Offloading pacing to hardware using Netronome NICs

Per Magne Kirkhus

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

Institute for Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2022

Offloading pacing to hardware using Netronome NICs

Per Magne Kirkhus

Offloading pacing to hardware using Netronome NICs

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

Printed: Reprosentralen, University of Oslo

Abstract

The innate behavior of TCP is bursty with a constantly varying density of packets being transmitted. This behavior is part of how TCP is, but as network speeds are increasing, this bursty behavior is causing more and more unnecessary pressure on networks.

An increasing number of modern TCP congestion control algorithms have a need for pacing out packets rather than transmitting as soon as possible. Some of the old algorithms also see performance advantages in employing pacing to well established solutions. Pacing may reduce pressure on network bottlenecks, improve bandwidth utilization, and enable senders to measure bottleneck capacity.

At the same time, faster and faster network links and network interfaces push the boundaries for CPUs trying to handle the flow of packets to and from the transport layer.

TSO was introduced to reduce CPU pressure by enabling the kernel to bundle packets into larger packets for the network interface card to process. This frees up the CPU by offloading work to the network interface card, reducing overhead and number of stack traversals.

As TSO has been put to widespread use, we have limited the possibilities of employing pacing where we want to. There is a discrepancy between pacing and TSO because TSO forces us to delegate responsibility to the network interface cards. The network interface cards are all about speed and quality and do not offer us control over transmit rates. They simply split the bigger bundles into smaller packets and push them all onto the wire as fast as possible.

Patches limiting these bursts have been introduced, but that does not resolve the incompatibility of TSO and pacing, and hence we are missing out on some unused potential in the collaboration between kernel and the network interface cards which could result in positive effects for networks in general.

Modern programmable network interface cards seek to enhance our ability to control the exact behavior of the cards. The ambition is to improve the capabilities of our networks and save more CPU cycles by enabling more offloading than before.

In this thesis we will explore the capabilities of the Netronome Agilio SmartNICs to see if it is possible to enable pacing in their hardware through the abstractions they offer. If so, it could help bridge the gap between pacing and TSO, and it could open up the possibility of new and more complex variants of pacing and congestion control in general.

Acknowledgements

What started out as a couple of reflections from Joakim Misund on the behavior of flows in DCTCP with TSO ended up with a year of investigations into the realms of TCP and pacing.

The funny thing is that I thought I understood congestion control in TCP before starting this endeavor, but I must admit that I still get surprised by the fantastically complex and various algorithms and distributed problems that arise in this world of TCP.

I am glad I reached out to Carsten and Joakim. The task has challenged me in many ways, and even though it has been a long journey, it has been incredibly educational, and some of it will follow me for years to come.

Thank you, Carsten, for all your great discussions, support and knowledge; A true legend at our institute, and I am proud to have worked with you both on this assignment and others.

A special thank you to you, Joakim, for all our conversations about Linux networking and all the nitty-gritty details of TCP. Your proofreading has been vital to how this thesis has turned out.

My decision to pursue my joy of programming and attend the institute is one of the best choices I have made. It has been an absolute joy thanks to my teachers and my fellow students.

Some of you will most likely be around for years to come, and I would like to thank you all for all the discussions, sparring and teamwork. Most of you know who you are, but I would like to mention some for the road. Perhaps too many some would say, but none of you should be excluded.

Jonas, Tobias and Oliver. The first fellow students I got to know well. We tackled the first few rounds together, and it was just a fun and great start.

Johanna and Ellina. Thank you for all the laughs, great discussions and walks. Who knew we would end up where we are today looking back at those first couple of years.

Lars. Thank you for your never ending passion for problem solving. Your relentless search for answers (or questions) is always an inspiration and joy to those of us who have been fortunate enough to have been there. The institute should be proud to have you on board.

Håkon. Thank you for all the talks we have had. Your presence and advices have been of great help support. Your curiosity, ambition and

ability to tackle any challenge is a delight.

Dan the man. One of the most heart warming and pleasant people I know. No matter what you do, you do it with all of your heart and all your attention. I am proud to be your friend. Thank you for all the talks.

