantaneous

Figure 8.12: Experiment #3A: Add

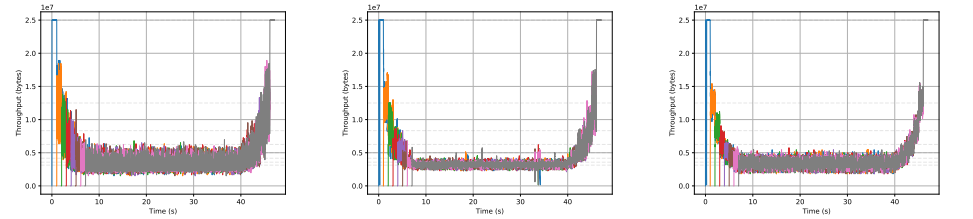


(a) Q: LS-AIMD Reno Ramp (b) Q: LS-AIMD DCTCP Step (c) Q: LS-AIMD DCTCP Instantaneous

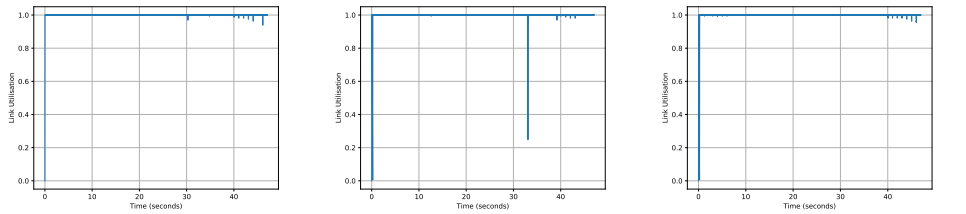


(d) SRTT: LS-AIMD Reno Ramp (e) SRTT: LS-AIMD DCTCP Step (f) SRTT: LS-AIMD DCTCP Instantaneous

Figure 8.13: Experiment #3A: Queueing delay & SRTT

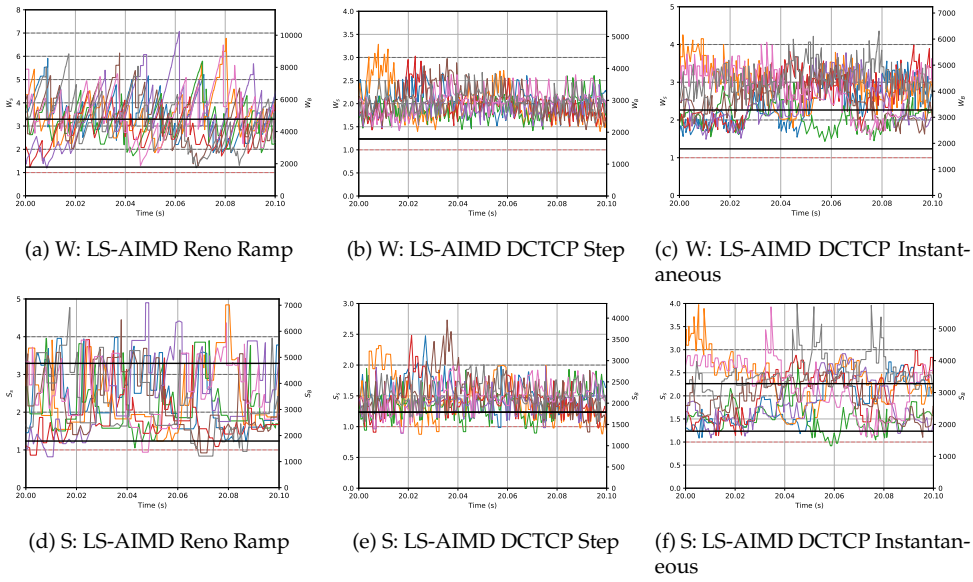


(a) Throughput: LS-AIMD Reno Ramp (b) Throughput: LS-AIMD DCTCP step (c) Throughput: LS-AIMD DCTCP Instantaneous



(d) Utilisation: LS-AIMD Reno Ramp (e) Utilisation: LS-AIMD DCTCP step (f) Utilisation: LS-AIMD DCTCP Instantaneous

Figure 8.14: Experiment #3A: Throughput & link utilisation



(a) W: LS-AIMD Reno Ramp (b) W: LS-AIMD DCTCP Step (c) W: LS-AIMD DCTCP Instantaneous

(d) S: LS-AIMD Reno Ramp (e) S: LS-AIMD DCTCP Step (f) S: LS-AIMD DCTCP Instantaneous

Figure 8.15: Experiment #3B ($\delta = 2$): W & S

Finally, in Figure 8.14, we see that the throughput between all competitors shareout evenly. LS-AIMD DCTCP step has the most stable shareout. The even sharing of capacity comes from the fact that all competitors induce only small modifications to the congestion window. We also confirm that all configurations maintain full link utilisation for the whole duration of the experiment period.

To sum up this experiment, we observe that some synchronisation is going on the LS-AIMD DCTCP step configuration and for the other configurations, we only see a slight decrease in performance from experiment #2.

We now repeat the previous experiment, but now with stretch acknowledgements enabled. The receiver is, thus, holding on to the acknowledgement such that only one acknowledgement returns for every second segment received. We again look through the same set of metrics, but now we additionally look at the SRTT pace.

In Figure 8.15, we immediately notice that the synchronisation among the competitors in the LS-AIMD DCTCP step configuration is now gone. This unsimilarity should explain to us why such a robust synchronisation took place in the previous experiment. We only introduced a tiny difference in our setup, which was to make the receiver hold on to every second acknowledgement. Given that the sender keeps at least two segments inflight paced in the submss regime will we always trigger the awaiting acknowledgement, but what happens when the receiver returns every acknowledgement? Moreover, what implication do we get if we are required to keep above one, but below two segments inflight? Let us say we have a congestion window of approximately 1.25 segments per RTT, which is the case for experiment #3. The sender will invoke a transmission of a full segment every $1/1.25 = 0.8$ RTT. We further now that the pacing timer initiates from the tail of the last transmission. We are, therefore,

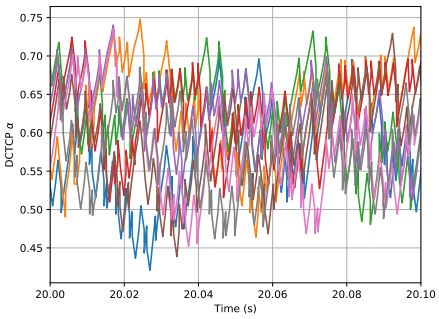
clocking out next transmission before an acknowledgement of the previous segment arrives. First, this means that any of our effort to postpone or early transmit the pending segment is of no use. Next, since we must transmit a new segment with outdated queue dynamics will we always repeat the burst of the previous round. However, why does the use of stretch acknowledgements eliminate synchronisation? Recall, that the sender may only keep δ, which is two here, segments inflight in the submss regime. When the sender sends its first segment, no credit is returned by the receiver, so as soon as the next transmission occurs, this limit reaches. Thus, no further segment may go inflight before an acknowledgement crediting a whole segment arrives. The queue desynchronise the sender since no further segment entered the network before new intel about the queue arrived. This process repeats itself periodically such that new transmissions only happens with updated queue dynamics.

We observe quite identical results in all configurations except the LS-AIMD DCTCP step experiment. Not only is this experiment free from any form of synchronisation, but additionally, no divergence occurs between the set of competitors. Unfortunately, this confirms our earlier claims we made in the previous experiment and the problem now diverge into two separate hard to solve problems. Starting with the first problem, we now know that that especially for a step marking network scheme, we are required to take in use stretch acknowledgements to hinder synchronisation. The next problem we deal with across all configurations comes from a mismatch between the pacing timer and the acknowledgement clock. The only reason why the step configuration is free from divergence dynamics in this experiment comes from the fact that all senders probe right below the boundary of the submss regime. Even though, all configurations mark close the boundary do the other configurations take in use a more complex AQM.

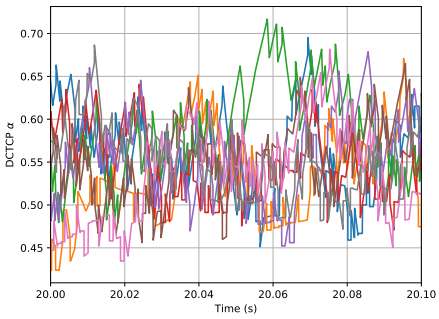
Next, in Figure 8.16, we relook at the marking probability of each configuration and compare it to the previous experiment. We notice that the step marking scheme no longer induce synchronisation in the marking. Meaning, all configurations now stay desynchronised. Otherwise, we do not see any marginal difference in the marking, which indicates that a more substantial stretch acknowledgement factor would not necessarily cause a higher marking rate. The design of LS-AIMD is, thus, assumed to scale just fine in an environment with a higher stretch acknowledgement factor.

In Figure 8.17, we see that like every other metric does we a similar trend in how each sender in each configuration applies their additive increase phase. The step configuration has the most stable additive increase phase.

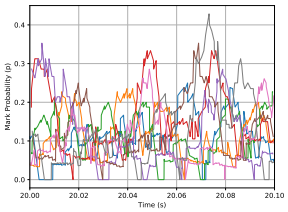
Next, in Figure 8.18, we take another round at evaluating the queueing delay and SRTT of the network. We notice no significant change in the queueing delay of the bottleneck except desynchronisation of the step configuration. However, we see a slight increase in the SRTT in all schemes. The added delay comes from the delay appended to the transmission by the sender in the submss regime and since the receiver uses stretch acknowledgements do we see an increase in the RTT between



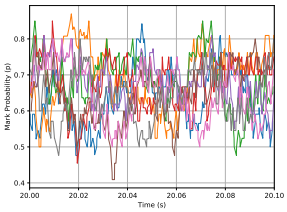
(a) α: LS-AIMD DCTCP Step



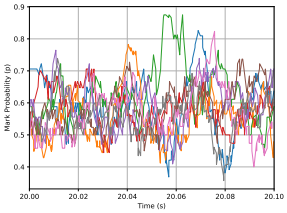
(b) α: LS-AIMD DCTCP Instantaneous



(c) p: LS-AIMD Reno Ramp

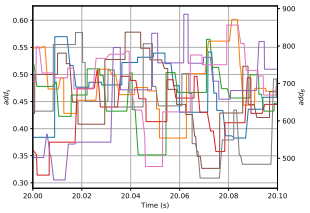


(d) p: LS-AIMD DCTCP Step

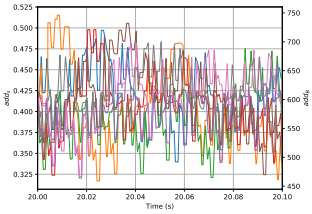


(e) p: LS-AIMD DCTCP Instantaneous

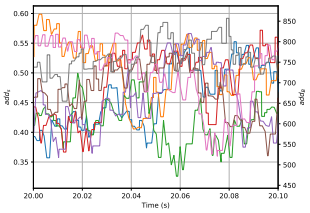
Figure 8.16: Experiment #3B ($\delta = 2$): Marking rates



(a) LS-AIMD Reno Ramp

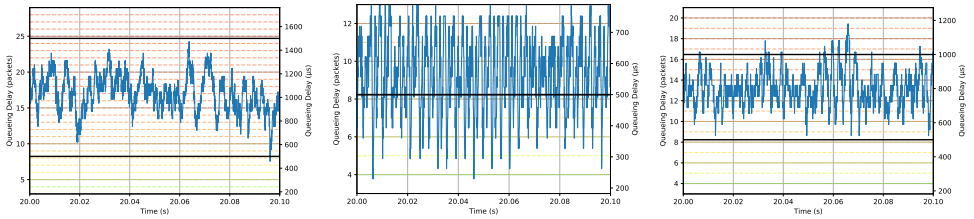


(b) LS-AIMD DCTCP Step

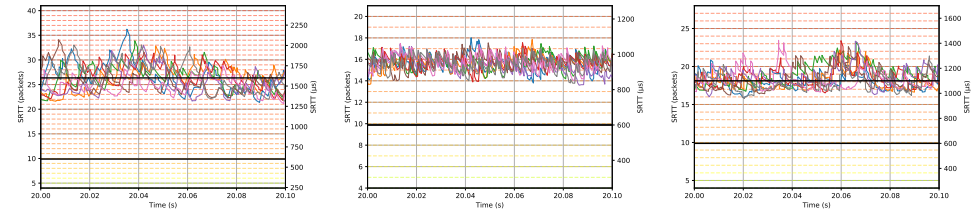


(c) LS-AIMD DCTCP Instantaneous

Figure 8.17: Experiment #3B ($\delta = 2$): Additive increase constant

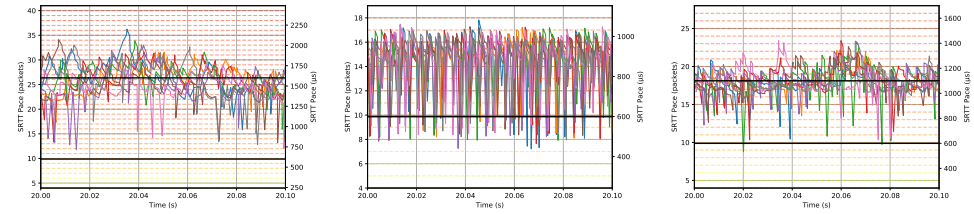


(a) Q: LS-AIMD Reno Ramp (b) Q: LS-AIMD DCTCP Step (c) Q: LS-AIMD DCTCP Instantaneous



(d) SRTT: LS-AIMD Reno Ramp (e) SRTT: LS-AIMD DCTCP Step (f) SRTT: LS-AIMD DCTCP Instantaneous

Figure 8.18: Experiment #3B ($\delta = 2$): Queueing delay & SRTT



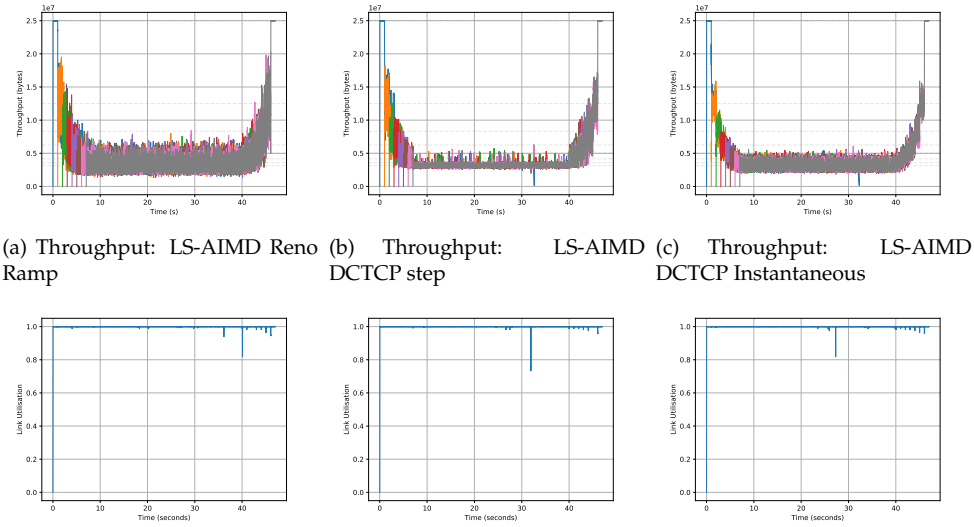
(a) LS-AIMD Reno Ramp (b) LS-AIMD DCTCP step (c) LS-AIMD DCTCP Instantaneous

Figure 8.19: Experiment #3B ($\delta = 2$): SRTT Pace

the endpoints. An increase might indicate the thoughts off a bug in the implementation or design, but this is, in fact, nothing else than a feature of the LS-AIMD algorithm. This dynamic let the sender schedule loss recovery long enough into the future to avoid spurious loss recovery.

In Figure 8.19, we investigate the dynamics of the SRTT pace. First, we notice that unlike SRTT, the pace version is sometimes lower. That is done intentionally such that the sender removes the appended delay to maintain a proper packet conservation clock. However, we observe a lack of stability caused by spikes in the measurements. We believe this comes from the fact that the sender does not know whether or not the incoming acknowledgement is from transmission in the submss regime or not. We have, thus, failed in instructing the sender to make the proper action as both have separate logic for what constitutes a valid RTT measurement.

Finally, in Figure 8.20, we go back to the evaluation of the throughput and link utilisation. There is no significant change in our measurements of these metrics contra the previous experiment. However, between 32-36 seconds in all configurations, we see a single episode of spurious



(a) Throughput: LS-AIMD Reno Ramp (b) Throughput: LS-AIMD DCTCP step (c) Throughput: LS-AIMD DCTCP Instantaneous

(d) Utilisation: LS-AIMD Reno Ramp (e) Utilisation: LS-AIMD DCTCP step (f) Utilisation: LS-AIMD DCTCP Instantaneous

Figure 8.20: Experiment #3B ($\delta = 2$): Throughput & link utilisation

retransmission. We never lost any packets in any of our experiments, so the exact reason for why the spurious retransmission took place is uncertain, but most likely indicate some problems as the bug happens consistently across all configurations in the same period.

8.2 Stability & Syncronisation Test

We continue evaluating Experiment #3 but now look more closely at the stability and synchronisation among the competitors. This additional intel should let us know about how good all our LS-AIMD algorithms performs throughout the experiment. We, as before, compare the experiment with stretch acknowledgements enabled against the other experiment where the receiver returns acknowledgements immediately. The primary purpose of this additional evaluation is to build up new knowledge to know where more work should be put in to get even more excellent results.

8.2.1 Link Utilisation

We, first, need to make sure that the link stays fully utilised while all competitors participate. It is crucial to prioritise this metric as any loss of utilisation is never desirable.

In Figure 8.21, we see that in each configuration the link never goes below 0.999 utilisation over an average of 10 ms. The step configuration has the most stable utilisation for the experiment without stretch acknowledgements. However, the instantaneous configuration performs best overall when also taking account of stretch acknowledgements. The ramp scheme, on the other hand, is in no way performing any worse but loses due to a

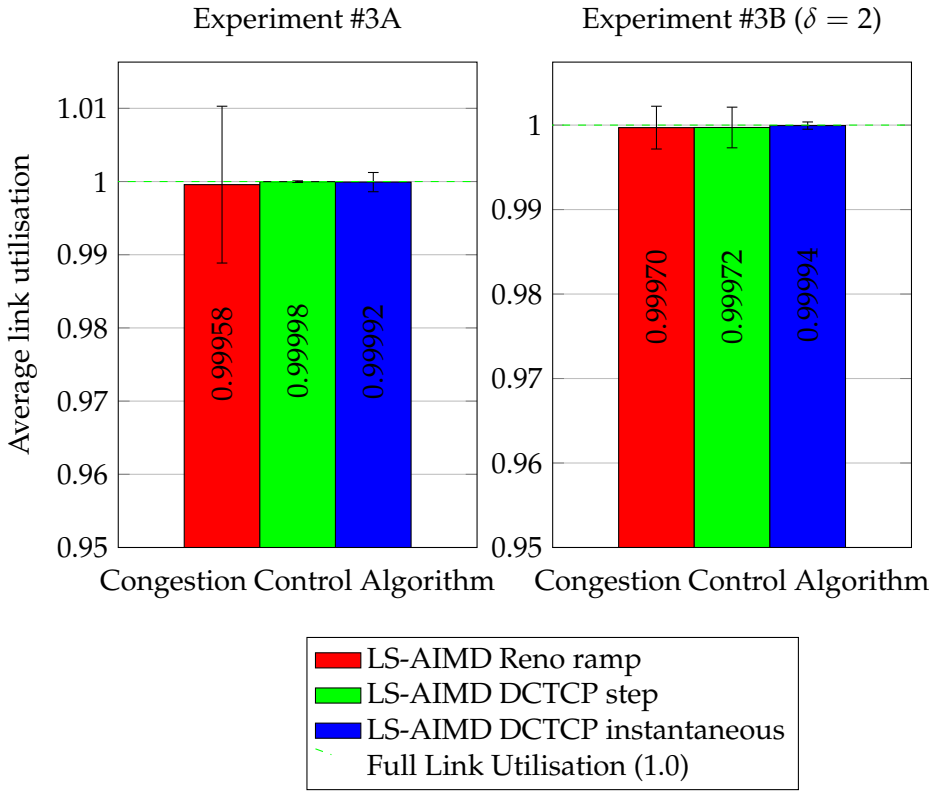


Figure 8.21: Experiment #3: Link Utilisation dynamics

marginal lack of stability. We can not make any conclusion from these results except that all configurations fulfil the requirement of full utilisation.

8.2.2 Queueing Delay

Next, we investigate the queue occupancy of RED to see what the typical wait time for a packet is when entering the bottleneck.

In Figure 8.22, we see the average queueing delay and its corresponding deviation. We, first, notice how well the ramp and instantaneous configuration perform in keeping the average a queueing delay a little bit above the middle of the minimum and maximum parameters of RED. However, the instantaneous configuration yields a smoother queue on average, deviation of 59 vs 107 microseconds, because of its less drastically reduction phase. We get an insight into what our earlier observations of synchronisation mean for the step configuration. The step configuration is not able to keep a stable queue, deviation of 176 microseconds, as long as the receiver returns an immediate acknowledgement. On the right-hand side, we see how the use of stretch acknowledgements get rid off the synchronisation. When step configuration is kept desynchronised, it not only performs as good, deviation of 107 microseconds, as the instantaneous configuration but additionally keeps a lower queue occupancy on average. When comparing the rest of the configurations between the use of stretch acknowledgement or not, we see a small decrease in the queue occupancy,

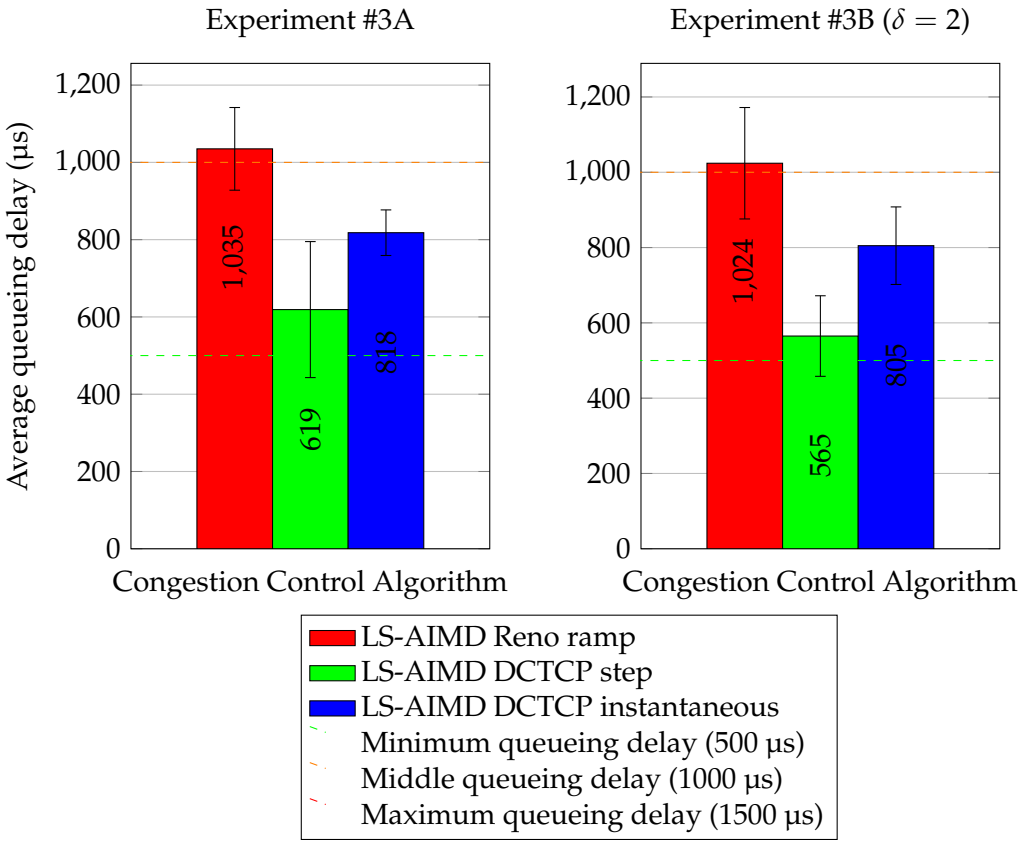


Figure 8.22: Experiment #3: Queue dynamics

but a significant higher deviation.

We conclude that the use of stretch acknowledgements does not cause any significant penalty for all configuration, but further analysis is needed to see how a more substantial stretch acknowledgement factor works out.

8.2.3 Smoothed Round Trip Time (SRTT)

We should now look at how each configuration traverses the bottleneck and investigate how the queue impacts the RTT of the traversing traffic. We should, as earlier, try to justify the use of stretch acknowledgements. However, since the SRTT of the stretch acknowledgement enabled experiment includes the penalty of the receiver, would it make more sense to compare the SRTT pace against the other experiments SRTT. Any significant difference in our measurements should then be able to tell us about an improper calculation of the SRTT pace. We expect the experiment with stretch acknowledgement enabled to perform with a slightly lower average SRTT but be less stable based on our findings from the queue occupancy dynamics of the bottleneck.