And finally my partner in crime, Sarek Høverstad Skotåm. Chaotically brilliant with no limits. Things could never have turned out the same without you.

`PMS OS` will forever be my favorite OS, and Rust is in for at treat.

This thesis could have been finished a while ago, but thankfully my son changed the original plan a bit. Heidi and I have been so lucky to have had two incredible years with Florian, and I have been so lucky to have spent the last ten years with Heidi. This thesis would not have been finished had it not been for your support, Heidi.

I thank you, love you both, and look forward to the rest of our lives.

Lastly, I would like to thank my mother for letting me love numbers and teach me maths before I could read. Most importantly, thank you for making me always believe in myself.

In memory of Gustav.

Thank you for the time we got to spend together.

Contents

List of Figures

Glossary

ACK Acknowledgement. A signal/message that is passed between communicating processes, acknowledging e.g. that a packet is received. 9

AQM Active Queue Management. Algorithm for actively handling queues by dropping or marking packets before the queues fill up, the purpose of which is to reduce latency and reduce congestion. 25

BBR Bottleneck Bandwidth and Round-trip propagation time. Congestion control algorithm created by Google. Rate-based, using latency rather than loss to detect congestion. 35

BDP Bandwidth Delay Product. The product of bandwidth and round-trip time, resulting in the maximum amount of data that can be in flight. 15

BIC TCP BIC. Congestion control algorithm doing binary search in Congestion Avoidance state. 17

BtlBw Bottleneck Bandwidth. The smallest bandwidth along the path of a flows path. 14

CoDel Controlled Delay. AQM algorithm using sojourn time as metric. If a packet uses more time than some threshold to traverse from ingress to egress queue, the packet is dropped. 26

CUBIC TCP Cubic. Congestion control algorithm with a more agressive approach than older TCP algorithms.. 16

cwnd Congestion Window. A limit to how much data one flow can have in a network. Can be measurd in bytes or or packets. 13

DCTCP Data Center TCP. Aimed at reducing latency and increasing utilization with smaller queues by using ECN to respond not only to the presence of congestion, but the proportion of congestion.. 30

DUPACK Duplicate ACK. An ACK with the same sequence number or lower (same bytes acknowledged) as the previous packet. 9

ECN Explicit Congestion Notification. End-to-end notification of network congestion by explicitly notifying the sender; as opposed to implicitly notifying the sender by dropping packets. 26

ECN-bits The two least significant bits of the Traffic Class field in the IPv4- or IPv6-header. 30

FPC Flow Processing Core. Programmable processing core (Micro Engine) found on Netronome Agilio NICs used for match-action events. 66

FRR Fast Retransmit and Fast Recovery. Part of the TCP Reno algorithm, combining the Fast Retransmit from TCP Tahoe with Fast Recovery, reducing cwnd to half and continuing until missing ACK is received. 19

GRO Genereic Receive Offload. A more general variant of LRO not restricted to TCP/IP, and more restrictive in requirements on packets suitable for bundling (must have very little variation in TCP/IP headers, and MAC headers and TCP timestamps must be identical. 60

GSO Generic Segmentation Offload. Generic segment bundling performed by the kernel much like LSO, but not restricted to TCP. 59

LRO Large Receive Offload. Enabling much the same as LSO only receive side, bundling up packets before passing them up the TCP/IP stack. 59

LSO Large Segment Offload. Enabling the transport layer to batch up packets to 64KB packets that are sent down to the NIC to split into MTUs. 33

LSS Limited Slow-Start for TCP. Algorithm introduced in RFC 3742 with a more aggressive growth than additive increase, but slower than Slow Start. 17

MTU Maximum Transmission Unit. The size of the largest packet that a network protocol can transmit. 2

NIC Network Interface Controller. Also called Network Interface Card. Hardware component connecting an entity to a network e.g. via Ethernet. 2

PaC Paced Chirping. Variant of slow start using chirps with measured rate, trying to estimate bottleneck capacity. 22

PPC Packet Processing Core. Core found on the Netronome Agilio NICs working with parsing, acceleration functions, classifications and load balancing depending on the direction of the flow. 66