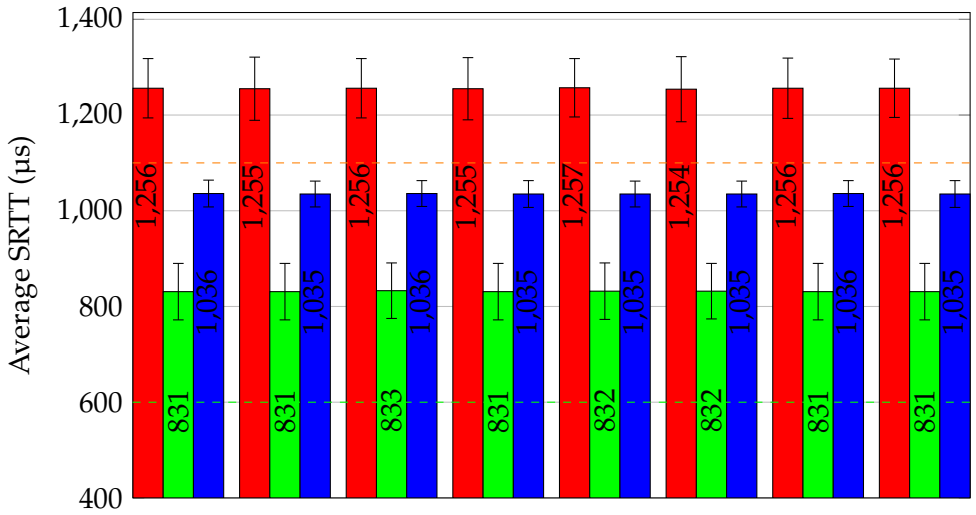
In Figure 8.23, we see that the SRTT across all senders in both experiments is of identical length and that the deviation spreads equally between the competitors. However, the first two flows of the bottom experiment are not as stable as the rest. We learn that this shewed result comes from the fact that both flows observed a single episode of spurious retransmission. It is essential to notice that the queueing delay stands for about 80% of the SRTT in both of our experiments, so maintaining a deeper queue is not the appropriate solution to deal with the submss regime.

We see that according to our expectation, the experiment with stretch acknowledgements has more fluctuations in the RTT, but the difference is a lot higher than expected. The average estimate is also not as low as predicted. We believe that by improving the accuracy of the calculation of SRTT pace should minimise the difference and equalise the two experiments.

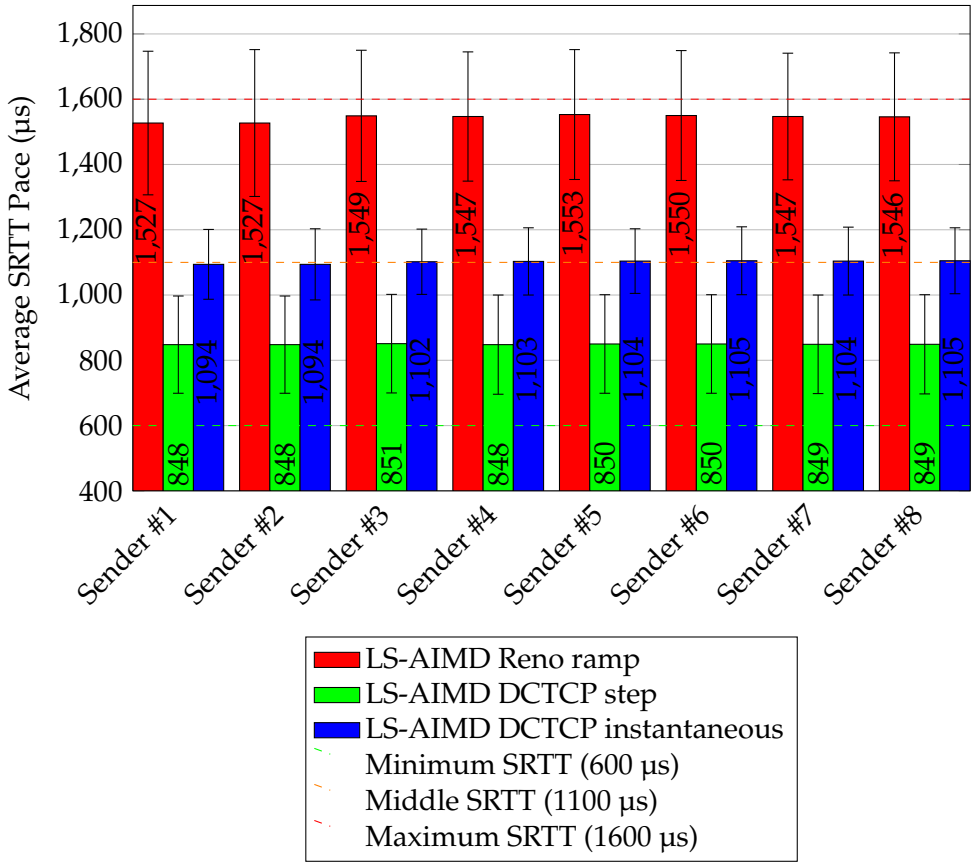
8.2.4 Marking Rate

We will now inspect the marking rate of RED to verify that the state of marking saturation is out of reach from all configurations. We also want to crosscheck that our algorithms remains stable even with the use of stretch acknowledgements.

In Figure 8.24, we see the average marking probability for the individual sender over a 10 ms interval. RED and LS-AIMD interact very nicely; notice how all senders see one unified marking probability and how the deviation is extremely stable between the competitors of the bottleneck. Remarkably, we do not see any significant change when we rerun the same experiment with stretch acknowledgements. The reason why we no longer observe the spurious retransmission to have any significant impact on the results is due to the 10 ms smoothing. We believe that by measuring the stability of the marking probability over approximately 10 RTTs is more than

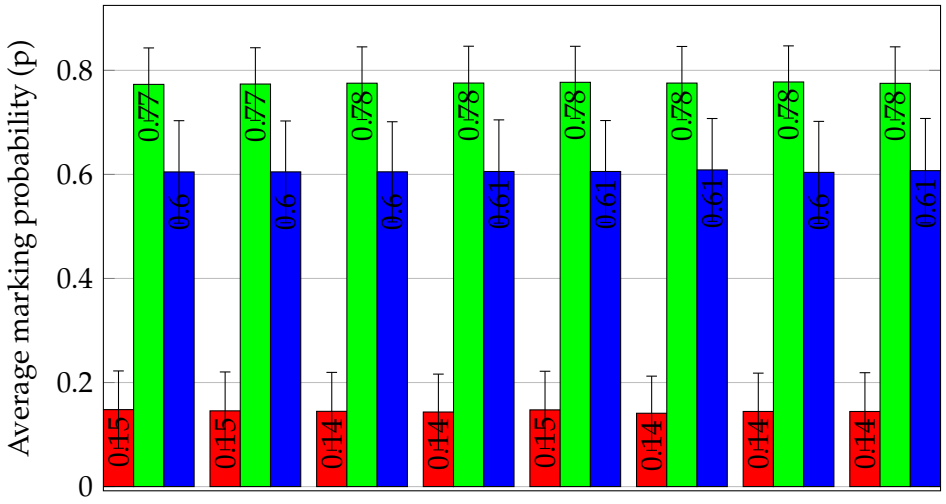


(a) Experiment #3A

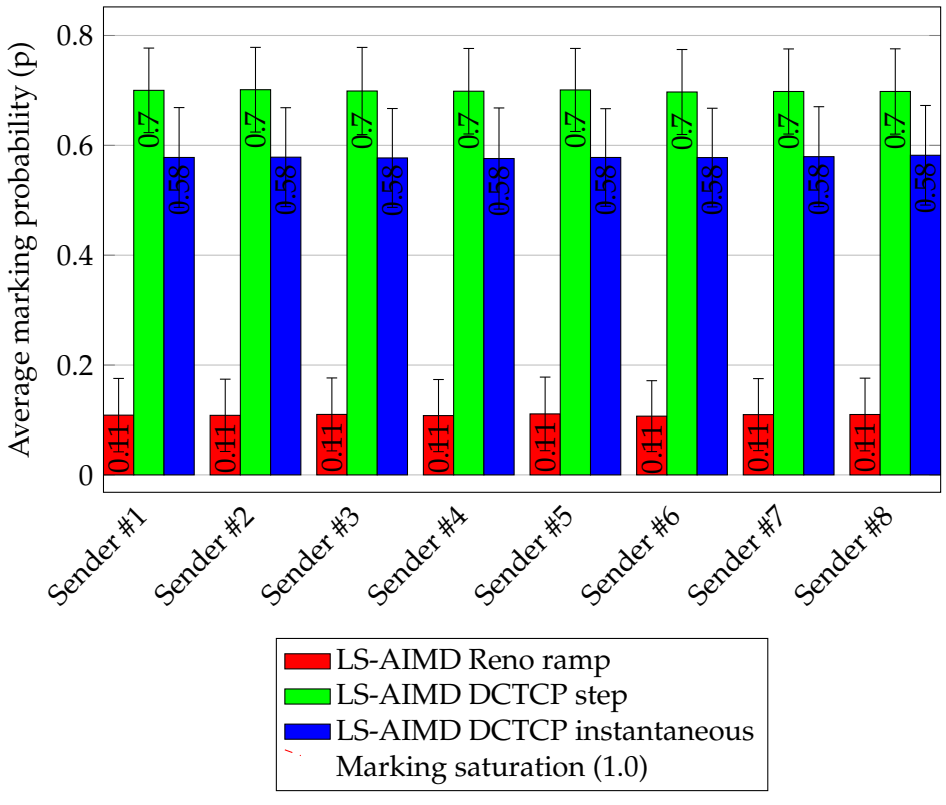


(b) Experiment #3B ($\delta = 2$)

Figure 8.23: Experiment #3: SRTT dynamics



(a) Experiment #3A



(b) Experiment #3B ($\delta = 2$)

Figure 8.24: Experiment #3: Marking rate dynamics

sufficient to show the essential characteristics of the individual configuration.

8.2.5 Additive Increase

Going over to the evaluation of additive increase, we now look at the stability of the additive increase phase. We expect the configuration which the lowest SRTT to keep a slower additive increase phase and vice-versa. However, since the add function is a slow-growing function, do we not expect a significant difference.

In Figure 8.25, starting with the upper experiment, we observe as expected a slower additive increase phase from the step configuration. We also see that it does fluctuate on average by just 12%. Surprisingly, we see that the ramp configuration keep a slower additive increase phase on average than the instantaneous configuration and that both configurations maintain deviation of the same degree. We can explain this by remembering that Reno does rate halving which means that the sawtooth falls into a much lower ssthresh value, and has enough of a drop to give the ramp configuration a slower additive increase phase.

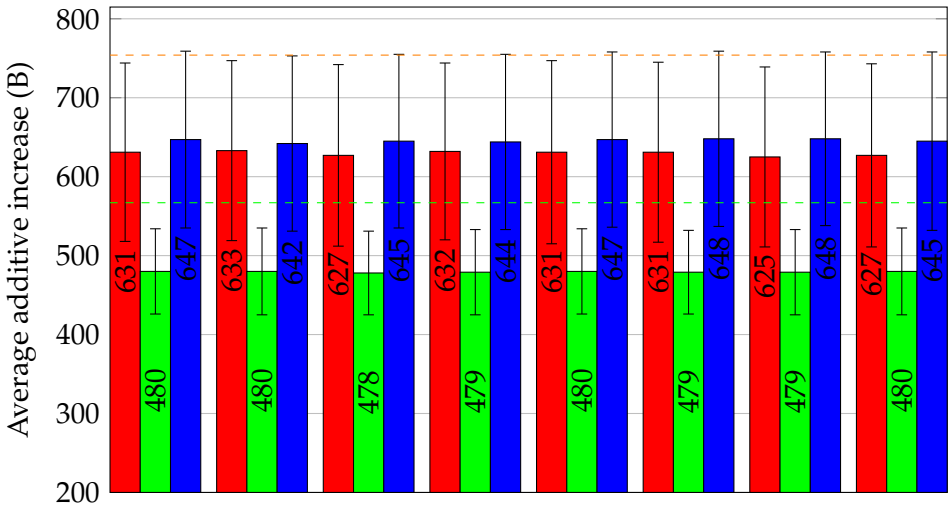
8.2.6 Throughput

Finally, inspecting the last metric, we now investigate whatever the capacity between the senders shares evenly, and we further look into how the stability of this shareout.

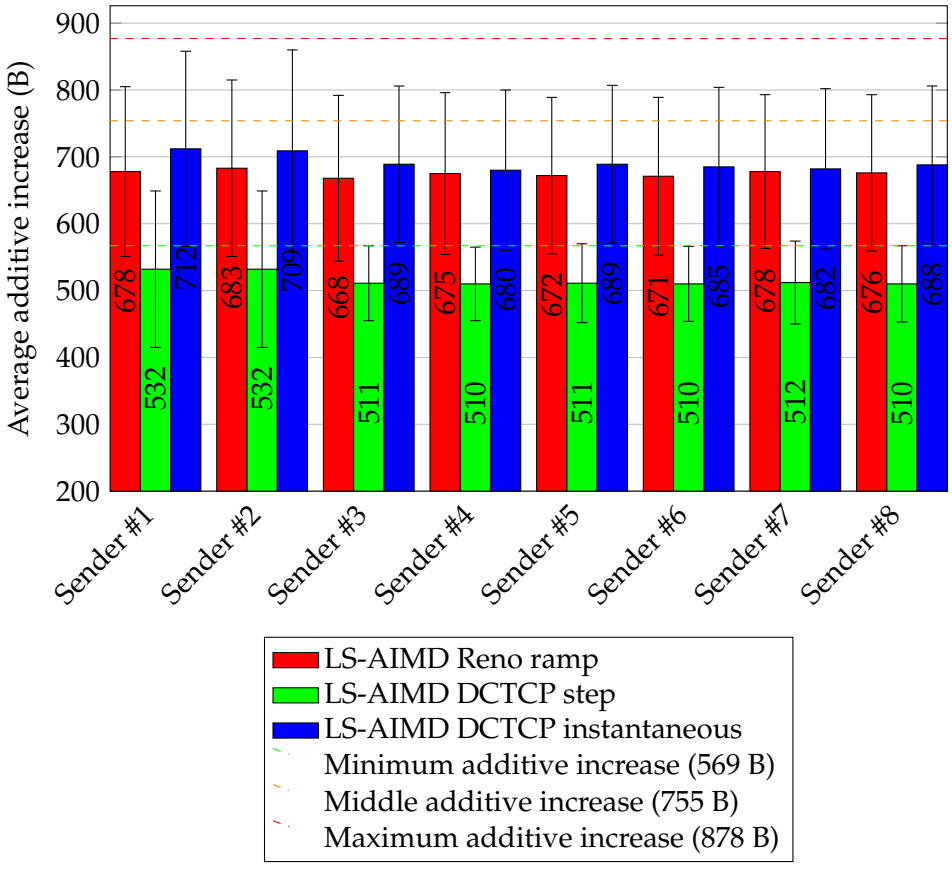
In Figure 8.26, we see that all configurations in upper experiments have a stable share of the capacity among the competitors. The ramp configuration performs worst with a deviation of 22.3%, whereas the step and instantaneous configuration have a small deviation of 5.6% and 16.3% respectively. Going to the lower experiment, we do not quite see the same shareout as the first two senders, as earlier, shew the balance for a more extended period of the experiment. The ramp configuration still performs worst, and we now observe deviation ranging from 25% to as significant as 50%. Similarly, we see that the step and instantaneous configuration now lies in a range of $6.4 - 54\%$ and $18.5 - 55\%$ respectively.

8.3 Exhaustive & Scalability Test

We should now have a rough idea of how well LS-AIMD perform under a variety of network topologies. The last thing we want to do before benchmarking it against other conventional congestion control algorithms is to see how far we can push LS-AIMD before it starts to misbehave. We would then know about any scalability issues which constraints the performance of our algorithm when pushed to the extreme. We resort at evaluating all configuration as before, but now under a variety set of base RTTs. The idea is to understand the performance of LS-AIMD better when moving to a lower transmission rate per RTT. We increase the bitrate from

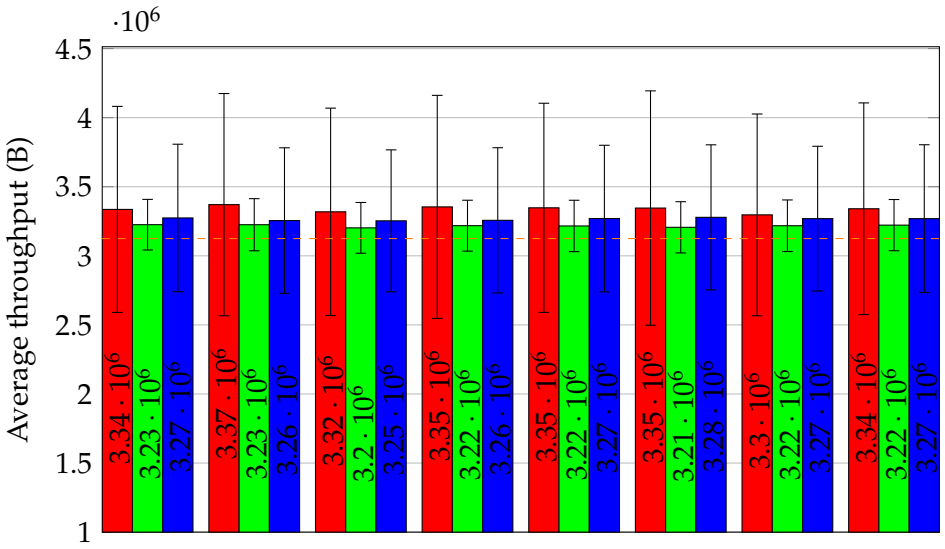


(a) Experiment #3A

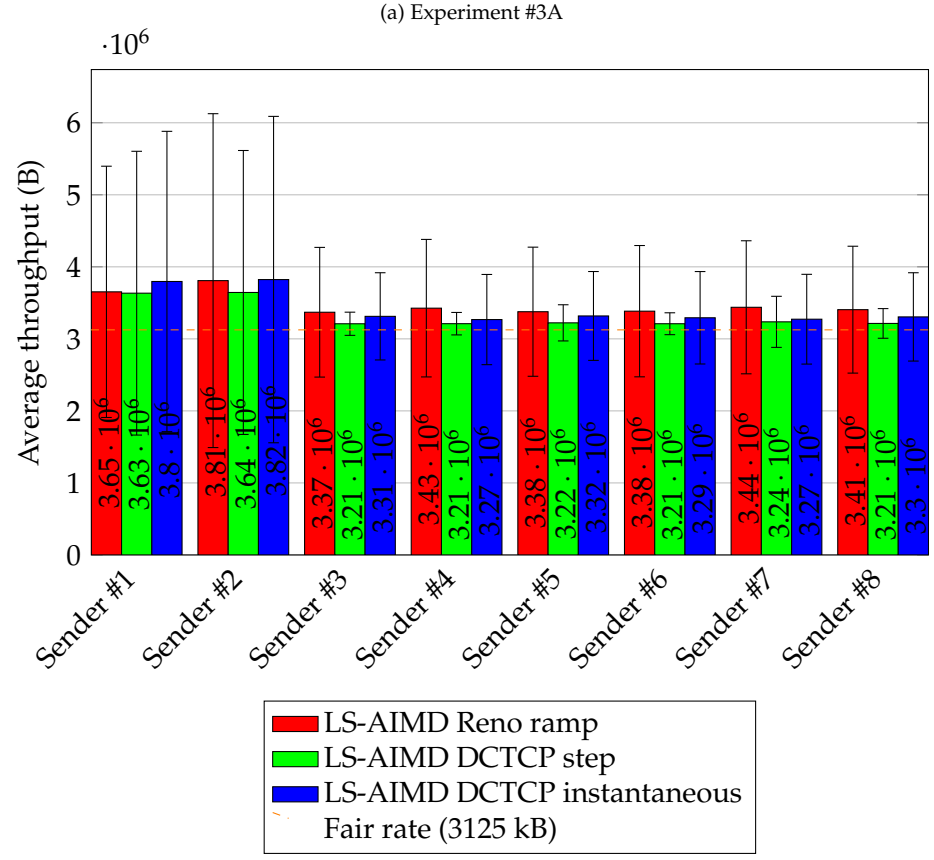


(b) Experiment #3B ($\delta = 2$)

Figure 8.25: Experiment #3: Additive increase dynamics



(a) Experiment #3A



(b) Experiment #3B ($\delta = 2$)

Figure 8.26: Experiment #3: Throughput dynamics

200 Mbps to 600 Mbps and at the same time double the number of senders, from 8 to 16, at the bottleneck. The goal of using a higher bitrate is to be able to set the marking threshold of RED even lower. We now aim for a queue occupancy of merely 250 microseconds and a base RTT ranging from 250 to 1000 microseconds.

We will now also evaluate the performance of LS-AIMD under MTU sizes of 1500 and 9000 bytes. A higher MTU should let us know if LS-AIMD is capable of attaining a low queue over the very same set of base RTTs. We follow our guideline in choosing an appropriate growth constant for the additive increase for both MTU sizes. More precisely, we choose k0 to be 256 for an MTU of 1500 bytes, and k0 equals 1024 for jumbo frames.

We will also test for two sets of scale factors (C) since we are now pushing our senders into holding a congestion window of less than one segment. We test for both a very low and a very high C corresponding to 0.78125% (2^{-7}) and 100% respectively. There is no good reason for selecting such a low C other than to keep the low and high value segregated by a large enough margin such that we can see a clear and distinguishable pattern between the two.

8.3.1 Link Utilisation

Same as earlier, it is essential to verify that the link never goes underutilised. A loss in link utilisation should not be a common occurrence in our experiments. We consider a configuration with lower utilisation to be worse than any other configuration with a higher link utilisation. Therefore, a loss in link utilisation is enough to disqualify the configuration based solely on this metric.

In Figure 8.27, we see four sets of experiments based on the combination of MTU and C. On top, we find the standard ethernet frame, and at the bottom, we have the same experiments just with jumbo frames. We notice how the link utilisation varies because of the size of C and at the same time gets affected by the frame size. We see that a small C is a better fit for lower base RTT because a lower transmission rate makes the sender build large sawtooth if C is also high, and a large sawtooth has no desirable properties as our previous results indicate. We also see that the opposite is also valid as the use of a higher C yield better utilisation for higher base RTTs. These properties can be best seen in the jumbo frame experiments because a larger frame size yields a lower congestion window for the same bitrate and base RTT. We additionally see a trend where a lower C on a low base RTT and high C on a high base RTT yield more stable utilisation of the link. We need to look at the queue of the bottleneck to better understand the cause of such a varying deviation of the link utilisation across the conducted experiments.

8.3.2 Queueing Delay

We will now evaluate the queueing delay part of this experiment. We compare the average link utilisation against the queueing delay to form

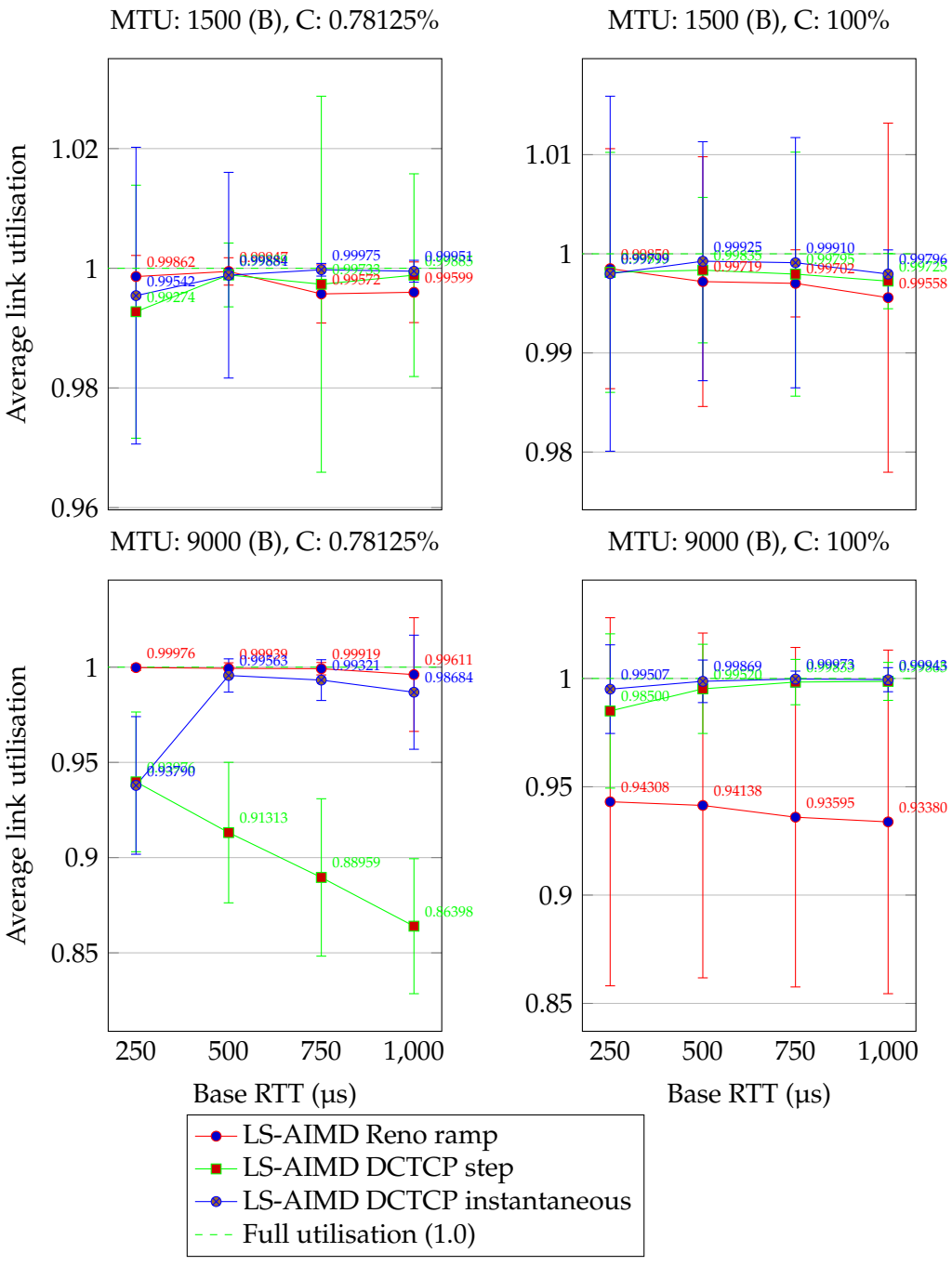


Figure 8.27: Experiment #4: Link utilisation dynamics

a better picture of what causes the link to go underutilised.

In Figure 8.28, we observe three patterns across the different values of MTU and C. We, first, see that the use of MTU of 1500 and C equals to 100% achieves the most optimal result. Meaning, the average queueing delay is kept within the desireable queue length of RED, and all configuration seems to have almost no impact between the base RTTs. Additionally, we notice that the deviation keeps on declining as the base RTT falls to some lower values. This property is intended as LS-AIMD take in use a logarithmically scaled additive increase phase which depends on the transmission rate of the sender; a lower transmission rate per RTT equates to a slower probing per RTT.

We do not get results of similar quality when going to C equals 0.78125% for the same experiment. We immediately see that all configurations happen to probe below the minimum queueing delay set for this experiment. This low queue occupancy explains our earlier observation of a steady decline in link utilisation. However, we observe that the queue occupancy of the bottleneck seems now to be more stable.

When we move to experiments conducted with larger frame sizes, is it clear that C at 100% is now way too much. The queue occupancy of the bottleneck seems to be on average above the maximum queue length throughout this experiment. Initially, this might indicate contradicting results, but when one remembers that the congestion window bases itself on the frame size is it evident that a prolonged transmission rate per RTT cause all configurations to misbehave widely. Reasoning, why some configurations managed to underutilise the pipe has to do with rapid fluctuations in the queue occupancy causing massive spikes in the queue periodically. Notice how the length of the spikes only gets deeper and deeper when going for lower base RTT. This fluctuation comes as a result of an additive increase, which is very low per RTT but is now magnitude larger because the sender has to apply the additive increase phase over more and more RTTs. Nonetheless, the theory of applying more additive increase phase for a sender pacing over a more extended period is to ensure that two competitors which different base RTT still converge. The length of the sawtooth is decided by the sender who travels with the most extended base RTT.

Finally, looking at the last viable permutation, we see that the queue occupancy of the bottleneck is no longer as high as earlier. While we still do not see optimal results for jumbo frames, do we now know that it is possible to select one optimal C which would give results of similar quality as the first permutation. However, this increases the complexity as we are now required to use two very different C because of a change in the transmission rate per RTT.

8.3.3 Smoothed Round Trip Time (SRTT)

We have seen that by making TCP cooperate with the AQM under a lower transmission rate, we can maintain a very stable queue over shallow base RTTs. However, this is just a metric measured at the AQM and do not tell us

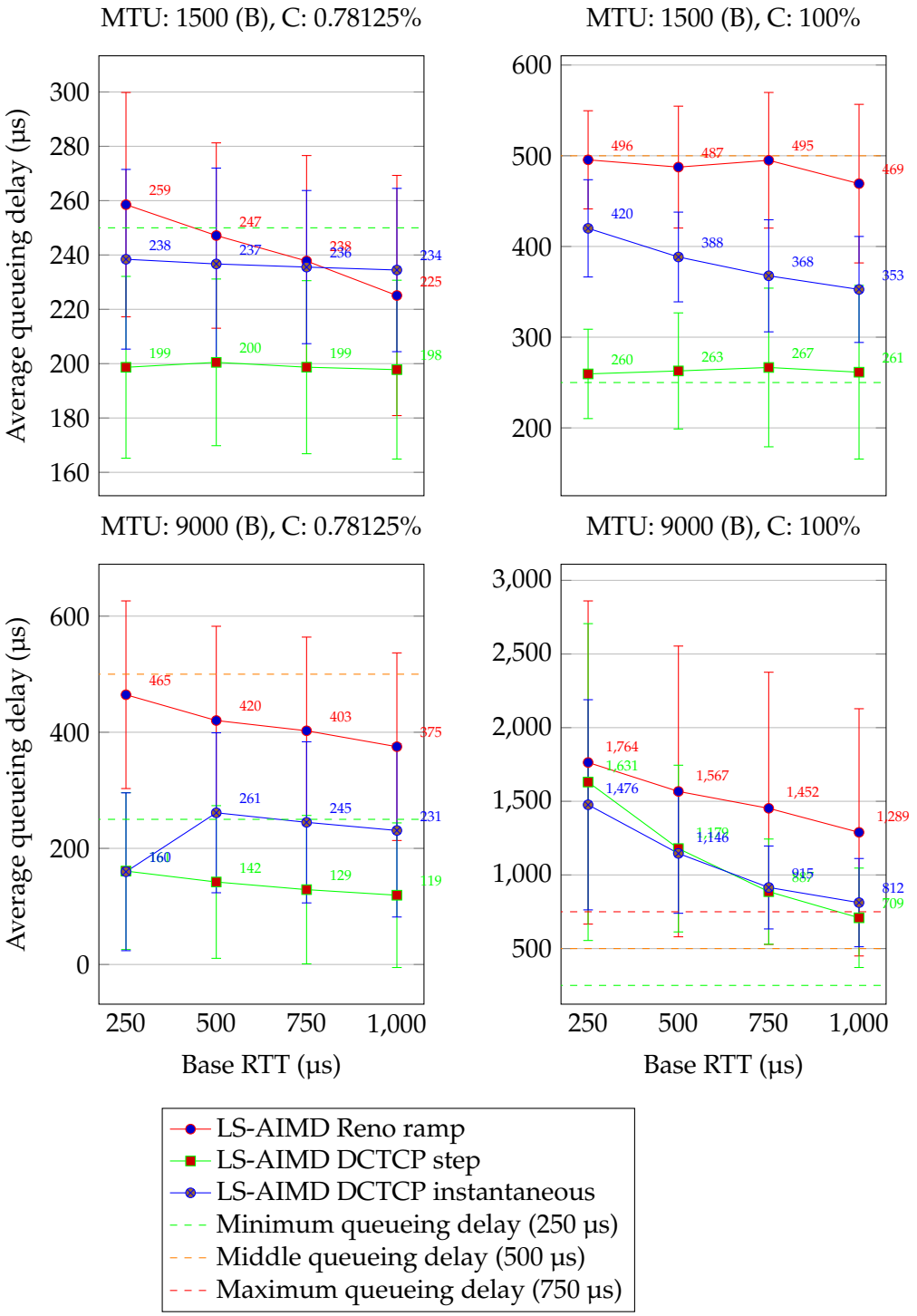


Figure 8.28: Experiment #4: Queue dynamics

how significant the RTT of the traversing traffic is. We, therefore, as earlier slay the doubt of whatever it is working or not by analysing the smoothed RTT of our senders.

In Figure 8.29, a look a the SRTT over the same set of experiments, we see that the RTT closely follow the queueing delay of the bottleneck. We also notice that since SRTT is as its name indicates a smoothed metric is its deviation more stable than the instantaneous queueing delay of the bottleneck. We also now start to see whats otherwise is not the usual case; a majority of the RTT is now something other than the queue occupancy of the bottleneck. Meaning, even at base RTT of 250 microseconds the queueing occupancy of the bottleneck stands for 40% of the total RTT seen by the sender. The only reason why we do not see a severe mismatch between the SRTT and the queueing delay comes from the fact that we can operate under a shallow base RTT. Other delays, such as propagation, transmission and processing delays, are not given a chance to undertake the now heavily reduced queueing delay. Adding more bitrate to this equation does not change anything as we can then maintain a proportional lower queueing delay with the now reduced transmission delay; so they balance each other out. Neither or does a lower possible propagation delay serve any difficulty as LS-AIMD has no intention in building a queue just because of a lower base RTT. However, we need to figure out a way to control C correctly all the time such that we can do exactly that and keep a shallow queue no matter the requested transmission rate per RTT.

8.3.4 Marking Rate

Let us now investigate how RED has done in its job at keeping the queueing delay within the parameter set for the experiments. We, as earlier, measure the average marking rate of RED over a 10 ms duration.

In Figure 8.30, we see the same layout as earlier of experiments and observe an experiment which draws our attention; an experiment seems to reach marking saturation periodically. This experiment is no other than the experiment with jumbo frames where C was 100%. This result does not come as a surprise as this experiment is indeed the same experiment which built a longer queue. Therefore, this proves that RED did do all it could to demand the competing traffic to slow down, but was unable to achieve the desired queue length because senders had no means to back off. The same experiment with C set to 0.78125% shows no sign of unresponsive traffic and is merely inducing any congestion, which implies that we should have set C to some higher value. Nonetheless, this experiment was not able to build a sufficient queue to maintain full utilisation.

Finally, we verify the optimal experiment where MTU was 1500 bytes, and C equals 100%. Our result indicates that the probability of being marked increases as base RTT becomes lower, and as expected the ramp, instantaneous and step configuration maintains marking probability from very low, moderate to extremely high levels respectively. Additionally, we learn that C impacts severely on how fast the marking probability grows towards holding a lower transmission rate. Nonetheless, we should refrain

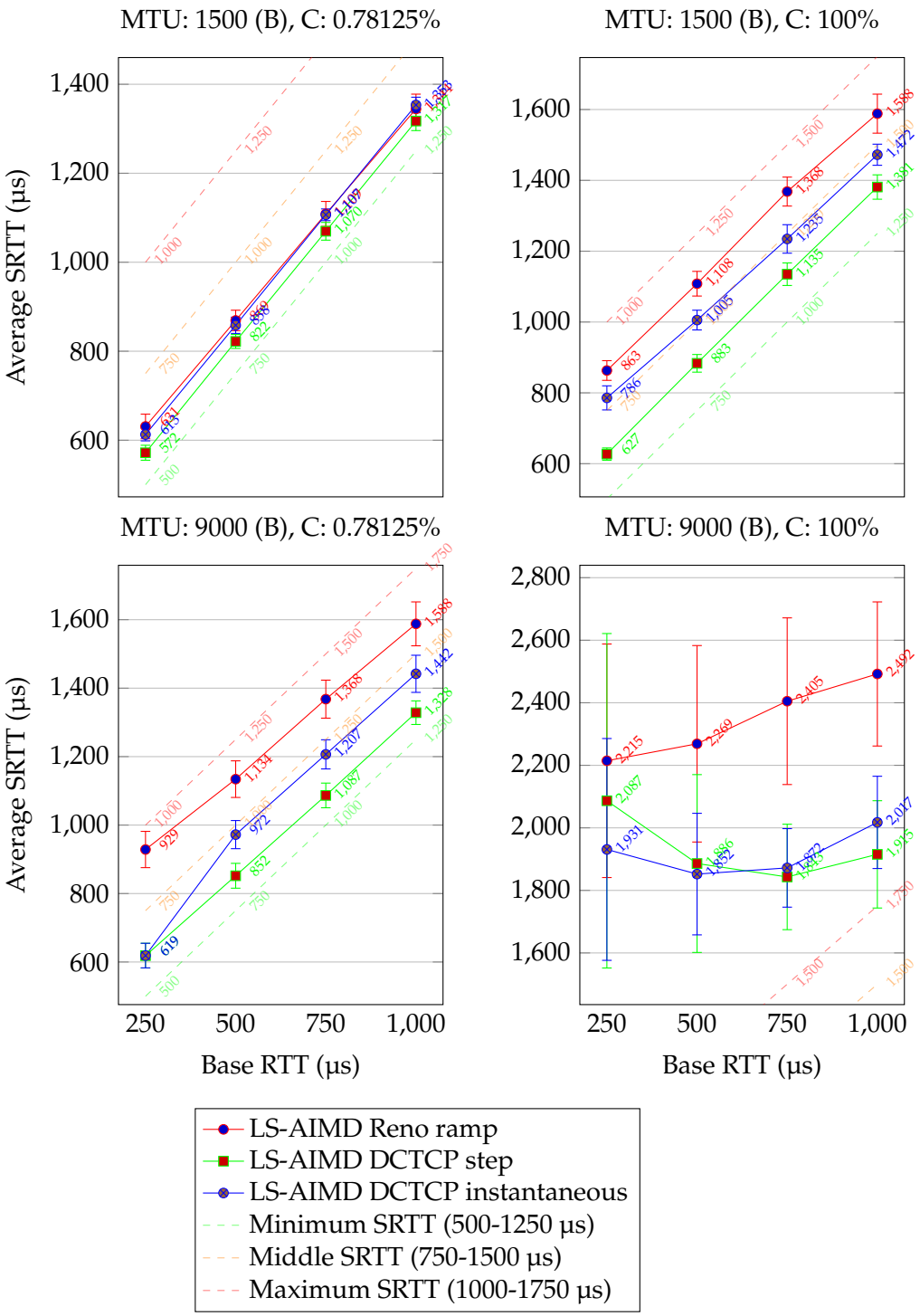


Figure 8.29: Experiment #4: SRTT dynamics

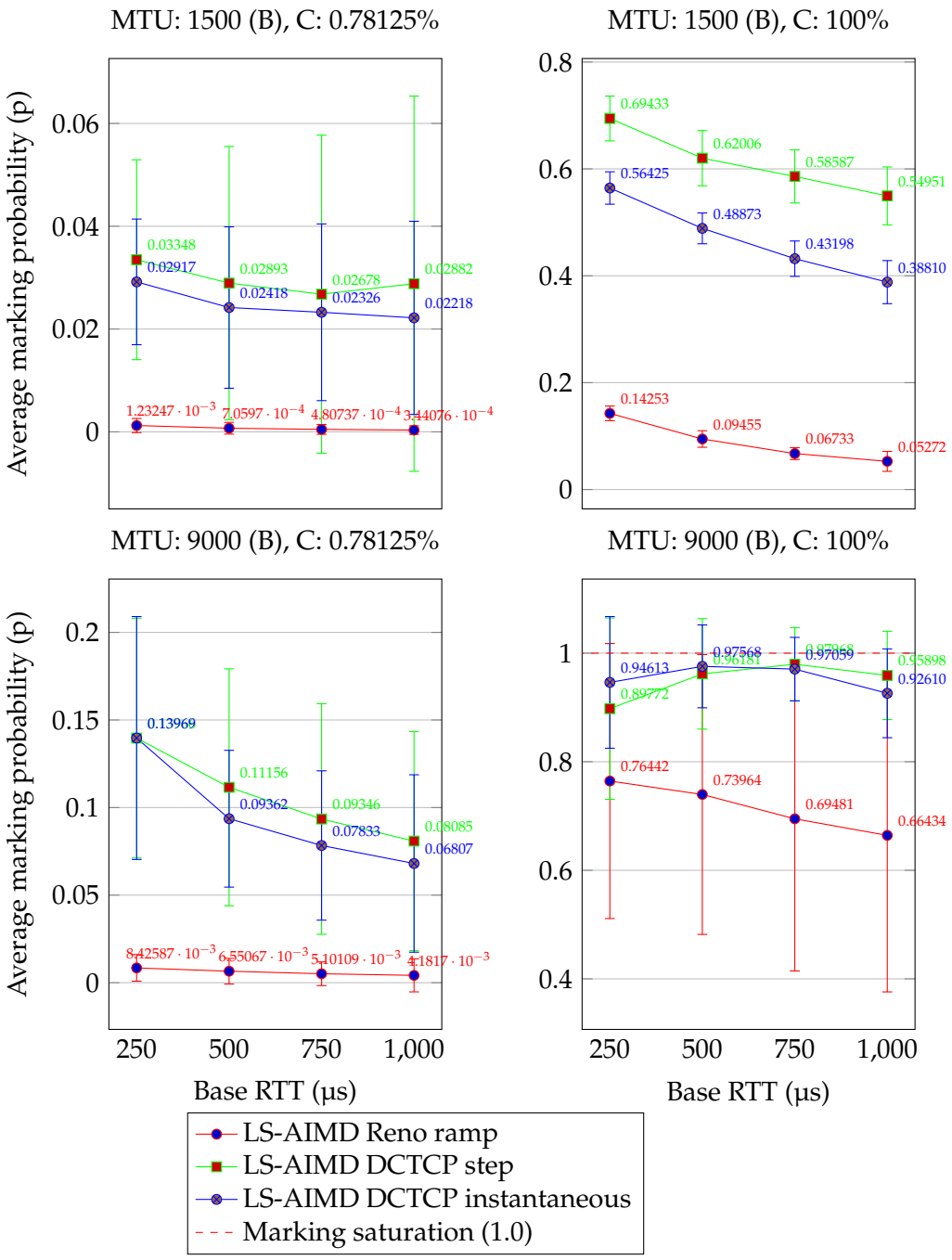


Figure 8.30: Experiment #4: Marking dynamics

from using a C high enough to cause marking saturation, but at the same time not a so low value that we lose up the utilisation of the link.

8.3.5 Additive Increase

Up until now, we have only prioritised the delay aspect of our experiments. We shall now investigate how good LS-AIMD perform in keeping additive increase phase of good value throughout our experiments.

In Figure 8.31, we look at the same set of plots, but for this metric now with indicators to show where we want our additive increase to stay for the different base RTT. These values are simply derived from the BDP of our experiments and fed straight into the add function with the appropriate growth constants and C. Although, we have one small remark which is that since we did not log the mantissa part of the additive increase values do we lack the sufficient information to get a very accurate reading for those experiments with low C. The value, therefore, has to be a more or less an approximation of the actual value.

We see that the additive increase grows very close to the desired parameters of the individual experiment. However, notice how the experiment with MTU of 9000 bytes and C set to 100% struggle to lie within the desired parameters of the experiment. We see that the line flattens out because the sender applies the additive increase phase more often than the multiplicative decrease phase. Performing multiple increase phases are needed when pacing over more extended periods, but doing the same with the multiplicative decrease phase is not desirable as we would then have large sawtooth in the queue. The outcome is a sender which is unable to back off since the growth per RTT is magnitude higher than what gets taken away. We say that the sender has become unresponsive and refuses to cooperate with the AQM. The AQM should have evicted the packets of the unresponsive traffic such that they are held responsible for the action they cause on others. A such AQM would have prevented the unresponsive traffic from building a longer queue than the desired range. However, an AQM only need to intervene when proper warnings have been given, through marking, and the traversing traffic fails to back off on its own.

8.3.6 Throughput

Finally, let us not forget to investigate how well the capacity shares among the competitors. We evaluate only the experiment with the lowest base RTT as we have already seen how well LS-AIMD converge under a high transmission rate. We still evaluate this experiment over both a changing MTU and C.

In Figure 8.32, we start with MTU of 1500 bytes. The low C is on the top while the other one is at the bottom. We should now be able to see how even the capacity shareout is between a set of senders with different RTT. Nonetheless, a sender who paces grows its RTT to match the desired transmission rate, so there might be senders not pacing. We, therefore, need to test whatever these competitors can reach equilibrium.

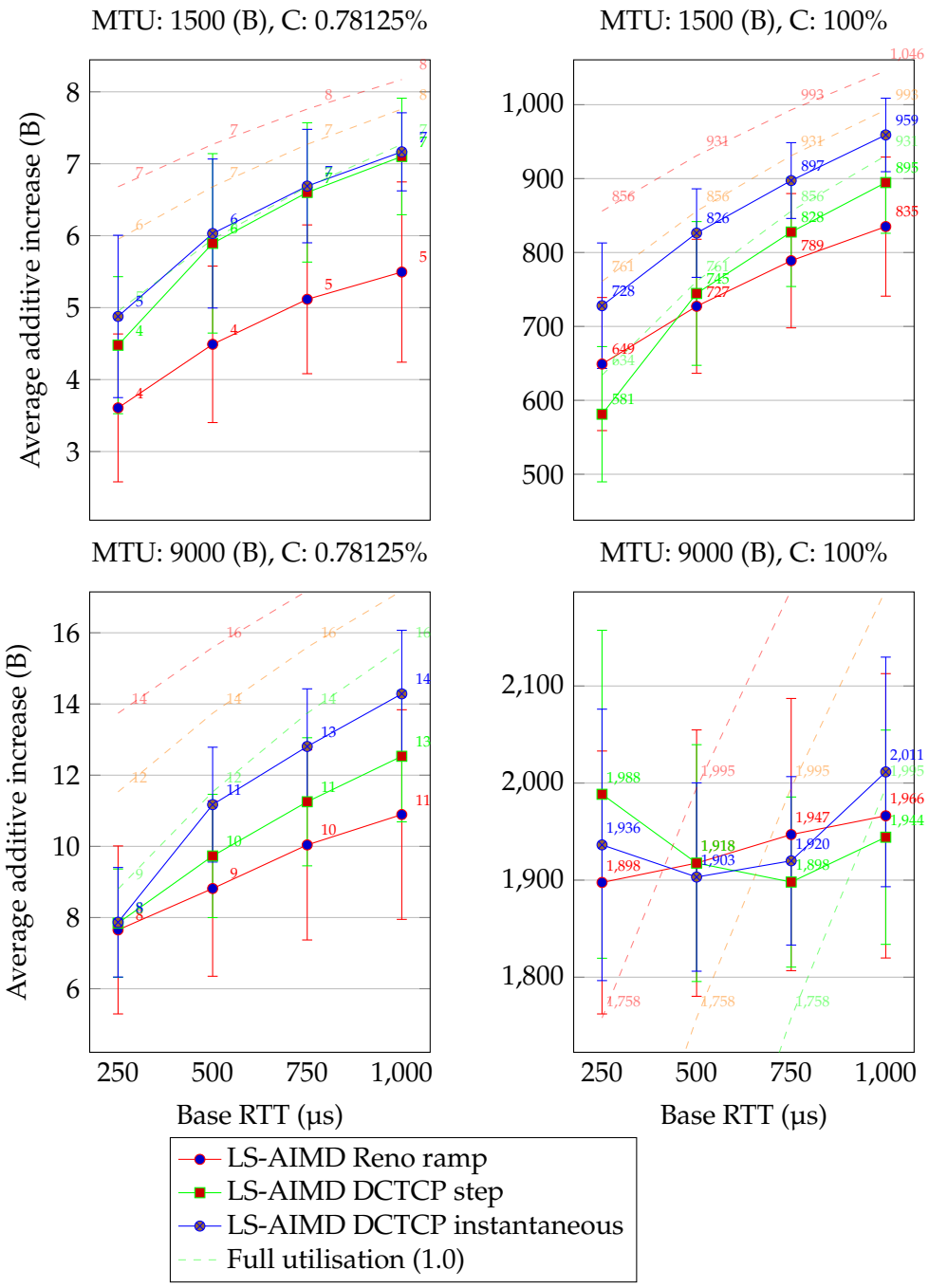
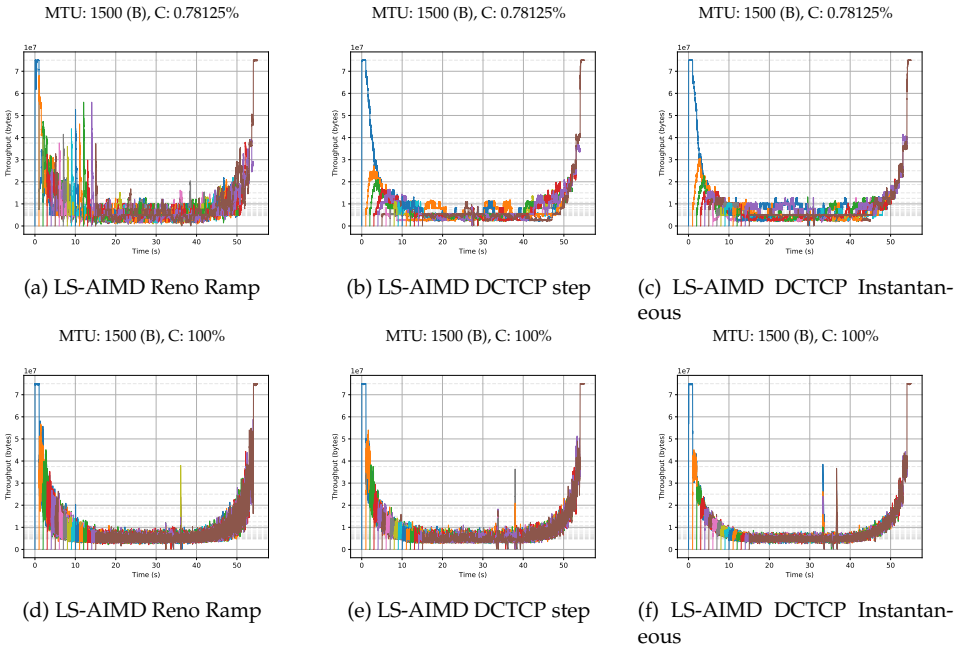


Figure 8.31: Experiment #4: Additive increase dynamics



(a) LS-AIMD Reno Ramp (b) LS-AIMD DCTCP step (c) LS-AIMD DCTCP Instantaneous

(d) LS-AIMD Reno Ramp (e) LS-AIMD DCTCP step (f) LS-AIMD DCTCP Instantaneous

Figure 8.32: Experiment #4A: Throughput

As expected, the very same experiment with C equals to 100%, which had the most stable queue now has the best convergence property, i.e. the competitors at the bottleneck have reached equilibrium. We also notice that each configuration almost follows the same pattern as earlier, where the step configuration had the most stable capacity shareout, ramp the worst and instantaneous fell in-between. However, we expected the step configuration to be more stable than the other configurations, but this is not the case as the DCTCP step configuration relies on a fine granularity marking signal from the AQM to smooth out the congestion level. Step configuration struggles as less than one measurement are taken per RTT, which translates into either 0% or 100% marking per RTT. The step configuration is, therefore, unable to smooth the congestion level properly as it lacks the proper information to achieve this.