PRR Proportional Rate Reduction. Algorithm to ensure smoother and more accurate Fast Recovery by pacing out retransmissions rather than waiting out to half the cwnd. 48

RED Random Early Detection. Algorithm used in active queue management with tail drop in order to evenly distribute what flows get packets dropped, thus avoiding that some flows get all packets dropped and some other synchronization effects. It drops packets before the queue is full, and selects the packet to drop at random. 25

RSS Receive Side Scaling. An efficient way to distribute receive processing across different CPUs in a system. 59

RTO Retransmission Time-Out. A timer used to retransmit a lost packet due to lacking feedback from the remote data receiver.. 13

RTprop Round-Trip propagation time. The time a signal uses from one point to another, physically constrained.
At time t, $RTprop_t = RTT_t - n_t$ where n_t is the amount of noise/delays caused by traffic, queues etc along the path. 36

RTT Round-trip time.. 13

SACK Selective Acknowledgement. An ACK with the possibility of notifying a sender about non-consecutive data enabling the sender to resend specific packages that are lacking. 9

ssthresh Slow Start Threshold. A threshold used in slow start as a limit to where slow start should end and congestion avoidance should start. 13

TCP Transmission Control Protocol. Protocol designed to ensure in-order, reliable streams of data between two entities in a network. 1

TSO TCP Segmentation Offloading. Reduces CPU-usage in TCP by passing larger segments down to the NIC for further processing into suitable frames for the link (e.g. 1500 bytes for Ethernet). 2

UDP User Datagram Protocol. Connectionless communication protocol for transmitting packages in a network. 6

Vegas TCP Vegas. One of the first congestion control algorithm using delay rather than packet loss to detect congestion. 34

Chapter 1

Introduction

Transmission Control Protocol (TCP) made its entry into networks decades ago when networks were constructed on completely different scales than they are today. The change in scale has forced our protocols and algorithms to continuously change. What once were unproblematic aspects or side effects of our algorithms have since turned out to cause problems previously unforeseen. One aspect discussed a lot in today's research is the innate bursty behavior of classical TCP.

Bursts naturally form as TCP flows change their rate, wait for signals or batch operations to save time and resources. The resulting oscillations found in networks have been there since flow control was first introduced to TCP, but as bandwidths, speeds and distances have increased by several orders of magnitude, such bursty behavior appears to be more and more problematic to the performance of our networks. Fluctuations and unpredictable variance in network load may lead to under-utilized networks.

1.1 Pacing - a promising solution

Several solutions to tackling bursts exist (buffer size increase, limiters and other creative methods), but as will become clear, most of them function more like band-aids patching up the effects, rather than removing the problem itself. Pacing differs from these by trying to tackle the cause of bursts itself.

Pacing was first introduced to TCP to tackle a quite specific problem related to bursts forming in networks due to ACK compression, but the principal idea has since seen more general usage and investigations.

Pacing is a technique of actively introducing gaps in time between packets where we would otherwise have a negligible gap or no gap at all. On the face of it, this should provide a more even flow of packets and reduce the stress on networks. Upon further inspection, pacing can have several other intricate advantages, but also a few limitations and negative effects which are not present in non-paced flows.

1.1.1 The limitations of pacing

Pacing is intended to give more stable utilization and less latency fluctuation, but it also poses some challenges for both end-systems and intermediate nodes. The implementation of pacing is not necessarily trivial,[1] and because bursts have been an intrinsic part of TCP for a long time, a lot of the dynamics surrounding the flow of data has naturally formed around it. Changing it has some unexpected effects.

When working with multiple concurrent paced flows in data center networks with high bandwidth and shallow buffers, tests have shown that they may cause a synchronized bursting effect across flows and a worsened bandwidth-utilization due to a synchronized or delayed congestion effect.[20]

In end-systems, pacing may impose more CPU cost, and for very high bandwidths, simply being able to time gaps small enough proves difficult even to operating systems even though they are running modern CPUs with high clock speeds.