Additionally, we can confirm this by reinvestigating the average marking probability of this experiment. We indeed observe that the step configuration keep a much higher marking rate, which explains why it is struggling, and it does not help that its probability grows faster than the other configurations. However, we can lower C to increase the accuracy of the congestion signal since a lower C translate into less fluctuation in the queue of the bottleneck. A queue with fewer spikes is much prefered as we can then achieve a more smoothed signal, but the consequence is of course that the sender is now slower to adapt to a significant change in the signalling.

For example, consider a scenario where a sink is full of water, simulating a draining queue, and let us say that when it reaches above a specific size, the threshold of the AQM, it is considered too much. Now further consider that we have a person, simulating a competitor of the

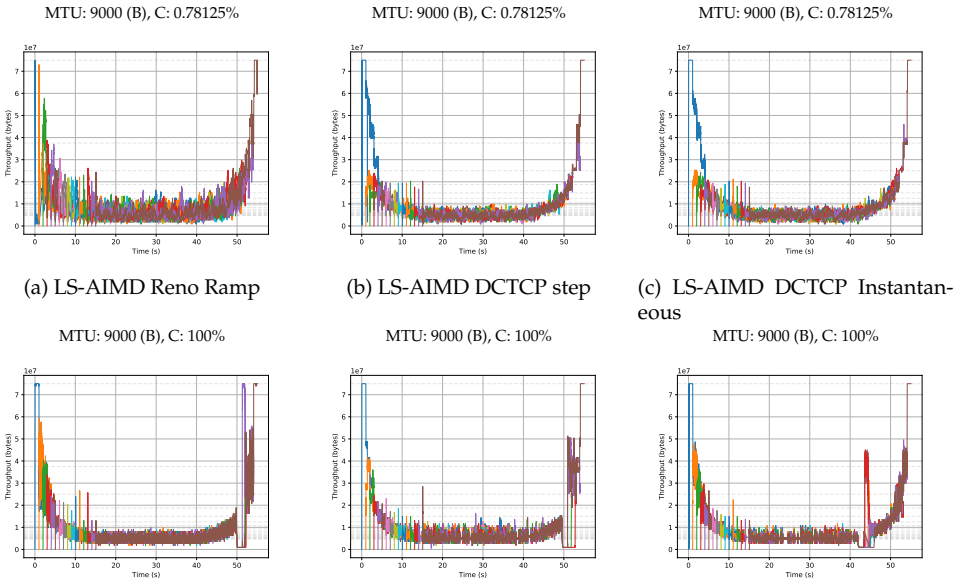
bottleneck, emptying water bottles into the sink. Let us say one third-person, simulating a receiver, spectate this experiment and is supposed to signal congestion within a given granularity, forming the RTT between the sender and receiver. Let us now take the more straightforward example first, emptying a bottle with fast feedback requires the person to only add a slight amount of water to the sink per RTT to maintain a shallow queue at the bottleneck.

Now let us make the matter a little more complicated, it is much prefered to empty larger bottles all at once, than empty smaller bottles one by one; think of the size of the bottle as the payload of a network packet. However, this requires the AQM to have at least two bottle size of water such that there is enough time for the sender to reach the bottleneck before it goes empty. The congestion window is thus the number of bottles that one person is allowed to empty within an RTT. Let us now say we have a few more persons emptying their bottles and in total inducing high enough congestion to makes all senders committed at keeping fewer than one bottle inflight per RTT. A person, of course, still want to empty their whole bottle if possible, so each person starts pacing, extending their RTT, such that not all persons empty their bottle all at once. A person is, of course, only first made aware of congestion after they have already emptied their bottle, and this feedback goes to only those who actively emptied their bottle in the previous RTT.

However, we have up until now ignored a vital detail, which is how fast should each sender increase their transmission rate; obviously, a person who waits to empty his bottle has the right to next time empty his bottle sooner than the rest who did not wait, i.e. forming a queue. Now recall that in the step configuration, the likelihood for a transmission to end with a mark depends on only the instantaneous queue length. Now if competitors of the bottleneck have to increase their send rate rapidly, then how do they know when they have gone too far? We can say that a person weakens its knowledge of congestion by waiting long times between its transmissions. Therefore, waiting for a long duration does not necessarily mean that it is now safe to do a rapid increase in the congestion window. If the person instead only slightly reduced its pacing timer, then the meaning of the congestion signal is much more valuable. However, to compete against other senders who have yet to know the precedence of congestion must a rapid increase in the congestion window sometimes be forced to reach a state of equilibrium.

It does not come as a surprise that all configurations in the other experiment, where C is a lower value, struggles to maintain the state of equilibrium. The competitors are not inducing enough congestion to maintain the proper fair share of the capacity over short periods.

In Figure 8.33, we see the same set of configurations over the now higher MTU. Jumbo frames bring the required transmission rate per RTT to shallow levels. This experiment is a stress test in an attempt to break LS-AIMD such that it malfunction. Starting with C equals 100%, we notice how all configurations appear to go silence for serval seconds, this episode of loss happens at the end stage of the experiment. Our explanation is that



(a) LS-AIMD Reno Ramp

(b) LS-AIMD DCTCP step

(c) LS-AIMD DCTCP Instantaneous

(d) LS-AIMD Reno Ramp

(e) LS-AIMD DCTCP step

(f) LS-AIMD DCTCP Instantaneous

Figure 8.33: Experiment #4E: Throughput

since LS-AIMD became unresponsive and had a long queue until the last period of the experiment, which caused a shift in the queue, resulting in a majority of the queue occupancy to disappear. This cause TCP to perform spurious retransmissions.

We notice how more stable the capacity sharing is between the competitors of DCTCP configurations when C is lower. This outcome has the same explanation as earlier; a smaller C yields a finer congestion signalling when the transmission rate per RTT is also low.

8.4 Summary

We have, in this chapter, evaluated LS-AIMD under a variety of experiments to verify that our design and implementation does indeed work, but not without some scalability issues we discovered late in our evaluation period which makes LS-AIMD misbehave when pushed to the extreme. However, we have found the cause of this to under fold, and we were additionally able to scale in this extreme environment, but not without making the overall algorithm unable to scale for the other environment.

We have, therefore, come to a conclusion which states that we should be able to scale well for both environments if we manage to adapt C to the desireable value depending on the transmission rate. However, this is not such an easy problem to solve as it requires one to find the optimal C such that all competitors remain in the state of equilibrium and at the same time does not go unresponsive.

Now that we fully understand how well LS-AIMD performs do we want to continue our evaluation by comparing it against the standard Reno

and DCTCP.

Chapter 9

Results

We shall now see how well the standard Reno and DCTCP match our results from the previous chapter. We rerun the same experiment over the same network and configurations. We preserve all, but the congestion control algorithm from our evaluation. Any change in our results comes from the performance of the individual congestion control algorithm. It is also important to emphasise that both Reno and DCTCP variants run over the very same configurations of RED. No one can, thus, blame a change in the results on the AQM or the network. We do, however, append the same modifications as earlier to DCTCP to increase precision in its calculation of alpha. These modifications are needed to ensure that DCTCP performs well also under low congestion levels.

We evaluate experiments based on the same routine as last time. First, we do a simple convergence test to verify that both Reno or DCTCP competitors perform well from the start. Next, we repeat the experiment #3 to inspect results for any form of malfunctions. We continue the evaluation by testing Reno and DCTCP in scalability over both the standard ethernet frame size and jumbo frames in a network with high to low base RTTs. We, finally, have one last discussion to talk about the significant difference in the results.

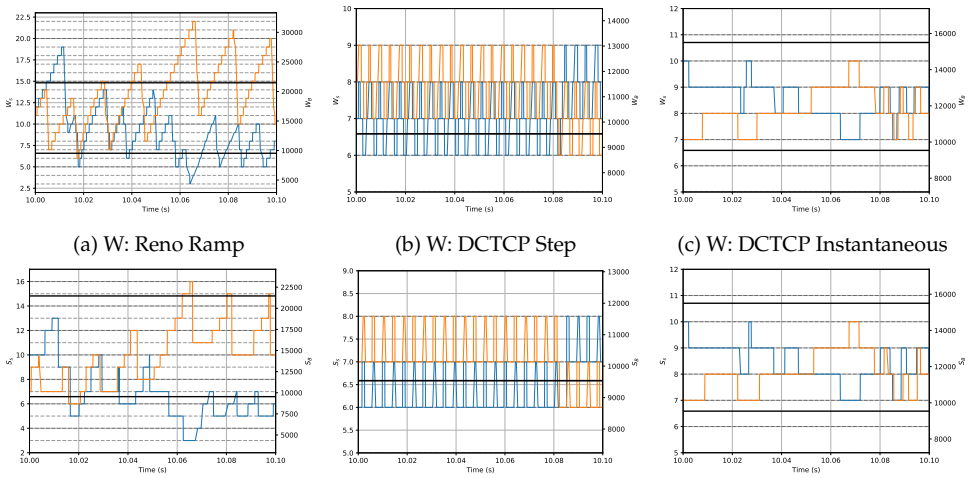
9.1 Convergence Test

We start with a convergence test between two senders in each configuration. Our goal with this test is to verify an excellent initial performance from both Reno and DCTCP.

9.1.1 Additive Increase Multiplicative Decrease (AIMD)

We inspect the congestion window and ssthresh of the two senders. As earlier, we expect both senders to fall within the markers of the experiment parameters.

In Figure 9.1, we notice how well all configuration performs in keeping their congestion window and ssthresh to the appropriate value of the experiment. Additionally, we observe the typical AIMD style congestion



(a) W: Reno Ramp (b) W: DCTCP Step (c) W: DCTCP Instantaneous

(d) S: Reno Ramp (e) S: DCTCP Step (f) S: DCTCP Instantaneous

Figure 9.1: Experiment #2: W & S

control, where one segment gets added to the congestion window per RTT. However, the multiplicative decrease seems to be slow and inefficient. The sender uses an RTT to decrease its congestion window to the new value of ssthresh.

9.1.2 Marking Rate

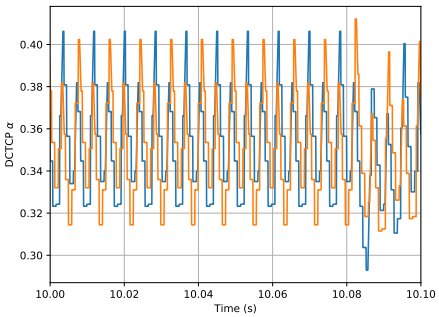
Next, we investigate the marking probability of the individual configuration. We expect a similar trend in AIMD as the one experienced by our LS-AIMD algorithm.

In Figure 9.2, we see that the step configuration still induces a very high marking rate and seems to be some synchronisation ongoing between the two senders. In the marking probability, we can see spikes hitting 50% periodically. On the other hand, the ramp configuration continues yielding the lowest marking rate. The instantaneous configuration does neither change and has, on average, a moderate level of congestion with a small deviation. We are, therefore, satisfied with how similar the marking rate of current Linux TCP compares to our algorithm.

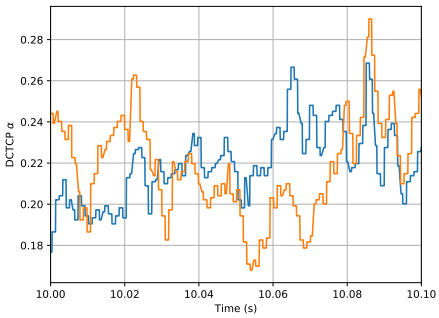
9.1.3 Queueing Delay & Smoothed Round Trip Time (SRTT)

Going over to the delay aspect of the experiment, we are once again interested finding out whatever Reno and DCTCP keep the queue within the parameters of RED. We are also looking into what form of stability each configuration bring vs the same configuration of LS-AIMD.

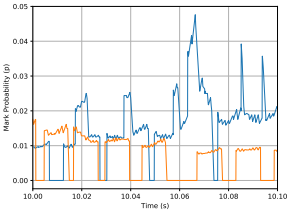
In Figure 9.3, we observe no major contradicting results. Reno and DCTCP appear to be going approximately as fast as recommended by the AQM. Neither algorithm causes any form of added delay due to unresponsiveness. We do, however, see that the ramp configuration yield the worst performance and is unable to keep a smooth queue.



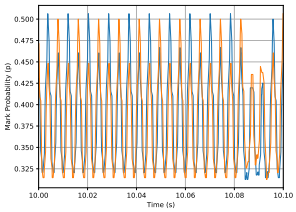
(a) α: DCTCP Step



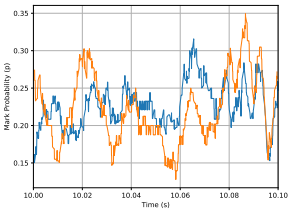
(b) α: DCTCP Instantaneous



(c) p: Reno Ramp

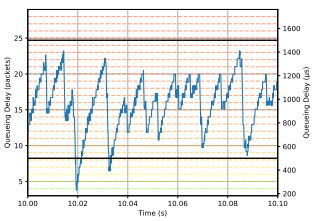


(d) p: DCTCP Step

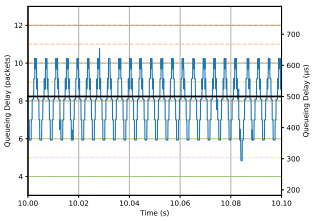


(e) p: DCTCP Instantaneous

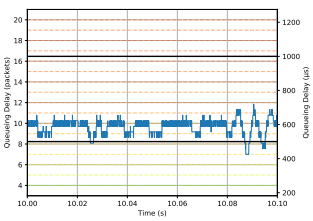
Figure 9.2: Experiment #2: Loss rates



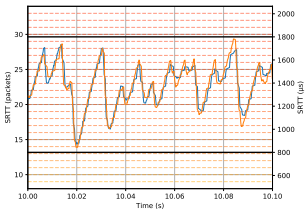
(a) Q: Reno Ramp



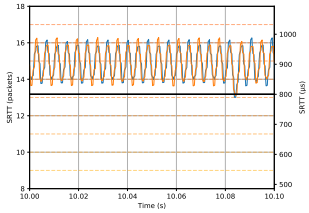
(b) Q: DCTCP Step



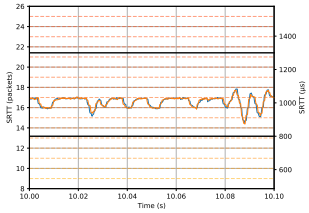
(c) Q: DCTCP Instantaneous



(d) SRTT: Reno Ramp

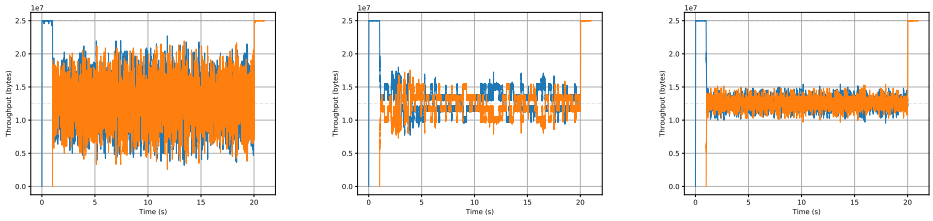


(e) SRTT: DCTCP Step



(f) SRTT: DCTCP Instantaneous

Figure 9.3: Experiment #2: Queueing delay & SRTT



(a) Throughput: Reno Ramp

(b) Throughput: DCTCP step

(c) Throughput: DCTCP Instantaneous

(d) Utilisation: Reno Ramp

(e) Utilisation: DCTCP step

(f) Utilisation: DCTCP Instantaneous

Figure 9.4: Experiment #2: Throughput & link utilisation

9.1.4 Throughput & Link Utilisation

Finally, we retake the throughput and link utilisation part of the evaluation. We dig into the rate at which all senders converge to the fair share of the capacity. Additionally, we measure the utilisation of the link. A small as possible deviation is prefered in both metrics.

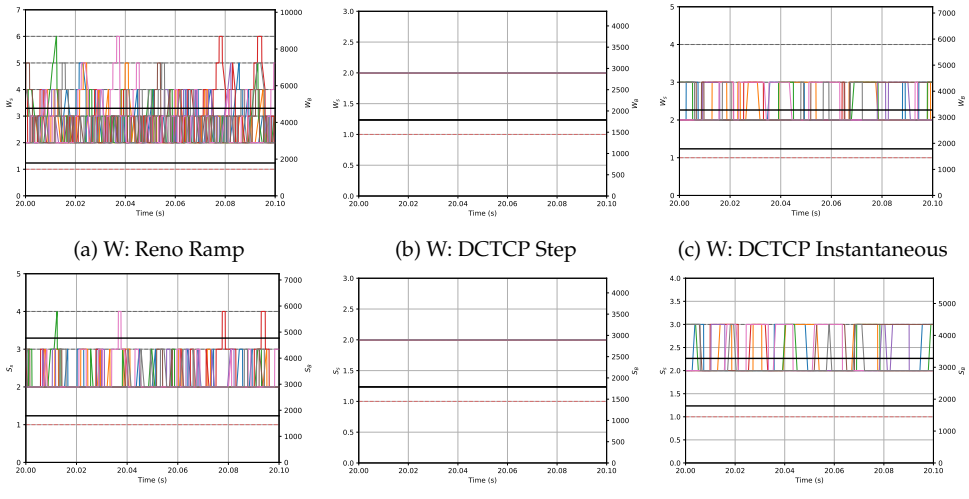
In Figure 9.4, we observe very similar results when comparing the AIMD side by side to LS-AIMD. However, we notice a very slight improvement in the rate at which the two senders converge in the ramp and step configuration in our former evaluation of LS-AIMD. On the other hand, the link utilisation is not a problem for neither of the algorithm.

9.2 The submss Regime

We are now ready to see how well Reno and DCTCP perform in the submss regime. We expect both algorithms to perform slightly worse in all configurations since each sender now has to keep a low congestion window made of just 1.24 segments per RTT. However, we expect the step configuration to have massive trouble as it has no maximum threshold, whereas the other two configurations have room to relax.

9.2.1 Additive Increase Multiplicative Decrease (AIMD)

We are now going to evaluate two major weakness of the AIMD style congestion control. The first problem is that a sender most often increases its congestion window by one segment per RTT, and another problem is that TCP refuses to keep a lower transmission rate than two segments per



(a) W: Reno Ramp (b) W: DCTCP Step (c) W: DCTCP Instantaneous

(d) S: Reno Ramp (e) S: DCTCP Step (f) S: DCTCP Instantaneous

Figure 9.5: Experiment #3A: W & S

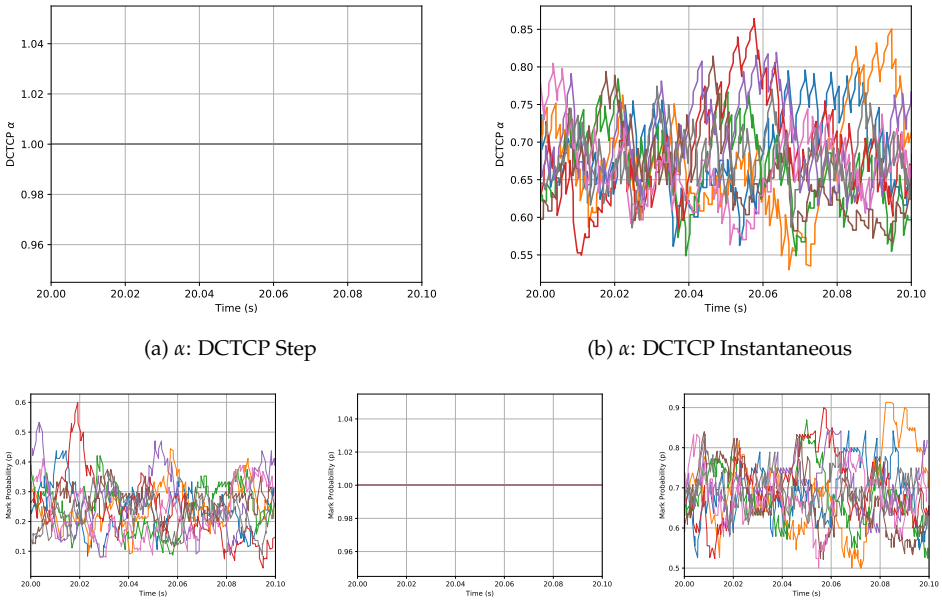
RTT. A shallow base RTT makes use of AIMD inconvenient, so we are not expecting any excellent performance from either Reno or DCTCP.

In Figure 9.5, we see precisely what we predicted, notice how neither of configuration has a congestion window of fewer than two segments. Meaning, the sender has never kept fewer than two segments inflight per RTT in either of the experiment. Notice how the step configuration is stuck at exactly two segments per RTT since the instantaneous marking threshold is lower than two segments. A Linux sender refrains from increasing its congestion window when all RTT ends up with a congestion mark. The instantaneous configuration also suffers as a probe of an additional segment inflight is too severe in the environment of submss regime and ends with a mark at least every second RTT. Finally, the ramp configuration comes out best but has the advantage of a wider gap between the minimum and maximum threshold of RED. However, Reno is nonetheless very aggressive in its attempt at occupying any additional chunk of the capacity.

9.2.2 Marking Rate

We investigate the marking probability of RED to understand how severe the congestion was and what picture each sender had of the bottleneck.

In Figure 9.6, we are not surprised when we see that the step configuration has reached marking saturation. It is the same reason why the sender has no chance at probing for additional capacity as holding two-segment inflight per RTT induce congestion mark every RTT. Notice how the ramp and instantaneous configuration now has slightly higher level congestion than under LS-AIMD.



(a) α: DCTCP Step

(b) α: DCTCP Instantaneous

(c) p: Reno Ramp

(d) p: DCTCP Step

(e) p: DCTCP Instantaneous

Figure 9.6: Experiment #3A: Loss rates

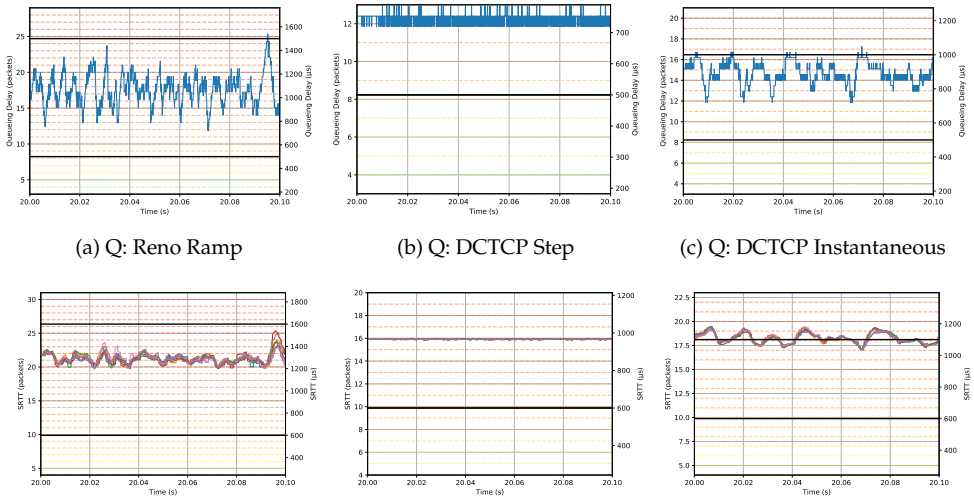
9.2.3 Queueing Delay & Smoothed Round Trip Time (SRTT)

Next, we examine the queueing delay and the SRTT to reveal what kind of penalty going unresponsive for a more extended period has when the AQM is not performing any form of eviction of packets, i.e. pure ECN plus tail-drop.