1.1.2 TSO

Pacing may also come into conflict with solutions made to rationalize CPU usage. One of these solutions, TCP Segmentation Offloading (TSO), reduces a lot of CPU usage by offloading work to the Network Interface Controller (NIC). This technique is used by most modern operating systems, but it is a well-known fact that using TSO does not mix well with pacing.

TSO is an offloading of tasks to the NIC, and by offloading we lose some element of control. The NICs are responsible for getting the packets out onto the wire, and they are specialized in doing so. The main focus has always been speed and quality, and it has been optimized over decades.

When using TSO, we batch up packets and deliver them to the NIC as larger segments. The NIC is then responsible of splitting up the segments into Maximum Transmission Unit (MTU)-sized packets and transmit them. It does this and it does it as fast as possible because that has been one of the main criteria for a well-functioning NIC. To have a working solution for pacing, the transport layer (L4) needs to maintain some type of control of what rate packets are transmitted at. The traditional way of achieving this is by having L4 control the flow of packets down to the NIC. The NIC then can transmit packets as fast as it receives them.

We see how these to interests collide. We can save CPU load by offloading the segments, but that would mean that L4 would lose control over the rate at which the packets enter the network.

[1] This depends on how the implementation is done (hardware or not), and what level of control one can expect to have in high performance networks. There is also the question of how dynamical the implementation can be. As with most data path programming, speed and dynamicity is typically correlated, but the use of SmartNICs shown in this thesis is showing promise of combining the two extremes

1.2 Pacing in hardware

One recent development in hardware is the upsurge of programmable hardware; more specifically programmable NICs (SmartNICs). These NICs allow us to program them dynamically with custom packet handling actions. This seems to open up a possibility of regaining control over the packet transmission rate while at the same time taking advantage of TSO.

Netronome has manufactured a series of SmartNICs that offer advanced abstractions for programming the NICs to a number of customized offloadable actions to boost efficiency and performance.

1.3 The goal of this thesis

The goal of this thesis is to explore the abstractions provided by Netronome to the Agilio SmartNICs to see if it is possible to implement pacing in hardware and have L4 control the pace at which packets are transmitted at the same time as TSO is enabled.

1.3.1 Outlining

TSO is standard in the most widespread operating systems and considered a necessity to keep CPU usage at an acceptable level in high speed networks. Pacing does not have the same status even though it has been gaining territory over time. We will therefore spend some time in part I mapping out the different aspects of congestion control and pacing, discussing what solutions exist and what advantages and disadvantages pacing has when used in networks of today.

This will motivate our decision and lead to part II where we explore the possibilities of pacing in hardware. Specifically we will look into the tools provided by Netronome to program their Agilio SmartNICs. They have several tools and abstractions aimed at making it as easy as possible to accelerate network with hardware acceleration.

The question is if the toolbox has what it takes for us to achieve an L4 controlled pacing that can contribute to modern TCP algorithms without having to disable TSO.

Part I

TCP congestion control

Chapter 2

TCP

The purpose of this thesis is to shed light on and provide new solutions and considerations to some of the challenges posed by how TCP works today. It is not the purpose of this thesis to give a detailed depiction of all the different algorithms and variations of TCP. That said; in order to ground the desire to attempt the solutions in part II, it is necessary to give an outline of TCP as a protocol, and then give a quite comprehensive walk-through and discussion of the different abilities and challenges of congestion control in TCP with special attention to the aspects related to bursts and offloading.

This first chapter gives an overview of TCP as a protocol. The next two chapters provide a more thorough explanation of the different parts of flow control and congestion control seen in TCP; how they have come into existence and what challenges they have encountered along the way. Some challenges have been resolved with new algorithms or modifications, while other still remain today.

2.1 The drunken mailman

Because networks are complex collections of many different entities (cables, wireless transmission entities (links), switches, routers etc.), sending a packet from one end system to another is not a trivial matter even though it is done all the time. Packets may get delayed, lost or damaged along the way, and communication needs to take this into account.

As far as networks go, the Internet is the biggest one today and serves as a good grounds for outlining the different layers of networking.

The Internet is based on the Internet Protocol (IP) for routing packets from one end system to another. IP is a protocol much like the postal system; an address system making it possible to find out where a packet should be sent.

While being much like the postal system, its mailman is more resemblant of a drunken mailman with nothing more than a best effort principle where nothing is guaranteed. Packets are sent towards the address of a machine, but they may get corrupted along the way, be

delivered in wrong order or simply lost. IP does not guarantee anything more than trying its best to deliver the packets to the address put on them. Any obstacles along the way will directly impact what one may expect to actually be delivered.

With IP being this unreliable, it makes it difficult to build applications or systems on top of it. If nothing is guaranteed, it makes all applications relying on IP unreliable as well. To deal with this, TCP is set up on top of IP[1] as a connection oriented protocol using IP.

2.2 Connection oriented protocol

TCP is a connection oriented protocol where the sender and receiver first agree upon having a connection and transmission between them before sending any data at all.[2] This as opposed to one of the other dominant network protocols today, User Datagram Protocol (UDP), which is a connectionless communication protocol.

Once a connection has been established, the sender and receiver keep communicating until all packets have been delivered in-order.[3] On the face of it this seems quite straightforward, but as will become apparent in the following chapters, it soon gets complicated to determine if a packet is lost,[4] when it should be resent, how many packets may be sent, and when a packet is successfully received. The internet seems to be quite simple and organized, but in reality it is a complex and at times chaotic collection of end-systems trying to communicate by sending out packets on an IP overlay with best effort precision.

2.2.1 Error detection and recovery

Packets are what we will define to be data of some size sent from one destination to another. TCP packets have a header[5] containing metadata for the packet and a payload containing the actual data intended for transfer.[6] As packets traverse a network they may get corrupted either by natural causes disturbing the bits in the packet, or by malicious entities altering the content intentionally. TCP guarantees integrity of data, enabling the receiver to check the contents of a packet against a check sum in the packet header.

[1] On top meaning that TCP uses IP to route packets, well aware of the limitations of its capabilities

[2] This is a generalization. Some modern protocols like TCP Fast Open send some payload with the SYN, but the connection is still made

[3] TCP guarantees to deliver the packets in-order

[4] If it is even possible to determine this for sure

[5] The packet header is the ”wrapper” sent at the head of the packet containing addresses, checksums, payload size and so on

[6] In this thesis we will mostly refer to these types of collections of data as packets even though lower layers usually refer to them as frames.

2.2.2 Multiplexing

The packet header also contains information enabling application multiplexing. Each end-system is commonly set up with one IP address used by other entities trying to communicate with it. One end-system may have multiple applications though, and to enable multiple applications to communicate at the same time, the TCP header provides room for port numbers.

A port number in a TCP header informs the end-system of what application the packet is destined for. This enables several applications on one end-system to communicate out through one IP address.

The flow between two end system applications is thus identified by a 5-tuple comprised of sender- and receiver-addresses, port numbers and the protocol used. This 5-tuple identification is used in both TCP and UDP for unique identification of flows.

Chapter 3

Flow control

TCP provides a connection oriented communication, but simply agreeing to have a connection is not enough to ensure that the packets are received as planned.

If one end-system is to send some amount of data to another end-system, it is a good idea to employ some sort of flow control. Flow control means the sender does not send more data than the receiver is capable of receiving at any given moment. To do this, the sender and receiver need to agree on either a specific sending rate or a specific amount of data that may be sent at any one time.

3.1 Variants

There are several ways to implement flow control, but they may be divided into two main categories:

- **Rate-based:** The receiver tells the sender what rate it may send at. This is communicated initially in the connection setup, and may be adjusted through the lifespan of the flow.
- **Credit-based:** The sender receives credits from the receiver telling it how much data may be sent at any one time. If there are no credits left, the transmission halts until otherwise informed.

Both of these flow controls provide a means to ensure that the receiver is not flooded with packets, but they have different use cases.

3.1.1 Sliding Window

TCP uses a variant of credit-based flow control called sliding window. In sliding window, the sender and receiver agree on a window of some size which tells us how much data may be in transit[1] at any one time.