In Figure 9.7, we notice how the queueing delay of all configuration has gone slightly up, and we see the same trend in the SRTT because a significant part of SRTT is the queueing delay. It is not a surprise that both queueing delay and SRTT is kept more stable now than before as a sender who reaches marking saturation cannot probe for additional capacity, i.e. an unresponsive sender. How much additional queue we get depends on how low the base RTT is and how many senders goes unresponsive. The queueing delay grows proportionally for either case. However, even under marking saturation, we do not see any form of synchronisation in the step configurations queue. Ironically, the problem we are solving is that the sender refuses to lose up its acknowledgement clock and builds a queue, but the fix to the synchronisation problem is also the acknowledgement clock. The transmission of a set of "acknowledge clocked" sender cannot become synchronised over more extended periods as the queue of the bottleneck break up the synchronisation.

9.2.4 Throughput & Link Utilisation

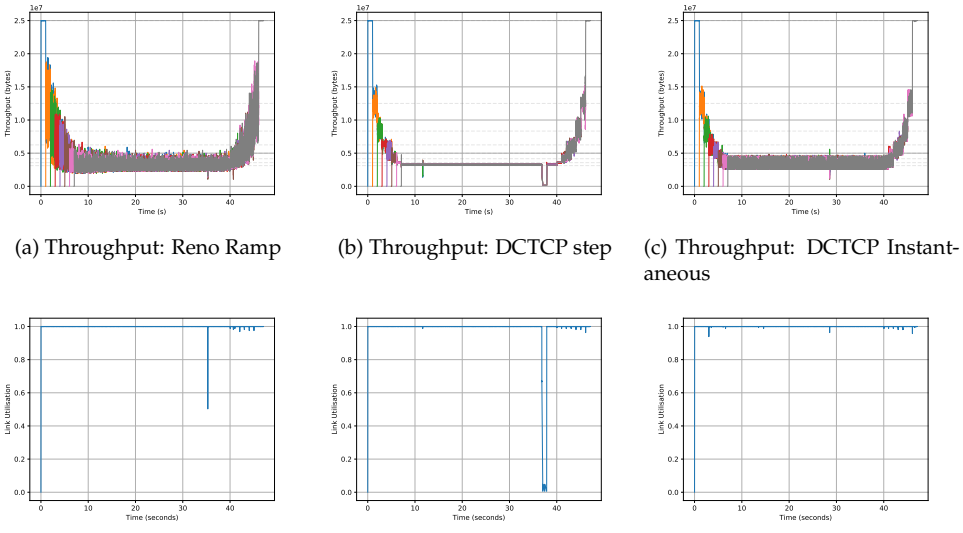
Finally, we go back at evaluating the throughput and link utilisation of the different configurations.



(a) Q: Reno Ramp (b) Q: DCTCP Step (c) Q: DCTCP Instantaneous

(d) SRTT: Reno Ramp (e) SRTT: DCTCP Step (f) SRTT: DCTCP Instantaneous

Figure 9.7: Experiment #3A: Queueing delay & SRTT



(a) Throughput: Reno Ramp (b) Throughput: DCTCP step (c) Throughput: DCTCP Instantaneous

(d) Utilisation: Reno Ramp (e) Utilisation: DCTCP step (f) Utilisation: DCTCP Instantaneous

Figure 9.8: Experiment #3A: Throughput & link utilisation

In Figure 9.8, we notice how well smoothed all configuration has become compared to LS-AIMD. Well, the only benefit of having a set of competitors being unresponsive is that the capacity shareout between them becomes very stable as the sender cannot probe. However, the benefit of a very smooth capacity is not worth trading for a penalty of more queueing delay.

9.3 Exhaustive & Scalability Test

One last thing left to do is to perform an extensive test under different base RTT and MTU sizes, and compare these results against LS-AIMD. An important goal with these experiments is to locate some of the different trends between AIMD and LS-AIMD. Nonetheless, we are looking to achieve the same set goals as we did in the evaluation of LS-AIMD. The question we want an answer to is who scales the most.

9.3.1 Link Utilisation

First, as always, we need to verify the utilisation of the link. We expect AIMD to perform better at keeping the link utilised as there is no risk of running out of a deep queue among unresponsive senders, whereas a shallow queue is extremely hard to maintain over a more extended period and especially when performing paced transmission greater than one RTT.

In Figure 9.9, we see that AIMD goes well in keeping the link utilisation quite nicely over the different experiments. However, we notice how the ramp configuration struggles to maintain the utilisation of a shallow queue over higher base RTTs. We can explain this outcome by remembering that Reno does rate halving, and is neither efficient at recovering loss of a high congestion window. This process is slow, and if enough sender gets marked, not hard with a shallow queue, then Reno is unable to recover fast enough to prevent the queue from drying. Remarkably, we observe that under jumbo frames the link in the most cases kept fully utilised without any form of deviation.

9.3.2 Queueing Delay

We inspect the queueing delay of the AQM to further look into what causes such a proper utilisation of the link for such a long period.

In Figure 9.10, we see a slight increase in the queueing delay for the experiment with MTU of 1500 bytes. However, under jumbo frames, we see remarkably high queueing delays and a steady increasing queue for lower base RTT. A deep queue explains why the link never got a chance to go unutilised.

9.3.3 Smoothed Round Trip Time (SRTT)

Further inspection of the SRTT of all senders should tell us what kind of impact a steady growing queue has on the traversing traffic. This delay

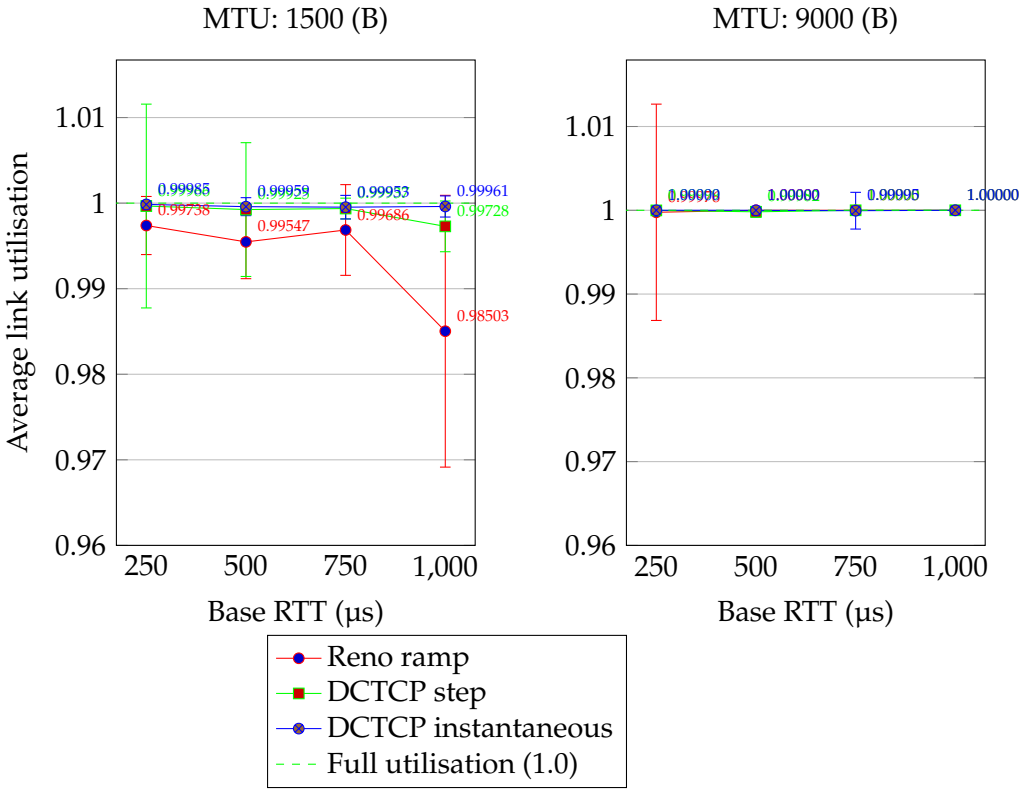


Figure 9.9: Experiment #4: Link utilisation dynamics

is approximately the average response time an application sees, so any growth in the RTT harms all applications over the bottleneck.

In Figure 9.11, we notice how the SRTT is straightening out horizontally instead of just being linear across the diagonal. So any advantage a lower RTT gives slowly decreases when going for a lower RTT since at the same time the queue grows. Remarkably, the jumbo frame experiment turns into sky-high queueing occupancy, and the sender sees a fully horizontal line across all base RTTs. Meaning, no particular benefit is gained through a lower base RTT, and neither has the network, or AQM has any blame in this. The reason why the line at its worst becomes a horizontal line comes from the fact that the BDP proportionally decreases for lower base RTT, and the competitors' contribution of queueing delay keeps on proportionally increasing since they have no mean in backing off. Meaning, a lower base RTT is enough to yield terrible results when TCP starts to malfunction. However, what about other factors such as higher MTU or more competitors? A higher MTU or any more level of congestion places the horizontal line higher up, resulting in a deeper queue.

9.3.4 Marking Rate

We once again have to revisit the marking probability of RED to get a clear picture of why our results indicate proportionally growing queue. We expect the marking rate of RED to be near marking saturation since the

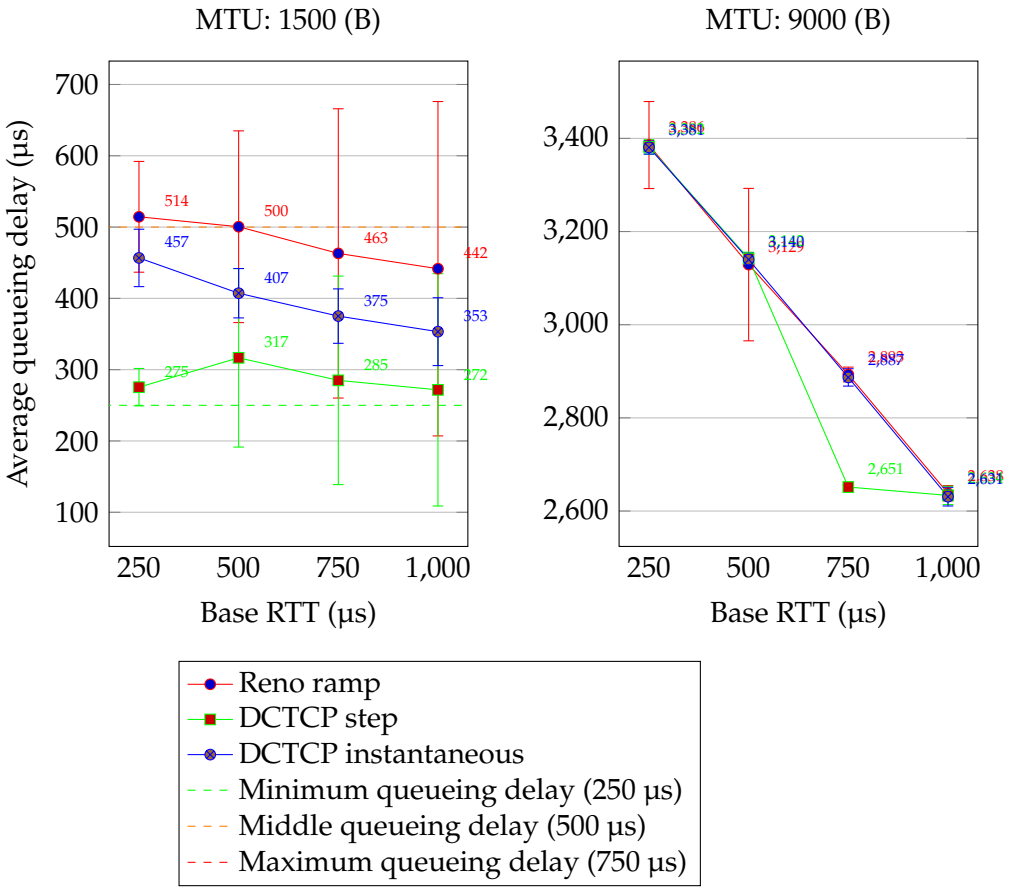


Figure 9.10: Experiment #4: Queue dynamics

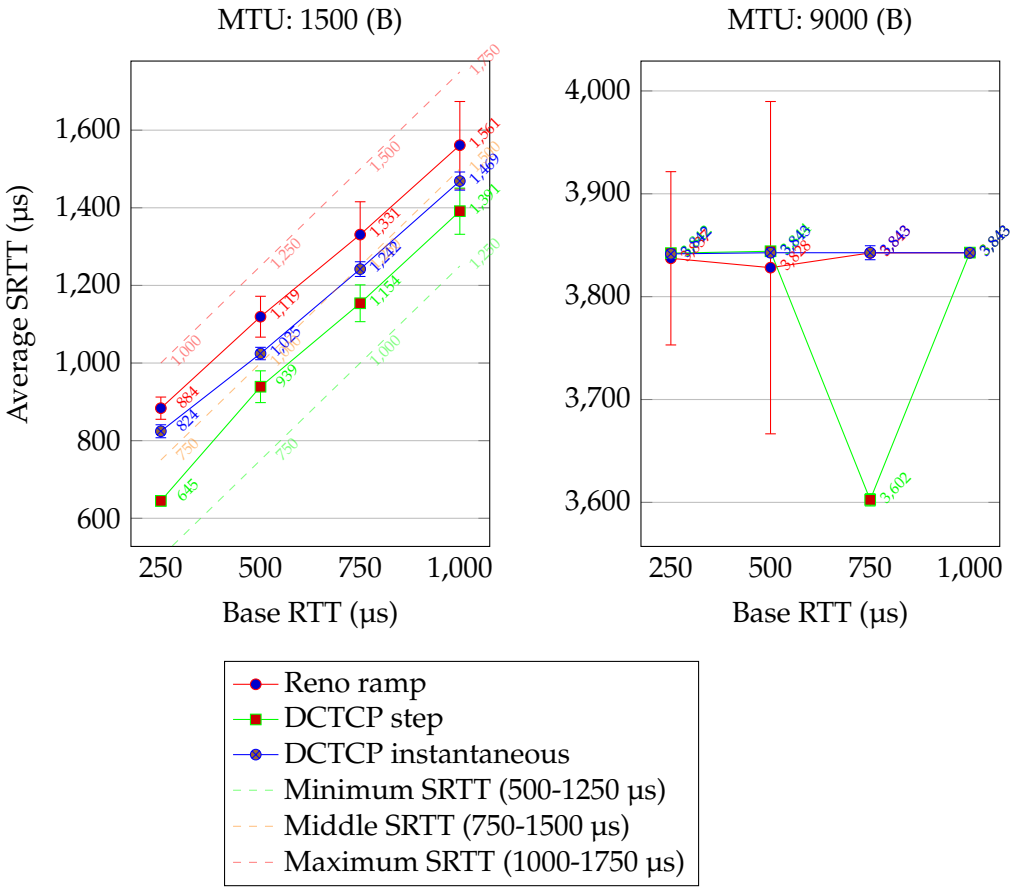


Figure 9.11: Experiment #4: SRTT dynamics

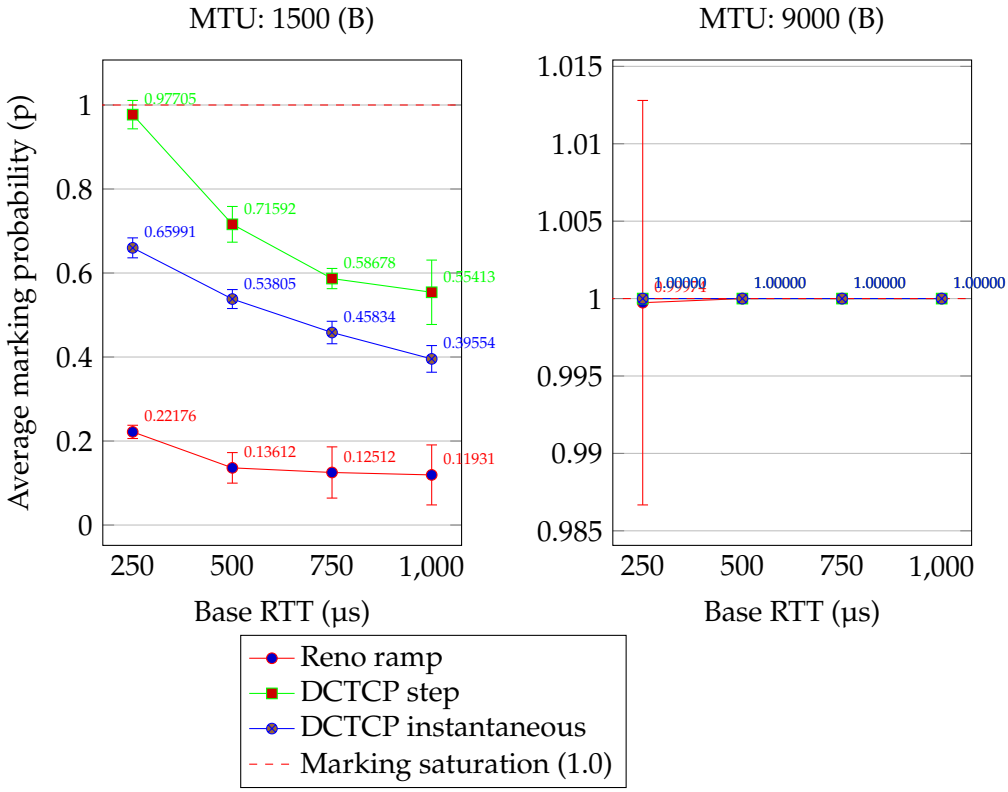


Figure 9.12: Experiment #4: Marking dynamics

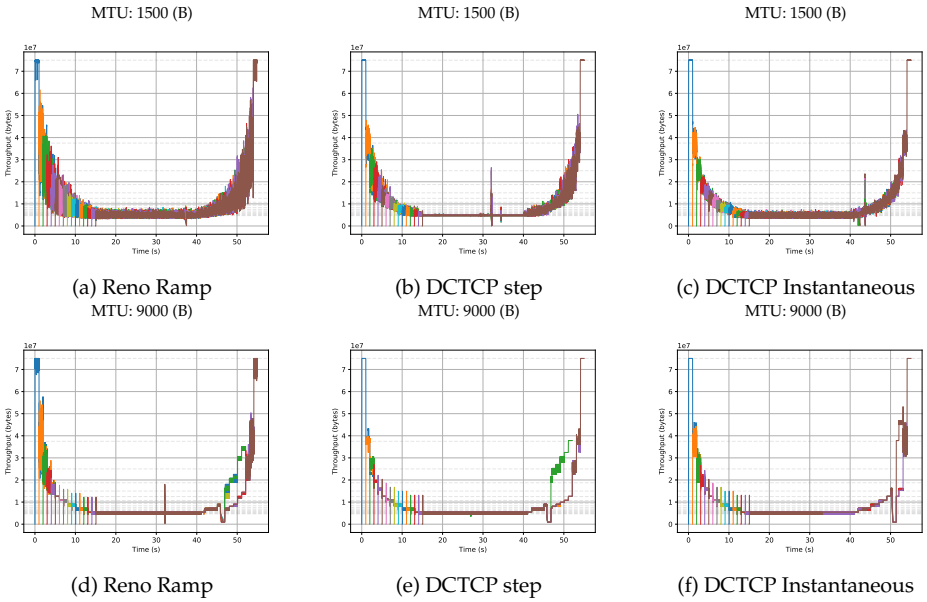
queue is nowhere near the parameters of the experiment.

In Figure 9.12, we see that the average marking rate of RED keeps on steady growing when nearing lower base RTT. The growth is much steeper than what LS-AIMD achieved. We see that the step configuration reaches marking saturation under the lowest base RTT we test. Even the ramp configuration has a way higher probability of being marked. In both experiments, RED show an apparent effort with the intent to reduce the transmission rate the traversing traffic. The experiment with jumbo frames reaches marking saturation across all configurations and all base RTTs. The transmission rate of traversing traffic is nowhere near the parameters of the experiment.

9.3.5 Throughput

As a concluding confirmation of whatever the stability in the capacity continues the same trend as earlier, we reexamine the throughput level of the experiment with the lowest base RTT for both MTU sizes.

In Figure 9.13, we verify that all except two configurations with lower MTU has a long-lasting period with unresponsive senders. Nonetheless, this level of stability in the capacity shareout indicates that the senders over the bottleneck stopped probing for an extended period and were proactively contributing toward a state of marking saturation.



(a) Reno Ramp (b) DCTCP step (c) DCTCP Instantaneous

(d) Reno Ramp (e) DCTCP step (f) DCTCP Instantaneous

Figure 9.13: Experiment #4A & #4E: Throughput

9.4 Queueing Delay Trends

We present a summary of our findings here about the dynamics of queueing delay across all configuration over the conducted experiments. We show a little bit more statistics about our data on what the instantaneous queueing delay was in each experiment over the period where all competitors participated.

We now give out the minimum and maximum value of the instantaneous queue to show some interesting remarks. We show the values of mean and standard deviation from earlier on. We also include 95th and 99th percentile to know what the highest queueing delay observed is in the respective percentages group of observation.

In Tables 9.1 to 9.3, we notice that already at the two first experiments, LS-AIMD has better stability in the results. For the next two following experiments, we notice how the maximum queueing delay sky-rockets for some configurations. We need to analyse more to locate the cause for this. We also notice how LS-AIMD keep lower percentiles than AIMD. Otherwise, the results are relatively similar.