By holding a window of data, the sender is able to keep track of which packets the receiver has received and which packets may be

[1] I.e. on the wire and not confirmed by the receiving end-system

deemed lost and need to be retransmitted. Keeping track of this is done using the header-field `sequence number`. The sequence number starts at an arbitrary value within the 32 bit range, and is set to indicate a number for the first byte of the first packet transmitted in the connection. For each packet, the number is incremented according to the number of bytes in the packet.

The window tells us how many packets the sender is allowed send into the network at any one time without receiving any confirmation from the recipient. Sliding window has the advantage of allowing a certain amount of data to be in transit at any one time rather than just allowing one packet at a time. It is like a frame defining the amount of data which can be sent, and the limits of it are defined through sequence numbers.

The frame slides when the first packet of the frame is confirmed. Hence the name sliding window.

3.2 Acknowledgments

The flow of information back from the receiver to the sender in TCP is done with packets called Acknowledgement (ACK)s. A pure ACK has no payload. It simply conveys information in the TCP header confirming the delivery of packets by returning the sequence number of the next byte to be expected.[2] An ACK acknowledges that individual or multiple[3] packets have been delivered As will become clear in the following sections, ACKs may provide us with a lot more valuable information about the network, but their very nature as messengers also comes with some limitations and room for interpretation.

Depending on how we use ACKs and what information they provide, we may categorize them and some of their effects with the following terms used throughout the thesis:

- **Delayed ACK**: A case where ACKs are not sent immediately for every packet received, but more rarely. Normally every other packet is ACKed. Often used in wireless networks
- **Duplicate ACK (DUPACK)**: ACK confirming the same contents as the previous ACK sent. DUPACKs serve the purpose of indicating that an out of order packet has been received, i.e. that an earlier packet is missing. More on their use in section 4.3 concerning Fast Retransmit.
- **Selective Acknowledgement (SACK)**: A modification to the original ACK used in DUPACKs to provide the sender with the extra information of which out of order packets that have actually been received, not just that a packet is missing. By doing so, we

[2] Or indicating a missing packet with duplicate ACKs
[3] Cumulative ACKs

avoid making the sender retransmit a lot of packets that have been successfully received (albeit being received out of order).

- **Stretch ACKs**: ACKs that acknowledge more than two packets. This is usually caused by ACKs being lost on their way to the sender, but can also occur by design (though that is not very common).
- **ACK thinning or ACK suppression**: A term much like delayed ACKs where some ACKs are discarded either by the receiver or elsewhere in the system to reduce the downstream pressure. This has shown positive effects in e.g. wireless networks.

As we will see in later sections, these various types of use cases each have their strengths and weaknesses. Either way, data delivery in TCP is confirmed by the receiver using ACKs in some way.

Chapter 4

Congestion control

Flow control is necessary to keep the receiver from overflowing, but between two end-systems there are usually some intermediate nodes that make sure the that the flow of packets reaches its destination.[1] Once the packets move out on the link on route to their destination, they may pass by several intermediate nodes along the way. If we do not take into account that these intermediate nodes have limitations just like the receiver, we risk ending up with the same type of problems that arise with a lack of flow control between two end-systems.

Because of this, we need some sort of control for the intermediate nodes as well. This control mechanism is called congestion control, and it will be the focus of this chapter.

4.1 Congestion collapse

The intermediate nodes in a network are normally routers or switches functioning as relays between end-systems. In the intermediate nodes the flow of data needs to be received, processed[2] and passed on to the next leg of the race. If the senders send at a rate higher than what the nodes are able to forward, packets may get lost (dropped).

The intermediate nodes must have some sort of buffers to handle incoming packets. These buffers normally exist both for ingress[3] and egress traffic and are usually referred to as *queues*.

Having buffers for intermediate storage[4] may reduce the amount of packets dropped, but if the difference is sustained, the buffers will fill up and the node will be forced to drop packets.