Moving to the fourth experiment with MTU of 1500 bytes, we see that LS-AIMD is generally closer to the middle of the ramp of RED and the percentiles do not indicate any queue build up either. We also notice how the standard deviation slowly decreases when moving to lower base RTT. However, LS-AIMD experience a higher maximum queue occupancy in all these experiments. LS-AIMD with a lower C underperforms as it held an inappropriate low queueing delay and no form of stability. Meaning, a lower queueing delay does not always mean a better configuration; it depends on the parameter of RED and how stable TCP is at cooperating with RED at holding them. RED has more information available, and TCP

	Experiment (#)	1	2	3A	3B	4A	4B	4C	4D	4E	4F	4G	4H
RED Ramp Marking	**Reno**												
	Min (µs)	0	0	720	294	196	0	0	0	0	950	2162	2424
	Max (µs)	1146	1572	6127	1998	917	1277	1474	1081	3768	3178	2916	2686
	Mean (µs)	555	875	1067	1047	514	500	463	441	3385	3129	2893	2638
	Standard deviation (µs)	212	214	145	188	77	134	202	234	93	163	15	16
	95th precentile (µs)	917	1212	1310	1376	655	720	753	786	3407	3145	2916	2654
	99th precentile (µs)	983	1343	1409	1507	688	786	819	851	3407	3178	2916	2654
	LS-AIMD Reno (C: 0.78125%)												
	Min (µs)	-	-	-	-	0	98	0	0	0	0	0	0
	Max (µs)	-	-	-	-	688	1343	491	458	1114	1081	1114	1933
	Mean (µs)	-	-	-	-	258	247	237	225	464	420	402	375
	Standard deviation (µs)	-	-	-	-	41	34	38	44	161	162	161	161
	95th precentile (µs)	-	-	-	-	327	294	294	294	720	688	655	622
	99th precentile (µs)	-	-	-	-	327	327	327	327	819	786	786	753
	LS-AIMD Reno (C: 100%)												
	Min (µs)	0	131	393	393	0	0	0	0	0	0	0	0
	Max (µs)	1081	1572	1540	1605	1572	1769	1867	2818	3571	3178	2916	2686
	Mean (µs)	571	866	1033	1016	495	487	495	469	1763	1567	1452	1289
	Standard deviation (µs)	185	184	103	138	54	67	74	87	1096	987	923	839
	95th precentile (µs)	851	1146	1212	1245	589	589	622	589	3342	3014	2654	2392
	99th precentile (µs)	950	1277	1277	1343	622	622	655	655	3407	3145	2883	2555

Table 9.1: RED ramp marking: queueing delay

	Experiment (#)	1	2	3A	3B	4A	4B	4C	4D	4E	4F	4G	4H
RED Step Marking	**DCTCP**												
	Min (µs)	294	229	0	262	0	0	0	0	3145	2883	2162	2490
	Max (µs)	655	720	1048	1081	1048	4653	655	950	3473	3211	2686	2686
	Mean (µs)	472	480	748	651	275	316	285	271	3381	3142	2651	2633
	Standard deviation (µs)	50	83	14	155	26	125	146	162	15	13	10	19
	95th precentile (µs)	557	622	753	917	294	524	524	524	3407	3145	2654	2654
	99th precentile (µs)	589	622	786	983	327	589	557	557	3407	3178	2654	2654
	LS-AIMD DCTCP (C: 0.78125%)												
	Min (µs)	-	-	-	-	0	0	0	0	0	0	0	0
	Max (µs)	-	-	-	-	360	1703	753	720	917	950	884	884
	Mean (µs)	-	-	-	-	198	200	198	197	160	142	128	119
	Standard deviation (µs)	-	-	-	-	33	30	31	32	135	131	127	124
	95th precentile (µs)	-	-	-	-	262	229	229	229	425	393	393	360
	99th precentile (µs)	-	-	-	-	262	262	262	262	524	524	524	491
	LS-AIMD DCTCP (C: 100%)												
	Min (µs)	294	0	0	196	0	0	0	0	0	0	0	0
	Max (µs)	1605	655	7929	3309	1310	4325	1933	622	3407	3178	2916	2654
	Mean (µs)	482	496	621	569	259	262	266	261	1630	1179	886	708
	Standard deviation (µs)	55	62	173	111	49	63	87	95	1075	565	357	337
	95th precentile (µs)	557	589	753	753	327	360	425	425	3375	2424	1441	1343
	99th precentile (µs)	557	622	786	753	360	425	458	491	3407	3145	2359	1900

Table 9.2: RED step marking: queueing delay

	Experiment (#)	1	2	3A	3B	4A	4B	4C	4D	4E	4F	4G	4H
RED Instantaneous Marking	**DCTCP**												
	Min (µs)	393	393	557	393	229	98	0	98	2785	2490	2392	2490
	Max (µs)	655	786	1179	1310	622	819	819	753	3407	3178	4030	2686
	Mean (µs)	546	584	885	893	456	407	375	353	3380	3139	2886	2630
	Standard deviation (µs)	46	45	59	99	40	34	38	47	14	13	18	20
	95th precentile (µs)	622	655	983	1048	524	458	425	425	3407	3145	2916	2654
	99th precentile (µs)	622	688	1048	1114	557	491	458	491	3407	3145	2916	2654
	LS-AIMD DCTCP (C: 0.78125%)												
	Min (µs)	-	-	-	-	0	0	0	0	0	0	0	0
	Max (µs)	-	-	-	-	491	5275	655	786	950	1114	1081	7012
	Mean (µs)	-	-	-	-	238	236	235	234	159	261	244	230
	Standard deviation (µs)	-	-	-	-	33	35	28	30	135	137	138	149
	95th precentile (µs)	-	-	-	-	294	294	262	262	425	491	491	458
	99th precentile (µs)	-	-	-	-	294	294	294	294	557	589	589	589
	LS-AIMD DCTCP (C: 100%)												
	Min (µs)	229	425	557	327	0	0	0	0	0	0	0	0
	Max (µs)	720	786	1114	2752	3211	1605	2490	1474	5210	3178	2916	2654
	Mean (µs)	556	604	821	812	419	388	367	352	1476	1145	914	811
	Standard deviation (µs)	37	45	58	101	53	49	61	58	713	406	281	299
	95th precentile (µs)	622	688	917	983	491	458	458	458	3211	1802	1343	1310
	99th precentile (µs)	655	720	950	1015	524	491	491	491	3407	2850	1835	1998

Table 9.3: RED instantaneous marking: queueing delay

should, therefore, go as fast as RED permits such that the bottleneck has enough queueing occupancy to maintain link utilisation.

Moving to the last set of experiments, now jumbo frames, we see that both AIMD and LS-AIMD with a high C build a very deep queue. However, notice how LS-AIMD did not become unresponsive but instead is unable at keeping the standard deviation low since too much get added and taken away from the congestion window. On the other hand, LS-AIMD with a low C performs exceptionally well, more than ten times better than the other, by both being responsive and accumulate just enough to keep a low deviation, although C should have been a little bit higher to avoid link underutilisation.

9.5 Final Discussion & Remarks

We have now evaluated the results of LS-AIMD against the Linux TCP AIMD for both Reno and DCTCP. We have additionally tried out two distinguishable setups of RED for DCTCP in an attempt at locating problems with the AQM. We have been unsuccessful in finding any problems with RED or the network. From our understanding, the problem lies in TCP and nothing else. We can always continue to ignore this problem and hope a higher quantity of bitrate would save us, but we have demonstrated that by making small changes to TCP, we can perform way better than we currently do. We have briefly explained how hard it is to maintain a shallow queue of two packets, but it is possible as our results of LS-AIMD show. The removal of a deep queue through the use of AQMs makes the transmission rate of a sender more depended on what base RTT the network delivers, the size of the packets and the number of competitors at the bottleneck. The precedence of the problem will, therefore, always be there when base RTT shrinks and a higher bitrate throughout the network does not help as the AQM can then keep a proportionally lower queue occupancy. E.g., a queue of two packets on a one Gbps link with MTU of 1500 bytes is roughly 24 microseconds of queueing delay, while the same setup with a 10 Gbps link allows approximately 2.4 microseconds of queueing delay.

We have also seen a trend where as soon as the traversing traffic stop following the recommendation of the AQM a lower base RTT result in a higher queue occupancy, and this added queueing delay diminishes any gain in the now lower base RTT.

However, LS-AIMD is not entirely perfect yet as it should adapt to the appropriate C by itself.

9.6 Summary

In this chapter, we have rerun the very same set of experiments as in our evaluation chapter. We have seen why clamping the congestion window is a problem that prevents the standard TCP from yielding any excellent performance in an environment mostly dominated by low base RTT.

We intend to conclude our work in the next chapter and give a summary of our main contributions. We also plan to give ideas on how our work can be continued to improve the results and perform tests under more realistic environments.

Part IV

Conclusion

Chapter 10

Conclusion

We are now ready to summarise up our work on the problem we initially thought was easy to solve. We then continue with our main contributions and finally end with a section on what we could have done differently and also look at what else we could have done to improve the results further. We also talk about experiments which could be worth testing against but were simply out of reach for our work because a tremendous work went into just scaling TCP for the submss regime. Experimentation was, therefore, a low priority, but still enough to give away the most vital properties of LS-AIMD. As far as our results predicate a further improvement of LS-AIMD should be worth testing out. We also believe that other researchers should criticise every aspect of our work and take nothing for granted. We could have overseen some more straightforward solutions because we had to add the complexity of C since we were at the beginning not aware that our problem is very similar to the problem of TCP were RTT bias cause one flow to accelerate faster than the other. We did, however, remember that flows in the submss regime should add more to the congestion window per ACK and expected this to work out just fine.

10.1 Summary

We have in this thesis worked against a problem of TCP, which prevents low queueing delays over already low base RTT. The problem boils down to a fundamental problem in TCP, which prevents a lower transmission rate than two segments per RTT. We, first, quantified the prevalence of the problem for broadband, data centre and IPC environments. We then revisited earlier work to form a better understanding of the problem and try to find any particular already existing solutions. This problem, at the beginning of our adventure, had no clear visibility in the previous work and had only gained the interest of a few researchers. It was first in the later stage of the project that we realised some of the work done a decade ago had some similarity with our work. However, it was more about reducing the loss level of TCP under sparse link rates. We also later found some researching teams working on DCTCP, but they suggested drastically modifications which would require a significant change in how

TCP operates and was therefore deemed not to scale over the Internet.

We then went on testing out TCP to see the problem in action and thus formed a set of experiments which we wanted to improve the results on. This brought us to a long design period where we had to solve two severe problems of TCP. First, we had to solve the problem of AIMD not scaling for either a very low or high congestion window. Then, we had to solve another problem which required us to implement a packet conservation clock which would allow us to sender fewer than one segment per RTT, or two when working with delayed acknowledgements.

We then dealt with the complexity of Linux and managed to implement a very simple implementation out of the design. This work was of course not without much hassle since we had to update our version of the kernel midway through the project and then we were surprised by a regression bug in the kernel. Much time went out at first understanding there was a bug there and then locating the cause of the bug. We implemented our work as mainly as a new congestion control module inside the Linux kernel, but some additional changes were needed to support the new functionality of our module.

We further continued our work by testing the implementation extensively to find any scalability issues; these three phases were repeated until we were happy with our results. By three phases, we mean design, implementation and testing period. We were stuck for a while in a period where we found scalability issues one after another. These three phases helped us in improving the design, implementation and also how we tested TCP. We also learned to be more transparent with how we conducted tests of TCP such that it would allow easily replicable tests for our self and others.

Finally, we compared our results against AIMD to reveal how better LS-AIMD performed under both Reno and DCTCP. In total, we tested three RED configurations to see if we could improve the results also from the AQM.

10.2 Main Contributions

- **Can we extend TCP for shallow base RTT?** Yes, we have modified TCP such that the sender adds a local transmission delay to keep a lower transmission rate. The protocol is now capable of scaling in environments which mandates a shallow transmission rate. We describe a solution where the transmission rate has no lower bound unless the implementer sets an artificial limitation.

- **Can we preserve stretch acknowledgements for low congestion window values?**

  Yes, we have designed the solution around this requirement. The solution itself does not depend on when the receiver decides to send an acknowledgement. The sender transmits packets at the minimum of the stretch acknowledgement factor but adds delays between its

packets. The sender later subtracts the added delay. The solution is meant to scale well with significant stretch acknowledgement factors.

- **Can we preserve a congestion window made out of segments in a network with a low transmission rate per RTT?**

  Yes, we propose the use of fractional congestion window. The congestion window extends to have an additional congestion window which allows finer transmission adjustments.

- **Are we able to create a minimal solution which is easy to implement?**

  Yes, the solution requires minimal modification to the sending host and is, therefore, out of the box partially deployable over the Internet. We not only made the solution simple but additionally made ease for other congestion control algorithms to modify the rate of the sender using a one confined congestion window.

10.3 Future Work

We now describe what most likely would be our next steps. Even though, much work has gone into making LS-AIMD do we think more time should be used to do proper experimentation.

More experiments should be conducted under different topologies. For instance, we should test LS-AIMD in a data centre and IPC like environment to reveal any surprises we may have overseen. We also need to perform some realistic tests over, for instance, a path on the Internet and measure whatever we cause starvation or not.

We need to perform experiments where the base RTT varies between the competitors. Although, we did test LS-AIMD in the sub-packet regime are we not confident whatever it would also scale among competitors with different base RTT.

We should also prioritise to benchmark LS-AIMD under a mix of delay critical and long-lasting flows. Another metric we should look closer at is the completion time of flows under a low base RTT environment. We expect LS-AIMD to perform better in either case since an LS-AIMD sender has no mean to build a queue over the maximum threshold of the AQM.

We should also benchmark TCP from the application layer to verify our results. We need to evaluate the goodput and response time between two applications in a typical server-client like setup. Realistic scenario

To further improve the scalability of LS-AIMD, we need to form a way to approximate the correct C such that LS-AIMD always performs well. However, since a sender may need to use a high C even when probing inside the submss regime, do we not see any possible ways to make them converge to a certain C without performing another level of AIMD. But even then, what should be the appropriate increase in C over which time scale? Since the RTT may not be the same.

We have benchmarked against Linux TCP, but it would also be useful to dig into how well TCP perform when correctly following RFC3168. It would also be worth testing against other proposals from earlier research.

We should have conducted experiments to measure the load our code cause to the system and compare against both Reno and DCTCP. It is good that we were able to achieve excellent performance under low base RTT but to scale for even lower base RTT we need to take the processing cost of LS-AIMD more seriously. For shallow base RTT, we may end up running in a network where the cost of LS-AIMD may be higher than the base RTT itself. Additionally, the computational footprint of TCP should also not be as severe that we steal the majority of CPU performing non-critical tasks. The CPU utilisation is critical to keep under low level per TCP connection to allow a higher level of multiplexing. For example, a server serving many clients should not use more CPU than needed for TCP to allow more incoming connections.

We are, of course, expecting our code to utilise more CPU as we have extended the scalability of the protocol, but it would be interesting to find the exact impact of LS-AIMD contra AIMD. We already know that the computation of the add function may have a severe impact even when performing the calculation at most once per RTT. However, the add function, at the moment, gives an accurate answer down to a given precision, we could have approximated the answer and even used lower precision. So, we see ways to bring the utilisation of the CPU within reasonable limits, but not easy to know without any concrete proof.

Bibliography

[1] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta and Murari Sridharan. 'Data center tcp (dctcp)'. In: *ACM SIGCOMM computer communication review* 41.4 (2011), pp. 63–74.

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

[3] M. Allman, S. Floyd and C. Partridge. *Increasing TCP's Initial Window*. RFC 3390. http://www.rfc-editor.org/rfc/rfc3390.txt. RFC Editor, Oct. 2002. URL: http://www.rfc-editor.org/rfc/rfc3390.txt.

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

[5] Mark Allman and Aaron Falk. 'On the effective evaluation of TCP'. In: *ACM SIGCOMM Computer Communication Review* 29.5 (1999), pp. 59–70.

[6] F. Baker and G. Fairhurst. *IETF Recommendations Regarding Active Queue Management*. BCP 197. RFC Editor, July 2015.

[7] S. Bensley, D. Thaler, P. Balasubramanian, L. Eggert and G. Judd. *Data Center TCP (DCTCP): TCP Congestion Control for Data Centers*. RFC 8257. RFC Editor, Oct. 2017.

[8] Theophilus Benson, Aditya Akella and David A. Maltz. 'Network Traffic Characteristics of Data Centers in the Wild'. In: *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*. IMC '10. Melbourne, Australia: ACM, 2010, pp. 267–280. ISBN: 978-1-4503-0483-2. DOI: 10.1145/1879141.1879175. URL: http://doi.acm.org/10.1145/1879141.1879175.

[9] Theophilus Benson, Ashok Anand, Aditya Akella and Ming Zhang. 'Understanding Data Center Traffic Characteristics'. In: *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*. WREN '09. Barcelona, Spain: ACM, 2009, pp. 65–72. ISBN: 978-1-60558-443-0. DOI: 10.1145/1592681.1592692. URL: http://doi.acm.org/10.1145/1592681.1592692.

[10] D. Black. *Relaxing Restrictions on Explicit Congestion Notification (ECN) Experimentation*. RFC 8311. RFC Editor, Jan. 2018.

[11] D. Borman, B. Braden, V. Jacobson and R. Scheffenegger. *TCP Extensions for High Performance*. RFC 7323. http://www.rfc-editor.org/rfc/rfc7323.txt. RFC Editor, Sept. 2014. URL: http://www.rfc-editor.org/rfc/rfc7323.txt.

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

[13] Bob Briscoe. 'Flow rate fairness: Dismantling a religion'. In: *ACM SIGCOMM Computer Communication Review* 37.2 (2007), pp. 63–74.

[14] Bob Briscoe and Koen De Schepper. 'Scaling tcp's congestion window for small round trip times'. In: *Tech. rep., Technical report TR-TUB8-2015-002, BT* (2015).

[15] Bob Briscoe, Koen De Schepper, Olga Albisser, Joakim Misund, Olivier Tilmans, Mirja Kühlewind and Asad Sajjad Ahmed. 'Implementing the 'TCP Prague' Requirements for L4S'. In: *Proc. Netdev 0x13*. Mar. 2019. URL: https://www.files.netdevconf.org/f/4d6939d5f1fb404fafd1/?dl=1.

[16] Bob Briscoe, Koen Schepper and Marcelo Bagnulo. *Low Latency, Low Loss, Scalable Throughput (L4S) Internet Service: Architecture*. Internet-Draft draft-ietf-tsvwg-l4s-arch-01. http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-l4s-arch-01.txt. IETF Secretariat, Oct. 2017. URL: http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-l4s-arch-01.txt.

[17] Jay Chen, Janardhan Iyengar, Lakshminarayanan Subramanian and Bryan Ford. 'TCP Behavior in Sub Packet Regimes'. In: *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '11. San Jose, California, USA: ACM, 2011, pp. 157–158. ISBN: 978-1-4503-0814-4. DOI: 10.1145/1993744.1993804. URL: http://doi.acm.org/10.1145/1993744.1993804.

[18] Jay Chen, Lakshmi Subramanian, Janardhan Iyengar and Bryan Ford. 'TAQ: Enhancing Fairness and Performance Predictability in Small Packet Regimes'. In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. Amsterdam, The Netherlands: ACM, 2014, 7:1–7:14. ISBN: 978-1-4503-2704-6. DOI: 10.1145/2592798.2592819. URL: http://doi.acm.org/10.1145/2592798.2592819.

[19] Inho Cho, Keon Jang and Dongsu Han. 'Credit-Scheduled Delay-Bounded Congestion Control for Datacenters'. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: ACM, 2017, pp. 239–252. ISBN: 978-1-4503-4653-5. DOI: 10.1145/3098822.3098840. URL: http://doi.acm.org/10.1145/3098822.3098840.

[20] J. Chu, N. Dukkipati, Y. Cheng and M. Mathis. *Increasing TCP's Initial Window*. RFC 6928. http://www.rfc-editor.org/rfc/rfc6928.txt. RFC Editor, Apr. 2013. URL: http://www.rfc-editor.org/rfc/rfc6928.txt.

[21] P. J. Denning, D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner and P. R. Young. 'Computing as a discipline'. In: *Computer* 22.2 (Feb. 1989), pp. 63–70. ISSN: 0018-9162. DOI: 10.1109/2.19833.

[22] G. Fairhurst, A. Sathiaseelan and R. Secchi. *Updating TCP to Support Rate-Limited Traffic*. RFC 7661. RFC Editor, Oct. 2015.

[23] W. -. Feng, D. D. Kandlur, D. Saha and K. G. Shin. 'A self-configuring RED gateway'. In: *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320)*. Vol. 3. Apr. 1999, 1320–1328 vol.3. DOI: 10.1109/INFCOM.1999.752150.

[24] Wuchang Feng, Dilip Kandlur, Debanjan Saha and Kang Shin. 'Techniques for eliminating packet loss in congested TCP/IP networks'. In: *U. Michigan CSE-TR-349-97* (1997).

[25] S. Floyd. *HighSpeed TCP for Large Congestion Windows*. RFC 3649. http://www.rfc-editor.org/rfc/rfc3649.txt. RFC Editor, Dec. 2003. URL: http://www.rfc-editor.org/rfc/rfc3649.txt.

[26] S. Floyd. *Metrics for the Evaluation of Congestion Control Mechanisms*. RFC 5166. http://www.rfc-editor.org/rfc/rfc5166.txt. RFC Editor, Mar. 2008. URL: http://www.rfc-editor.org/rfc/rfc5166.txt.

[27] S. Floyd, M. Handley, J. Padhye and J. Widmer. *TCP Friendly Rate Control (TFRC): Protocol Specification*. RFC 5348. http://www.rfc-editor.org/rfc/rfc5348.txt. RFC Editor, Sept. 2008. URL: http://www.rfc-editor.org/rfc/rfc5348.txt.

[28] S. Floyd, J. Mahdavi, M. Mathis and M. Podolsky. *An Extension to the Selective Acknowledgement (SACK) Option for TCP*. RFC 2883. RFC Editor, July 2000.

[29] Sally Floyd. 'RED: Discussions of Setting Parameters'. In: (1st Nov. 1997). URL: http://www.icir.org/floyd/REDparameters.txt (visited on 11/05/2019).

[30] Sally Floyd and Kevin Fall. 'ECN Implementations in the NS Simulator'. In: (1998).

[31] Sally Floyd, Ramakrishna Gummadi, Scott Shenker et al. *Adaptive RED: An algorithm for increasing the robustness of RED's active queue management*. 2001.

[32] Sally Floyd, Mark Handley, Jitendra Padhye and Jörg Widmer. 'Equation-based congestion control for unicast applications'. In: *ACM SIGCOMM Computer Communication Review* 30.4 (2000), pp. 43–56.

[33] Sally Floyd and Van Jacobson. 'Random early detection gateways for congestion avoidance'. In: *IEEE/ACM Transactions on networking* 4 (1993), pp. 397–413.

[34] T. Henderson, S. Floyd, A. Gurtov and Y. Nishida. *The NewReno Modification to TCP's Fast Recovery Algorithm*. RFC 6582. http://www.rfc-editor.org/rfc/rfc6582.txt. RFC Editor, Apr. 2012. URL: http://www.rfc-editor.org/rfc/rfc6582.txt.

[35] Pai-Hsiang Hsiao, H. T. Kung and Koan-Sin Tan. 'Video over TCP with Receiver-based Delay Control'. In: *Proceedings of the 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video*. NOSSDAV '01. Port Jefferson, New York, USA: ACM, 2001, pp. 199–208. ISBN: 1-58113-370-7. DOI: 10.1145/378344.378372. URL: http://doi.acm.org/10.1145/378344.378372.

[36] Pai-Hsiang Hsiao, HT Kung and Koan-Sin Tan. 'Streaming video over TCP with receiver-based delay control'. In: *IEICE transactions on communications* 86.2 (2003), pp. 572–584.

[37] J. Huang, Y. Huang, J. Wang and T. He. 'Adjusting Packet Size to Mitigate TCP Incast in Data Center Networks with COTS Switches'. In: *IEEE Transactions on Cloud Computing* (2018), pp. 1–1. ISSN: 2168-7161. DOI: 10.1109/TCC.2018.2810870.

[38] J. Huang, Y. Huang, J. Wang and T. He. 'Packet Slicing for Highly Concurrent TCPs in Data Center Networks with COTS Switches'. In: *2015 IEEE 23rd International Conference on Network Protocols (ICNP)*. Nov. 2015, pp. 22–31. DOI: 10.1109/ICNP.2015.39.

[39] V. Jacobson. 'Congestion Avoidance and Control'. In: *Symposium Proceedings on Communications Architectures and Protocols*. SIGCOMM '88. Stanford, California, USA: ACM, 1988, pp. 314–329. ISBN: 0-89791-279-9. DOI: 10.1145/52324.52356. URL: http://doi.acm.org/10.1145/52324.52356.

[40] V. Jacobson and R. Braden. *TCP extensions for long-delay paths*. RFC 1072. Oct. 1988. DOI: 10.17487/RFC1072. URL: https://rfc-editor.org/rfc/rfc1072.txt.

[41] Filip Johansson and Christoffer Lindström. *Inter-Process Communication in a Virtualized Environment*. 2018.

[42] Dzmitry Kliazovich, Fabrizio Granelli and Daniele Miorandi. 'Logarithmic window increase for TCP Westwood+ for improvement in high speed, long distance networks'. In: *Computer Networks* 52.12 (2008), pp. 2395–2410.

[43] I. Komnios, A. Sathiaseelan and J. Crowcroft. 'LEDBAT performance in sub-packet regimes'. In: *2014 11th Annual Conference on Wireless On-demand Network Systems and Services (WONS)*. Apr. 2014, pp. 154–161. DOI: 10.1109/WONS.2014.6814738.

[44] H. T. Kung, Koan-Sin Tan and Pai-Hsiang Hsiao. 'TCP with sender-based delay control'. In: *Proceedings ISCC 2002 Seventh International Symposium on Computers and Communications*. July 2002, pp. 283–290. DOI: 10.1109/ISCC.2002.1021691.

[45] Yee-Ting Li, Douglas Leith and Robert N Shorten. 'Experimental evaluation of TCP protocols for high-speed networks'. In: *IEEE/ACM Transactions on Networking (ToN)* 15.5 (2007), pp. 1109–1122.

[46] M. Mathis, J. Mahdavi, S. Floyd and A. Romanow. *TCP Selective Acknowledgment Options*. RFC 2018. RFC Editor, Oct. 1996.

[47] M. Miao, P. Cheng, F. Ren and R. Shu. 'Slowing Little Quickens More: Improving DCTCP for Massive Concurrent Flows'. In: *2015 44th International Conference on Parallel Processing*. Sept. 2015, pp. 689–698. DOI: 10.1109/ICPP.2015.78.

[48] Greg Minshall. *A suggested modification to Nagle's algorithm*. Tech. rep. Internet-Draft draft-minshall-nagle-01, Internet Engineering Task Force, 1999. URL: https://tools.ietf.org/html/draft-minshall-nagle-01.

[49] R. Morris. 'TCP behavior with many flows'. In: *Proceedings 1997 International Conference on Network Protocols*. Oct. 1997, pp. 205–211. DOI: 10.1109/ICNP.1997.643715.

[50] J. Nagle. *Congestion Control in IP/TCP Internetworks*. RFC 896. RFC Editor, Jan. 1984. URL: http://www.rfc-editor.org/rfc/rfc896.txt.

[51] Cisco Visual Networking. 'Cisco global cloud index: Forecast and methodology, 2016–2021'. In: *White paper. Cisco Public, San Jose* (2016).

[52] Pai-Hsiang Hsiao, H. T. Kung and Koan-Sin Tan. 'Active delay control for TCP'. In: *GLOBECOM'01. IEEE Global Telecommunications Conference (Cat. No.01CH37270)*. Vol. 3. Nov. 2001, 1626–1631 vol.3. DOI: 10.1109/GLOCOM.2001.965855.

[53] V. Paxson, M. Allman, J. Chu and M. Sargent. *Computing TCP's Retransmission Timer*. RFC 6298. http://www.rfc-editor.org/rfc/rfc6298.txt. RFC Editor, June 2011. URL: http://www.rfc-editor.org/rfc/rfc6298.txt.

[54] V. Paxson, M. Allman, S. Dawson, W. Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke and B. Volz. *Known TCP Implementation Problems*. RFC 2525. RFC Editor, Mar. 1999.

[55] J. Postel and J. Reynolds. *Telnet Protocol Specification*. STD 8. http://www.rfc-editor.org/rfc/rfc854.txt. RFC Editor, May 1983. URL: http://www.rfc-editor.org/rfc/rfc854.txt.

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

[57] Pawan Prakash, Myungjin Lee, Y Charlie Hu, Ramana Rao Kompella et al. 'Jumbo frames or not: That is the question!' In: (2013).

[58] Lili Qiu, Yin Zhang and Srinivasan Keshav. 'Understanding the Performance of Many TCP Flows'. In: *Comput. Netw.* 37.3-4 (Nov. 2001), pp. 277–306. ISSN: 1389-1286. DOI: 10.1016/S1389-1286(01)00203-1. URL: http://dx.doi.org/10.1016/S1389-1286(01)00203-1.

[59] K. Ramakrishnan and S. Floyd. *A Proposal to add Explicit Congestion Notification (ECN) to IP*. RFC 2481. RFC Editor, Jan. 1999.

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

[61] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert and R. Scheffenegger. *CUBIC for Fast Long-Distance Networks*. RFC 8312. RFC Editor, Feb. 2018.

[62] KW Ross and JF Kurose. 'Delay and Loss in Packet-Switched Networks'. In: (7th June 2000). URL: https://www.net.t-labs.tu-berlin.de/teaching/computer_networking/01.06.htm (visited on 08/07/2019).

[63] Pasi Sarolahti and Alexey Kuznetsov. 'Congestion Control in Linux TCP.' In: *USENIX Annual Technical Conference, FREENIX Track*. 2002, pp. 49–62.

[64] S. Shalunov, G. Hazel, J. Iyengar and M. Kuehlewind. *Low Extra Delay Background Transport (LEDBAT)*. RFC 6817. http://www.rfc-editor.org/rfc/rfc6817.txt. RFC Editor, Dec. 2012. URL: http://www.rfc-editor.org/rfc/rfc6817.txt.

[65] N. Spring, D. Wetherall and D. Ely. *Robust Explicit Congestion Notification (ECN) Signaling with Nonces*. RFC 3540. RFC Editor, June 2003.

[66] J. Touch, A. Mankin and R. Bonica. *The TCP Authentication Option*. RFC 5925. RFC Editor, June 2010.

[67] Clay S Turner. 'A fast binary logarithm algorithm [DSP tips & tricks]'. In: *IEEE Signal Processing Magazine* 27.5 (2010), pp. 124–140.

[68] Arun Venkataramani, Ravi Kokku and Mike Dahlin. 'TCP Nice: A Mechanism for Background Transfers'. In: *SIGOPS Oper. Syst. Rev.* 36.SI (Dec. 2002), pp. 329–343. ISSN: 0163-5980. DOI: 10.1145/844128.844159. URL: http://doi.acm.org/10.1145/844128.844159.

[69] Michael Welzl and David Ros. *A Survey of Lower-than-Best-Effort Transport Protocols*. RFC 6297. June 2011. DOI: 10.17487/RFC6297. URL: https://rfc-editor.org/rfc/rfc6297.txt.

Appendices

Appendix A

TCP Background

A.1 Transmission Control Protocol (TCP)

Transmission Control Protocol (TCP)[12, 56] is a standardised transport protocol by the Internet Engineering Task Force (IETF). TCP is widely used over the Internet Protocol (IP) and has evolved through updates given as Request For Comments (RFC). A TCP implementation uses a variable header for communication with another TCP endpoint, and the application data is appended on top of this header (see Figure A.1). TCP in contrast to the alternative User Datagram Protocol (UDP) also have a source, destination port and Cyclic Redundancy Check (CRC). The source and destination port allow process-to-process communication, and CRC is used to detect transmission errors. In TCP the use of CRC is mandatory for both the sender and the receiver, while in UDP it is optional. UDP is a straightforward protocol, and if applications need some mechanisms found in TCP, it must be either be implemented by the application or use other protocols that extend UDP.

Network elements over the Internet try their best to deliver the packets (end-to-end principle), but there is no guarantee for either the delivery, ordering or free-for-error transmission. TCP recovers from this by implementing retransmission of lost segments, reordering of incoming segments and use of Cyclic Redundancy Check (CRC). Meaning that TCP provides reliable and inorder communication between two nodes in the network, the application only sees an in-order stream of bytes.

Packets loss usually occur over the Internet due to queue constraint at the network element in the path between the sender and receiver. The packet could also get corrupted during transmission between two links in the network. Although, corrupted packets are more likely to happen in wireless communication. TCP, therefore, uses acknowledgement number to receipt newly received data. TCP sets the ACK flag in the TCP header to acknowledge data. The flag is used to indicate that the acknowledgement number in the TCP header is the next expected bytes in the stream. The acknowledgement let the sender knows that the receiver has successfully received bytes sent before the acknowledgement number.

Segments may come in the wrong order over the Internet. A segment

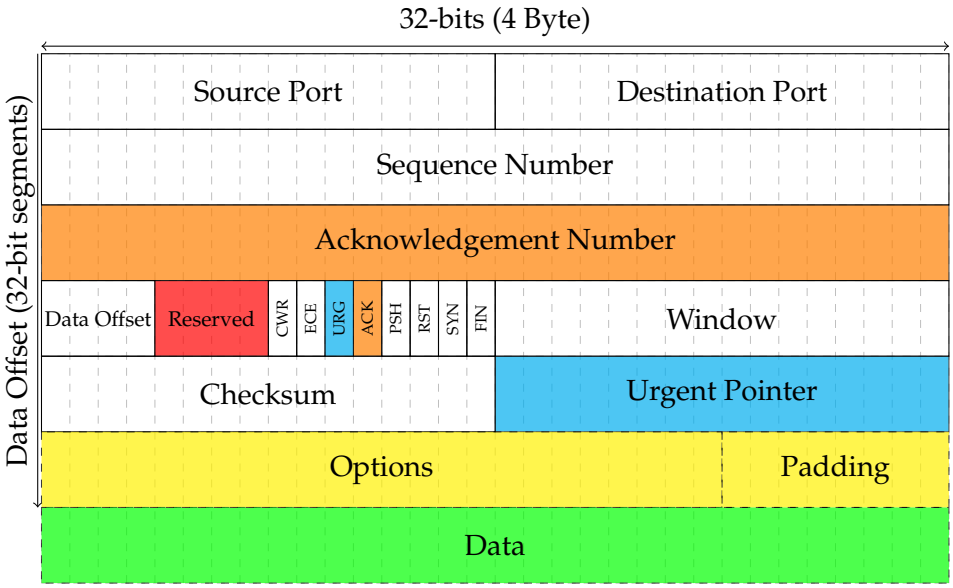


Figure A.1: TCP Header

loss could have occurred, and the next segment may arrive first, making a hole in the data. A segment received in the wrong order cannot be delivered to the application yet since the objective of TCP is to deliver the data in the same order as sent by the sender. Therefore, TCP use sequence numbers in the header to provide data in the right order to the receiving application. Additionally, receiving TCP use the sequence number for the selection of the acknowledgement number.

Maximum Segment Size (MSS) can be altered by both hosts when initiating the connection to override the default value; the default value is 536 bytes. The MSS value tells the sending host how large segments the receiving host is willing to accept. This limitation does only limit the amount of octets/bytes sent after the variable TCP headers (past any TCP option used). However, the sender may not always be able to send a segment as large as MSS bytes due to other factors such as MTU, path MTU discovery or flow window. Therefore, the sender may send segments of less than MSS bytes, known as Sender Maximum Segment Size (SMSS). Conflicts occur as both hosts may have different requirements for what fulfils a full segment. The receiver uses the MSS value for transmission of stretch acknowledgements. However, the sender may be forced to send smaller segments than MSS bytes. The latest TCP congestion control standard[4] does not define how a receiver must respond to such packets. The standard let the receiver freely choose an appropriate response.

TCP implements both flow control and congestion control, while UDP implements neither of these.

A.1.1 Explicit Congestion Notification (ECN)

The Explicit Congestion Notification (ECN)[60] is an update to the IP and TCP header. The moment the network element runs out of buffer

space is packet drop for incoming packets certain and packets continues to drop until the previous packets start to dequeue (FIFO tail drop). Packet loss is an indication of congestion based on a timeout from the sender's perspective. The disadvantage of only using packet loss as an indication of congestion is the overwhelming queue that has buildup before the loss event took place. Queue got huge since the senders went too fast for a long enough period. ECN tries to solve the enormous queuing scenario with the use of explicit congestion signal, but help from network elements is needed. The network element currently bottlenecks in the path must let senders be aware of the congestion so they can reduce their transmission rate before the queue get a chance to build. A network element who actively tries to control its queue is generally known as an Active Queue Management (AQM) (see Section A.4). The AQM must remain stateless for the solution to be scalable.

The IP header updates with two bits; these bits are used to tell if the endpoint supports ECN for this particular packet. These bits are also used by the AQM to notify about congestion in its queue. The hosts set these bits to either the value 2 (ECT(0)) or 1 (ECT(1)) to indicate that the host supports ECN for this packet, the recommended approach is to use of ECT(0) so its backwards compatibility with earlier version of ECN[59]. Further, an AQM in the path will set these bits to the value 3 (Congestion Encountered (CE)) when the packets end up being in the buffer for more an extended period. An AQM only sets these bits if the host supports the use of ECN, in other words, the value of these bits must not yield 0 (Not-ECT)). The reason for using bits to detect if the host supports ECN is to be able to deploy ECN incrementally; this extra information let the AQM take the correct decision regarding ECN capable and other hosts. For the ECN-Capable hosts, an ECN mark is enough to signal congestion, but only a packet loss gives the same effect on other hosts. This functionality is, therefore, needed in networks where it may coexist a mix of ECN-Capable and other hosts.

The use of ECN in TCP is optional and is decided between the sender and the receiver when they initiate the connection (segments with the SYN-bit set). Two bits are used in the TCP header to signalise ECN-Echo (ECE) and Congestion Window Reduced (CWR). The very first segment that is initialising the TCP connection has both the ECE and CWR bits set to indicate that the TCP sender is ECN-Capable. The use of ECE-bit here makes the sender committed to responding to the CE code point, and likewise, the sender sets the CWR-bit to promise to make an appropriate reduction of the transmission rate upon receiving a segment with the ECE-bit set. The receiver responds to the very first segment with a segment, SYN-ACK segment, with only the ECE-bit set to tell the sender that it will participate as ECN-Capable host. The host may not change its commitment later for this TCP connection.

The receiver sets the ECN-Echo (ECE) bit on a segment to signal congestion to the sender and does so by inspecting the IP header for the CE codepoint. The sender transmits the next segment with CWR-bit set to acknowledge that the sender has received the segment with the ECE-bit set, and thus acknowledging the appropriate reduction of the transmission

rate. The receiver continues to transmit any new segments with the ECE-bit set until the segment with the CWR-bit set is received.

Senders are now more aware of how congested the network is and does not observe packet loss anymore due to congestion. The sender reacts the same way to ECN mark as it would to a packet loss, and that is a reduction of the transmission rate by half (see Section A.2). An ECN mark is usually only triggered once in an RTT and is also how NewReno responds to congestion signals (see Section A.3.4).

A.1.2 Stretch vs. Delayed Acknowledgments

Stretch acknowledgements are used by a TCP implementation to reduce the number of acknowledgements sent by the receiver. A stretch acknowledgement means that the receiving host acknowledges every second full segment received[4, 12]. Usage of stretch acknowledgement is not a requirement per the TCP standard but merely a recommendation for better utilisation of the network. Stretch acknowledgements save on the bandwidth and processing cost throughout the network since fewer control packets are in flight per RTT.

The receiver is required to timeout the stretch acknowledgement to prevent deadlock in situations where the sender cannot transmit more segments (see Figure A.2). The TCP standard, therefore, requires the receiver to not wait for more than 500 ms before finally returning the acknowledgement to the sender. Although, most implementations initiate the timer to no more than 40-200 ms. The receiver has, in this case, transmitted a delayed acknowledgement. Although, the receiver still returns the acknowledgement immediately, as one data segment, if both hosts participate in the communication. The delayed acknowledgement mechanism is the reason why the sender likes to maintains a congestion window of a minimum of two whole segments, or 2*SMSS bytes, per RTT. The sender then avoids delayed acknowledgements and can generally expect stretch acknowledgements to arrive within an RTT (see Figure A.2a).

Delayed acknowledgement mechanism is very problematic as the sender has to send at least 2*SMSS bytes per RTT, and does not allow a shallow RTT in the network. A shallow RTT network may require the sender to keep a congestion window of less than two whole segments per RTT. The sender cannot choose to be less aggressive since the receiver would end up delaying the acknowledgement in hope for more bytes to arrive (see Figure A.2b) and the sender may risk waiting for a very long time. Meanwhile, there is no extra delay to the RTT had the receiver not used stretch acknowledgements and thus not delayed its acknowledgements (see Figure A.3), but this results in lots of control packets in the network. Control packets which will consume resources of the network and only extend the sender to go as low as one segment per RTT in scenarios where the sender continues to clock out new segments based on the acknowledgement clock. The assumption here is that the sender would never want to send segments containing less than SMSS bytes where transmission of a full segment is possible. Transmission of

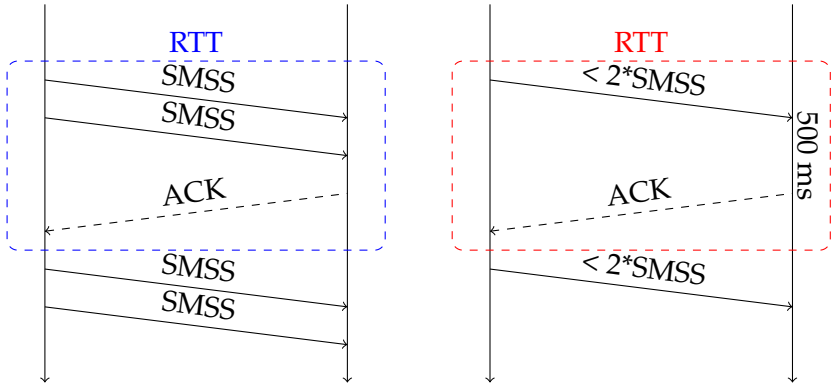
smaller segment impacts the goodput of the application since there is more overhead from network headers compared to useful data per packet. Meaning, the sender should not use its acknowledgement clock to clock out new packets in this environment as it would yield terrible performance as the RTT get smaller in the network. We call such an environment the sub-mss regime. The sender is said to be in a sub-mss regime if the congestion window of the sender fall below the stretch acknowledgement factor found at the receiver. TCP does, therefore, not perform well in the sub-mss regime and avoid the regime by keeping its congestion window at a minimum of the stretch acknowledgement factor of the network.

A stretch acknowledgement violation happens when an implementation tries to keep a more significant stretch acknowledgement factor than two. As discussed earlier a higher stretch acknowledgement factor is beneficial to a variety type of network topologies. However, a TCP implementation is said to have violated the principle of stretch acknowledgement if no acknowledgement returns after two segments received. The problem is so widespread that it got its name and is called stretch ACK violation. It is a generally a known problem, and it has been an open issue for a very long time[54] (Section 2.13). Although the latest TCP standard for congestion control does not prohibit the implementer from using a more significant stretch ACK factor, the standard clearly warns the implementer of degraded performance. The penalties come from the direct fact that the sender falls into the sub-mss regime. The sender is unable to send enough segments to trigger the acknowledgement. The sender has no other choice but to wait for the delayed acknowledgement to arrive. The sender, while waiting for the delayed acknowledgement to arrive, assumes the worst and initiates loss recovery and gets repetitive occurrence of timeouts. The sender has to reprobe the network and yield poor performance as a result. One could argue that in controlled environments, such as data centres, it is easy to configure the sender from falling into the sub-mss regime, but this only moves the problem to higher bitrates. We have successfully reduced the processing cost of the sender and the network but at the cost of higher RTTs. The problem gets even worse in not so controlled environments as a negotiation between endpoints is needed to choose an appropriate stretch acknowledgement factor.

A.1.3 Nagle's Algorithm

Nagle's algorithm[50] by John Nagle became part of the TCP standard[12]. The purpose of Nagle's algorithm is to avoid having a network with many small successive segments from the sender. For each packet, there is overhead added from the TCP/IP stack headers which consume bandwidth. Therefore it is better to send whole segments than many small segments. This algorithm has shared goal as usage of stretch acknowledgements which is to have fewer segments inflight per RTT to save the limited resources of the network.

Nagle's algorithm sends the segment always immediately if there are no previous segment inflight. Any further small segments do not enter



(a) non-sub-mss (b) sub-mss

Figure A.2: Delayed acknowledgement: enabled

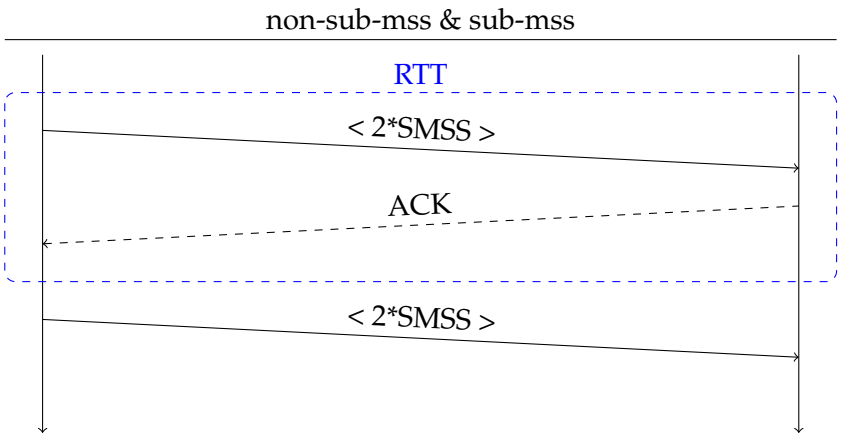


Figure A.3: Delayed acknowledgement: disabled

the network as long as there are some previous segments inflight. The segment is instead buffered up by the sender and held until a whole segment is ready for transmission. The algorithm transmits the pending small segment whenever there are no segments inflight. Meaning, the sender may send one small segment for every RTT. This algorithm tries to preserve bandwidth consumption in the network with the assumption that the application will continue to send many small segments, e.g. Telnet[55].

Nagle's algorithm has unintended effect when used together with stretch acknowledgements. The application data may split up into two or more segments, and the algorithm ends up buffering the last small segment. Usage of both Nagle's algorithm and stretch acknowledgements causes a soft deadlock between the two hosts. The delayed acknowledgement eventually returns which in return allows the sender to transmit the last missing segment. The whole process has caused the application to wait for extra 40-500 ms for the response, and this limits short-lived flows which depend on a quick response from the server (e.g. web-browser).

An alteration was suggested to Nagle to delay the last segment only if the previous segment was also small[48]. Now when the application data split into serval segments is the final segment sent immediately, while if multiple small chunks of data are sent within an RTT by the application does the algorithm have the same behaviour as earlier. Nagle's algorithm could also be disabled per socket basis using the TCP_NODELAY option by the application.

A.1.4 Selective Acknowledgment (SACK)

Selective Acknowledgement (SACK)[46] is used in the recovery phase of the standard TCP to recover from loss efficiently. The sender uses the additional information appended to the acknowledgement to make intelligent decisions in which segments to retransmit in the recovery phase.

The use of SACK is only permitted if both hosts during the connection setup allowed its use, using a two-byte TCP option in the TCP header. The receiver reports segments which arrive out-of-order. The receiver uses a TCP option in the TCP header to report these segments; each segment contains contiguous sequence space, the receiver reports the left and the right edge of each segment. The left edge is the sequence number of the first byte in the segment and where the right edge is the sequence number of the next byte after this segment. The receiver can report up to 4 such entries (blocks) in a single packet; the exact number depends on the use of other TCP options since the TCP option space is limited. The sender can from this information learn which segments have been successfully received by the receiver out-of-order and know which segments to retransmit.

Duplicate-SACK (D-SACK)[28] is an extension to SACK and can be used to report duplicate segments received (spurious retransmission). Each segment may contain one single D-SACK block. When the D-SACK block is a subset of larger contiguous sequence space is the next block used to report this massive block. Remeaning blocks are used as before to report segments which have been received out-of-order.

A.2 Congestion Control (CC)

Congestion Control (CC) is very fundamental to how senders on the Internet work together in harmony to use the limited resources efficiently. CC is a way to converge a set of consumers to share the limited resources evenly.

The very first TCP standard[56] had a description of flow control, but not any detailed description of a congestion control mechanism. The receiver would return the flow window found in the TCP header to control the transmission rate of the sender. The TCP standard did, however, define Retransmission Timeout (RTO) which was used to retransmit lost segments. A segment would be copied to the retransmission queue as part of the transmission process and taken out once the segment was acknowledged. The timeout had a lower and upper bound and was calculated based on the smoothed RTT with a delay variance. The sender would retransmit the very first unacknowledged segment once the timer expired for a given segment in the retransmission queue. The sender then reinitiates the RTO.

TCP implementations based on this simple transmission logic lead to congestion collapse[50] in networks. Congestion collapse occurs when the RTT suddenly increases, e.g. a large transmission is initiated, and is high enough that the sender retransmits segments since the timer keeps expiring (RTT > timeout). The estimate of the timeout is not updated fast enough by the sender. The sender continues to retransmit segments and generates multiple copies of the same segments in the network. These packets are likely to be dropped or continue to keep the queue filled. The receiver receives lots of duplicate segments and eventually some new segments. The goodput of the sender remains severely degraded and leads to a network with significantly reduced overall goodput.

A better way to control each sender was suggested in a paper by Van Jacobsen[39]. The proposed solution preserved the flow control mechanism and sent segments using an acknowledgement "clock" to keep a fixed amount of segments inflight per RTT. The acknowledgement clock was initiated using the new algorithm *slow start* (see Section A.2.4). The RTO calculation was updated to use an exponential backoff timer for segments in the retransmission queue, and the sender now had to reduce its transmission rate upon detecting packet loss. The paper also defined the *congestion avoidance* algorithm (see Section A.2.5) to probe for capacity in the steady state of TCP. These algorithms become part of the next TCP standard[12], and all TCP implementations are now required to implement these two algorithms. Although the newest TCP congestion control standard[4] gives details for two additional algorithms that a TCP implementation should use to recover from a loss event: *fast retransmit* and *fast recovery* (see Section A.3 for a more detailed discussion on loss recovery).

The sender uses a slow start threshold (ssthresh) state variable to determine to either use slow start or congestion avoidance algorithm in the additive increase phase of TCP. The sender uses the slow start algorithm as

long as the sender has fewer segments inflight than the ssthresh value. The sender uses the congestion avoidance algorithm otherwise. The congestion avoidance algorithm, as the name predicates, is slower and probes for additional capacity in a more conservative approach. Value of ssthresh, therefore, decides how fast the sender probes for capacity. The loss recovery mechanism of TCP essentially slows the sender down by reducing the ssthresh value.

A sender initially set the ssthresh state variable to the maximum value possible (all bits to 1). The slow start algorithm is therefore usually initiated as a new connection is opened to explore a network with an unknown level of congestion. In some scenarios may the ssthresh value be set below the maximum value, but this is an optimation done by the TCP implementation. The TCP implementation reuses various values saved from an earlier flow. The goal is to initiate this flow as if it were an earlier flow. The success of this depends on the current state of the network. The condition of the network might not be the same as earlier. Therefore, this lookup is often set up as a cache where the entry is valid if it is recent enough. While such optimation is good and may yield better performance, is it usually turned off when benchmarking TCP. The result of the experiment is thus not biased from previous experiments.

A.2.1 Congestion Window

The Congestion Window (W_B) specifies the number of bytes the sender is allowed to emit into the network. A cumulative acknowledgement usually arrives after one RTT and tells the sender that the receiver has received a certain number of bytes. Such acknowledgement also tells the sender that these bytes are no longer inflight in the network and permits the sender to continue transmission of new bytes to what previously left the network. The sender uses cumulative acknowledgement to keep a fixed number of bytes inflight per RTT (acknowledgement "clock").

The congestion window can be visualised as a sliding window (see Figure A.4). The sender will for every RTT emit W_B new bytes (see Figure A.4a), the byte-rate of the sender (see Equation (A.1)). Transmission rate can thus be controlled by adjusting W_B.

$$x = W_B \,/\, RTT \qquad \text{(A.1)} \qquad\qquad r = W_s \,/\, RTT \qquad \text{(A.2)}$$

Alternatively, the sender may use the concept of whole segments, where each segment contains up to SMSS bytes (see Figure A.4b). Now the sender sends W_s segments every RTT , the senders' packet rate (see Equation (A.2)). The transmission rate can now be controlled by adjusting W_s.

The idea behind the segment based approach is the reasoning that transmission of segments which less than SMSS bytes is never a good idea. The sender always wants to send a whole segment if possible to minimise overhead from networks headers. A TCP implementation which implements the segment approach can easily enforce a congestion window

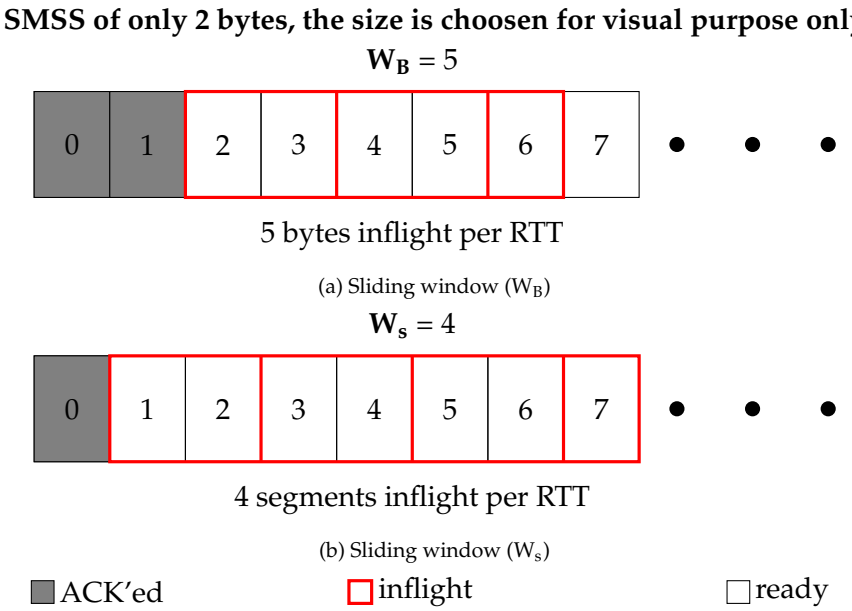


Figure A.4: Sliding window

of a minimum of one whole segments to keep its packets filled with whole segments. There may be situations where the sender is forced to transmit smaller packets, but the sender can work around this by using Nagle's algorithm. In contrast, a TCP implementation using the byte-based approach has to keep its congestion window at a minimum of SMSS bytes and only transmit whole segments to achieve the same benefits. There are, therefore, no apparent benefits of using the byte-based approach and the use of the segment-based approach allows a more straightforward implementation.

The congestion window is usually set to no lower than two segments since the receiver might be using delayed acknowledgements. The sender tries to send at a minimum of two segments per RTT to force an immediate acknowledgement from the receiver. The sender only uses a congestion window of one segment when the RTO expires (the loss window).

TCP standard[3, 4] defines an initial congestion window for a new connection between two and four whole segments. A modification to boost up the overall performance was an experimental standard[20] proposed to allow an initial congestion window of ten whole segments. Short-lived TCP connections benefit from this as they usually do not live long enough to maintain a large congestion window.

A sender updates its congestion window based on the feedback from the receiver. A cumulative acknowledgement let the sender either increase or decrease its congestion window. Growth in the congestion window usually refers to an acknowledgement in the steady state of TCP. Meanwhile, an acknowledgement in the loss recovery phase of TCP refers to a negative acknowledgement. The acknowledgement has a negative meaning since it results in a reduction of the congestion window rather than a growth.

Additive Increase Multiplicative Decrease (AIMD) schemes are widely used by senders to update their congestion window. The sender updates its congestion window based on the current congestion control algorithm employed. A usual response to no congestion is to increase the congestion window by one whole segment per RTT. The usual response to congestion, on the other hand, is to reduce the congestion window to half of its original size. Therefore, a sender with a large congestion window has a higher penalty than those with smaller congestion window. All senders using AIMD schemes eventually converge to have an equal share of the available bandwidth as their congestion window becomes equal.

ACK Division attack is an attack used against TCP implementations that increase their congestion window by precisely one whole segment for every acknowledgement received. The receiver abuses the use of partial acknowledgements and lures the sender to use a much larger congestion window. The sender emits more bytes than what is leaving the network. The malicious receiver breaks the acknowledgement clock of the sender. Protection against such attack is to increase the congestion window using Appropriate Byte Counting (ABC)[2] which is to only increase by the number of bytes acknowledged by the receiver, but no more than one whole segment.

A.2.2 Congestion Window vs Flow Window

Congestion window must not be confused with the flow window found in the TCP header (16 bits). Flow window is the receivers' storage limitation and updated with every acknowledgement. Flow window keeps the sender updated of how much storage capacity the receiver has at the moment. A sender may not send more bytes to the receiver than this limitation, bytes that are currently inflight also counts toward this limitation. Instead, the sender needs to wait for the acknowledgement from the receiver with an updated flow window and only then transmit more bytes if the new flow window allows it. The congestion window does not allow the sender to send beyond the flow window advertised by the receiver. The sender governs its transmission by the minimum of flow window, congestion window and senders' send buffer size.

A.2.3 TCP Friendly Rate Control (TFRC)

TCP Friendly Rate Control (TFRC)[27] may be used in place for window control to do congestion control. A TFRC unicast flow acts fair against other TCP flow using AIMD schemes.

TFRC uses a throughput equation based on congestion signals and RTT measurements. The sender uses feedback from the receiver for these measurements. In contrast to window control, A TFRC sender has a more stable throughput (small oscillations), but it is slower to adapt to changes in the network. TFRC is a good fit for applications doing video streaming. Meanwhile, TFRC is not a good choice if the application wants to reach as high throughput as possible. A TFRC sender normally uses a fixed

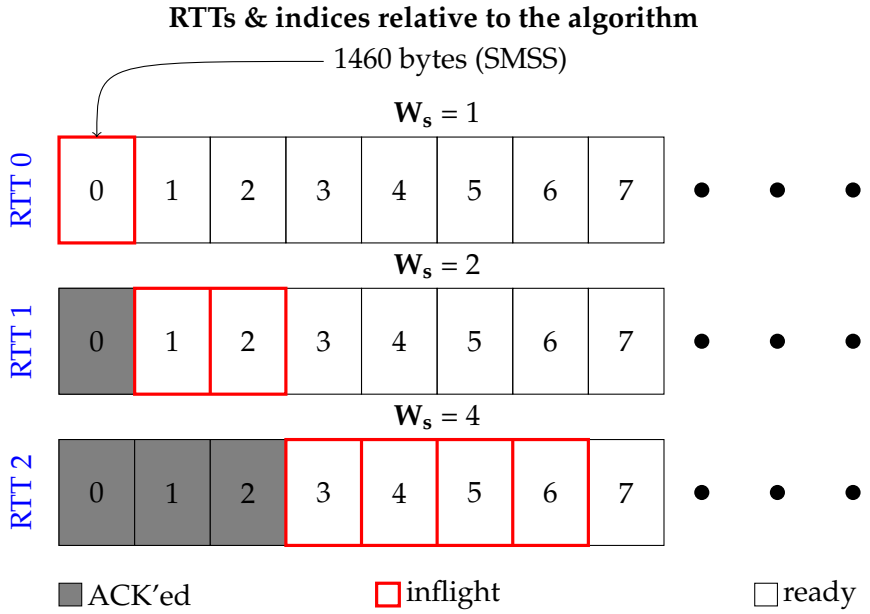


Figure A.5: Slow start sliding window (W_s)

segment size and modifies its sending packet rate per second to control the transmission rate. A sender should only use TFRC if a smooth throughput is a requirement, as a general congestion control algorithm using AIMD do rate halving.

A.2.4 Slow start

The sender starts by slowly probing the network using the slow start algorithm and initiates the acknowledgement "clock". The slow start algorithm increases the congestion window by *maximum* SMSS bytes, or one full segment, on each incoming acknowledgement. Although, the recommendation is to only increase by the number of bytes newly acknowledged by the receiver, N bytes, to defend against ACK division attack (see Equation (A.3)).

$$W \mathrel{+}= min(N, SMSS) \tag{A.3}$$

The slow start algorithm has therefore exponential growth of the congestion window for every RTT, doubling the congestion window every RTT (see Figure A.5). The slow start algorithm grows very fast with the goal to seek the available bandwidth as fast as possible and is prone to "overshoot" above the available bandwidth. This behaviour may result in queuing and bursty behaviour at the bottleneck.

A.2.5 Congestion Avoidance

The sender switches to the congestion avoidance algorithm when the congestion window exceeds the ssthresh value. The congestion avoidance

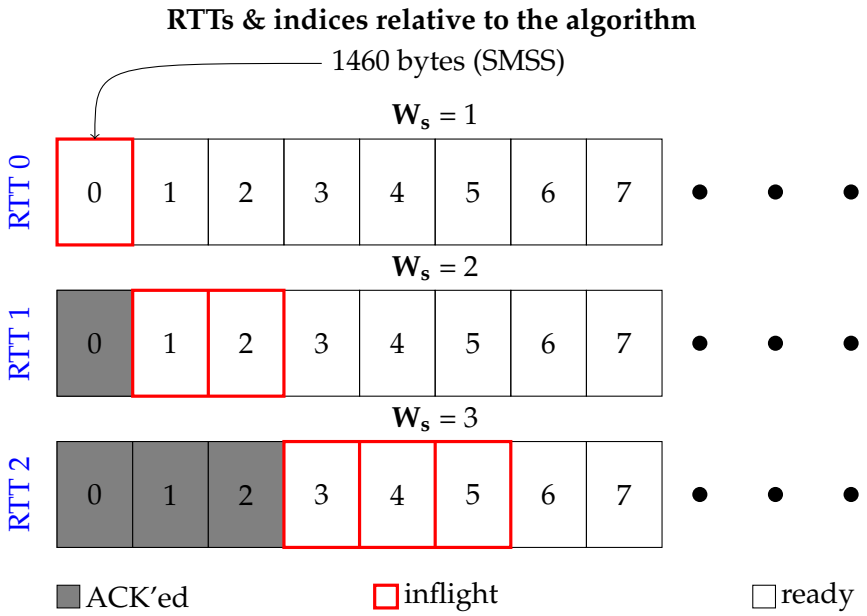


Figure A.6: Congestion avoidance sliding window (W_s)

algorithm is more conservative than the slow start algorithm. The algorithm increases the congestion window by one full segment for every RTT (see Figure A.6). The equation (A.3) from the slow start can still be used to update the congestion window. The only difference is that the equation is applied only once in an RTT, while in the slow start algorithm it is applied for every acknowledgement received in the RTT. It means that the congestion avoidance algorithm has linear growth of the congestion window for every RTT where the slow start algorithm had exponential growth of the congestion window for every RTT.

The sender can still update the congestion window on each received acknowledgement since for every RTT are W_B bytes inflight (see Equation (A.4)).

$$W_B \mathrel{+}= SMSS * SMSS \;/\; W_B \tag{A.4}$$

However, the latest congestion control standard[4] recommends updating the congestion window during congestion avoidance using an additional state variable. The number of bytes newly acknowledged increases the state variable and the congestion window is first increased by one full segment as the state variable is as significant as the congestion window. The state variable is then reset back to 0 for the next RTT. This method protects the TCP implementation against ACK Division attack, and ensure that the congestion window has linear growth for every RTT in situations where the receiver is using delayed acknowledgements.

Likewise, if the sender uses the concept of whole segments could the congestion window still update on each acknowledgement since there are W_s segments inflight. The sender updates the congestion window by $1/W_s$ for each acknowledgement received. The same strategy can be applied by

the sender to defend against ACK division attack. The sender accumulates the state variable whenever the receiver acknowledges reception of a whole segment. The sender increases the congestion window by one segment as soon as the state variable has acknowledged one window of segments inflight.

Congestion avoidance algorithm preserves the already obtained bandwidth and tries to claim for additional bandwidth until a congestion signal is received. The sender reduces its transmission rate through the loss recovery phase. The congestion avoidance phase repeats as the loss recovery phase completes. The nature of the congestion avoidance algorithm makes the algorithm rather inefficient for an enormous BDP path. The congestion window only increases by one full segment in an RTT no matter how large the BDP path is. In such situations, the process of increasing the congestion window from the reduced window back up to the limitation of the network is slow and requires lots of RTTs. The congestion avoidance algorithm is neither good for a low BDP path as add of one whole segment could be more than what the low BDP path offer. In other words, the congestion avoidance algorithm does not scale well for either a low or large BDP paths in the network.

A.3 Loss Recovery

The loss recovery phase is initiated by the sender to recover from lost segments. A lost segment is a segment which is assumed to be lost by the network. TCP uses different approaches to detect segment loss with varying efficiency. A loss is always eventually repaired by the Retransmission Timeout (RTO), but this is rather inefficient. The sender uses RTO as a fallback mechanism to reprobe the network when all other intelligent mechanisms have failed.

The TCP implementation usually implements fast Retransmit and Fast Recovery algorithms for an improved recovery phase. NewReno is an optimation of the fast recovery algorithm and allows an efficient recovery phase even with partial acknowledgements. A sender using NewReno must updates the ssthresh state variable to half the data inflight as soon as detecting segment loss. This update is essentially the Multiplicative Decrease phase in the AIMD scheme and is needed to make senders converge at the bottleneck. The sender only reduces ssthresh once for a given segment in the window. However, the sender must reduce its ssthresh value even further if the loss event happens after retransmission has taken place. The state variable ssthresh also have to be at least two full segments to interwork with delayed acknowledgements. The sender only uses the new value of ssthresh if it is above the lower bound (see Equation (A.5)).

$$ssthresh = max(inflight\ /\ 2, 2 * SMSS) \tag{A.5}$$

The sender has to limit transmission of new segments in the loss

recovery phase until there are less outstanding segments than the new value of ssthresh. The sender has to wait until the recovery phase is over and only then probe for more capacity through the congestion avoidance algorithm. The recovery mechanism also kicks in whenever an ECN mark returns from the receiver. The conventional TCP sees ECN mark as the same level of congestion as a loss event. Meaning, the sender reacts with the same response and reduce its transmission rate by half. (according to Equation (A.5)).

Selective Acknowledgement (SACK) allows the sender to recover from a loss very efficiently with the additional information the receiver append to the acknowledgement. Duplicate-SACK (D-SACK) can be used with SACK to detect spurious retransmissions. The use of Selective Acknowledgement (SACK) information is the recommended way to recover from a loss and should be used whenever possible.

A.3.1 Retransmission Timeout (RTO)

Retransmission Timeout (RTO) is used as a last resort if no acknowledgement returns from the receiver within a given duration. The use of Retransmission Timeout (RTO) was updated and made mandatory[53]. The sender has to restart the probing of the network given that condition of the network has changed, which is to update ssthresh and set congestion window to 1 (loss window). The sender reinitiates the slow start algorithm and switches over to the congestion avoidance algorithm as the congestion window reaches the new value of ssthresh.

Retransmission Timeout (RTO) specifies the duration a sender must wait before retransmitting the first unacknowledged segment. The timeout is set using Smoothed Round Trip Time (SRTT) which is calculated using RTT measurements. The sender must update the timeout every RTT (take at least one RTT measurement per RTT). Retransmitted segments can only be used for RTT measurement if they held the timestamp option, else the sender cannot distinguish the most recent segment from an earlier segments sent (acknowledgement could be from the segment that was assumed lost).

The timeout is set to initially one second when the sender lacks an RTT measurement. The sender is not allowed to initiate an RTO of less than one second. The sender must use a timeout of 3 seconds if the first segment was lost (SYN). The timeout grows exponentially for each time the sender retransmits using RTO[39]. The timeout reinitiates upon reception of a new RTT measurement (successful delivery). The RTO work as a "backoff" algorithm, the penalty grows for every time the sender fails to receive an acknowledgement from the receiver within the timeout. RTO helps avoid congestion collapse since the timeout grows at an exponential rate. The sender is forced to wait for longer durations. The sender eventually receives the acknowledgement as the timeout gets more significant than the RTT. New RTT measurement is used to reinitiate the RTO.

A.3.2 Fast Retransmit

A high loss indicator is when a segment arrives making a hole in the sequence number space. The receiver receives the segment out-of-order and cannot forward the segment to the application yet since part of the earlier data in the stream is missing. The arrival of out-of-order segments is, therefore, of crucial information to the sender. The sender needs this information to recover from a loss event efficiently.

The receiver uses duplicate acknowledgements to notify the sender about the out-of-order event. A duplicate acknowledgement acknowledges the very first missing segment rather than the incoming segment since the receiver is not allowed to acknowledge beyond the next expected byte in the stream. The receiver has, in this case, generated a duplicate acknowledgement instead of an ordinary acknowledgement. The receiver must not delay such acknowledgement so the sender can initiate the loss recovery phase as soon as possible. Acknowledgements that repair parts of the hole are also of importance as the sender needs to quick feedback in the recovery phase. The receiver must, therefore, immediately acknowledge any segment beyond the hole.

Although, it is not unheard of that a segment arrives earlier than the next expected segment due to a couple of reasons (reordering, loss, duplication). The sender does not initiate loss recovery right away on the first duplicate acknowledgement hoping for a false positive. The sender waits until at least three successive duplicate acknowledgements have been received for the same segment[4] before initiating the fast retransmit algorithm. This algorithm is the first phase of the loss recovery mechanism with duplicate acknowledgements.

The sender must not update its congestion window before the third duplicate acknowledges has arrived. Although, the sender may send an additional segment as a duplicate acknowledgement arrives since duplicate acknowledgement usually means that there is one less segment in the network.

Finally, when three consecutive duplicates acknowledgements have arrived can the sender with high probability assume the segment as lost. The sender reduces its transmission rate by reducing the state variable ssthresh (see Equation (A.5)). The sender now retransmits the missing segment and updates its congestion window to the new value of ssthresh + 3. The addition of constant 3 is for all the segment that left the network before entering fast retransmit. The sender then enters the next stage of the recovery phase known as the fast recovery algorithm.

A.3.3 Fast Recovery

The sender initiates the Fast Recovery algorithm after Fast Retransmit has retransmitted the lost segment. The sender *temporarily* increase the congestion window during the fast recovery phase, by one whole segment for any additional duplicate acknowledgement received[4]. The sender may send one whole segment of new data if allowed by the new congestion

window and flow window.

The next acknowledgement that receipts new data *should* acknowledge up to the last segment sent in response to the duplicate acknowledgement in Fast Retransmit. The sender should have repaired the hole and must, therefore, set the congestion window back to ssthresh before leaving the fast recovery phase. The congestion avoidance is then reinstated to reprobe for available capacity.

The sender has thus avoided the penalty of reinitiating slow start and time has also been saved by not waiting for the RTO to expire.

A.3.4 NewReno

The Fast Retransmit and Fast Recovery algorithm are often called Reno. NewReno[34] is an optimisation of the Fast Recovery algorithm in Reno.

The sender assumes the Fast Recovery algorithm repaired the hole in the data as an acknowledgement receipting new data arrives. The sender expects the acknowledgement to have acknowledged up to the last segment sent before the start of loss recovery. However, this is not always the case since the sender could have lost multiple segments, or the network could have reordered segments. In a situation like this, the sender only receives a partial acknowledgement. This partial acknowledgement acknowledges some but not all data sent before entering Fast Retransmit. The sender exits Fast Recovery upon reception of a partial acknowledgement thinking the hole has been repaired, but instead risk waiting until the RTO expires. The sender may recover from multiple segment loss by going into Fast Retransmit again (three additional duplicate acknowledgements received). However, the sender does not perform well since by going through Fast Retransmit again makes the sender reduce its transmission rate a second time. The Fast Recovery algorithm, therefore, yields terrible performance in conditions where partial acknowledgements are received.

NewReno does not declare the recovery phase over before the receiver has receipts all missing data before the start of the loss recovery phase. NewReno uses an additional state variable "recover" to keep track of the beginning of the recovery phase.

As the sender receives a partial acknowledgement, which does not acknowledge up to the state variable "recover", is the first unacknowledged segment transmitted. The sender then updates its congestion window to reflect the additional number of bytes acknowledged and sends another segment containing new data if permitted by the new congestion window. This approach ensures that the sender has a ssthresh amount of bytes inflight as the recovery phase completes. Finally, the sender deflates its congestion window as before and reprobes the network using the congestion avoidance algorithm.

A.3.5 Recovery with SACK Information

The receiver notifies the sender about segments that have been received out-of-order using duplicate acknowledgements with SACK information. This information can be used by the sender to learn about multiple lost segments in a single RTT, wherein the standard TCP recovery phase the sender only learn about one lost segment for each RTT. However, the sender must not be more aggressive, that is retransmissions of more segments, than what is allowed in the standard TCP recovery phase.

The receiver must specify the most recent segment that triggered the duplicate acknowledgement in the first block of the SACK information. The sender gets timely information from the receiver in the recovery phase. The sender can thus know which segment triggered the duplicate acknowledgement and that the segment must be within the cumulative acknowledgement number and SACK information of the first block. The receiver fills rest of the SACK blocks with information of the most recent isolated segments received. Meaning, that the sender will receive the first block repeated three times in three successive duplicate acknowledgements. Therefore the SACK information for each block is repeated up to three times. SACK is, therefore, robust when the network loses some acknowledgements containing SACK information. However, it is a bit different with D-SACK as it also takes the first block of the acknowledgement. The receiver does, therefore, not repeat the information about the spurious retransmission.

The sender still has to depend on the RTO as a fallback mechanism. The moment the RTO expires must the sender ignore any previous SACK information provided. The sender then has to reprobe the network and assume any segments above the cumulative acknowledgement as lost.

A.4 Active Queue Management (AQM)

Active Queue Management (AQM)[6] tries to eliminate the high queueing delay observed in data networks due to an enormous queue (see Section 2.2). The AQM evict packet from the queue as the queue starts to get out of control. The AQM may ECN mark packets if both hosts support ECN and if the host allowed its use on the current packet with the appropriate codepoint (see Section A.1.1). There exist serval AQM schemes with different approaches and complexities, and is an actively researched field. AQMs allows partial deployment on network elements found on the Internet.

An AQM usually emits a congestion signal when a packet sits in the queue for a more extended period. A sender does reduce its transmission rate as soon as detecting the congestion signal. Meaning, the AQM can suggest how fast each sender should go to achieve the desired queue length. The primary goal of using AQM is to have an input rate that corresponds to the output rate with a minimal queue. The idea is to let the rest of the queue be available for bursts. Another goal often for an

AQM is to lower the probability for global synchronisation between flows. The AQM does not want to signal all senders at the same time to prevent the syncronised sawtooth behaviour of TCP in the queue. The congestion window of senders converge and usually becomes the same values. The unwanted scenario is when all senders go equally fast as it then becomes period with a small queue and in other periods where there is no queue at all. The AQM usually employs a random probability to signal a packet after the queue has reached a certain threshold to desynchronise flows.

Deployment of AQMs seems to fix some of the queuing delay caused by the capacity seeking behaviour of TCP, but this brings up another issue which is most TCP implementation does not send less than two full-sized segments per RTT. The sender likes to send at minimum two fully-sized segments in an RTT to avoid the delayed acknowledgement mechanism found at the receiver. The sender purposely ignores any recommendations given by the AQM to keep the congestion window above the floor of two segments. This cause problem as the AQM tries to force a shallow RTT. No matter how many NACK the AQM emits are they ignored by the TCP traffic, so the AQM ends with a longer queue than selected. The sender refuses to cooperate with the AQM and becomes unresponsive.

A.4.1 Random Early Detection (RED)

Random Early Detection (RED) is an AQM scheme often used on the Internet and data centres. RED is implemented as a classless qdisc on Linux and allows configuration on a per-interface basis. RED uses average queue size in the decision of marking the incoming packet. RED drops packets if the configurable *ecn* parameter is not set or the hosts do not make use of ECN.

RED defines two configurable parameter *min* and *max*, both in bytes. Another configurable parameter *probability* is used to set the highest probability to mark a packet, the probability grows linearly up to this threshold. When the average queue length (in bytes) is below *min* RED has the same behaviour as a FIFO scheduler. RED calculate the probability to mark the packet as soon as the queue length exceeds *min*. The highest probability (*probability*) is used by RED as soon as the average queue length reaches *max*.

The maximum length of the queue is set using the configurable parameter *limit*, in bytes. RED drop any new incoming packets when the length of the queue grows this far and works same as tail-drop for a FIFO scheduler.

Appendix B

Source Code

The source code can be obtained from here: https://bitbucket.org/asadsa/kernel420/src/master.