This issue was not accounted for in TCP until 1988 when Van Jacobson and Mike Karels published [24] as a result of investigations into a series of congestion collapses in October 1986 in the ARPANET connection between LBL and UC Berkeley. They experienced the

[1] That is, if we are to use some sort of connection-oriented protocol like TCP

[2] E.g. lookup to route the packet to the next node

[3] Ingress - entering or incoming. Egress - exiting or outgoing

[4] More on buffers and queue management in Chapter 5

throughput[5] dropping from 32 Kbps to 40 bps, and realized it was the lack of congestion control that caused it. The problem was that once congestion formed in the network, no action was taken, and as a result, the congestion worsened until a full congestion collapse was a fact.

The big congestion collapses of '88 were a result of packets overflowing the three Interface Message Processors (IMP) between LBL and Berkeley. The packets arrived at the IMPs at a higher rate than they could handle, which caused the buffers to get filled up. Once the buffers were full, the packets got dropped, and that caused those packets to be sent once more from the sender. In TCP a sender can potentially retransmit all the packets in its window, causing buffers to keep getting filled. Once that happened, more packets were lost and we got even more copies of packets being retransmitted.

This lead to the throughput dropping drastically, and eventually we had what we call a collapse. As long as no one slows down their sending rate (often referred to as *backing off*), the congestion persists. The majority of packets experience retransmissions, and the throughput stays at a minimum.[6]

4.2 Conservation principle

To tackle the issues described, Jacobson and Karels introduced the conservation principle; a flow on a TCP connection "should obey a 'conservation of packets' principle"[24], meaning that for flows 'in equilibrium', no new packet enters the network before an old one has left the network. This would ensure that congestion collapses like those in 1986 "would become the exception rather than the rule" [24].

By creating control mechanisms that enforce the conservation-principle, the big collapse may be avoided. As long as a whole window of packets is on the wire, no new packet enters the network until a time-out occurs (indicating packet loss) or an ACK is received. This natural pacing of packets where a new packet is sent when an ACK has been received, is called ACK-clocking.

Several algorithms were presented in [24]. Not all will be examined in depth here, but a couple of them are crucial both to the understanding of congestion control, and to making congestion control work even today. We first take a look at slow start, the algorithm that starts the ACK clock and tries to get flows up to speed. Once slow start has done the first ramp up, the sender needs to monitor the congestion level and try to close in on equilibrium. This is done in congestion avoidance; a state explained in detail after slow start.

[5]The amount of data transferred from sender to receiver at a time interval (usually per second)

[6]With several competing flows, one flow may actually experience its throughput dropping to 0 because another flow is sending or retransmitting at such a high rate that nothing else gets into the constantly full queues.

4.3 Slow Start

The conservation principle states that no new packet is introduced into a network before one leaves. An important element here is that this holds only for a flow 'in equilibrium'. Reaching equilibrium is a whole challenge on its own. We need get to equilibrium, and to get there Jacobson and Karels introduced two algorithms, the first of which was named the Slow Start algorithm:

- Add a Congestion Window (`cwnd`) to the per-connection state (this is common to all algorithms using sliding window, not only in slow start)
- When starting or restarting after a loss, set `cwnd` to one packet.[7]
- On each ACK for new data, increase `cwnd` by one packet.
- When sending, send the minimum of the receiver's advertised window and `cwnd` [24]

The algorithm seems simple enough, but it has some clever solutions. When loss is detected, instead of sending the same amount of packets per Round-trip time (RTT) (maintaining the congestion-level), the `cwnd` is reduced to one packet and slow start restarts. The Slow Start Threshold (`ssthresh`)[8] is set to half of the amount of data inflight in the network at the time when the loss was detected. Halving the size of the window means going down to the the last window size that got through the path without loss. Using this as a new threshold seems rational and safe.

When packet loss is detected, we react by reducing the window size. Importantly, this reaction is quite substantial; according to the original algorithm presented by Jacobson and Karels (named Tahoe), the reaction to loss (interpreted as congestion) is to reduce the sending rate to 1 packet per RTT. This eases the pressure on the bottleneck to avoid congestion building up further.

But packet loss is detected implicitly, not explicitly. The absence of an ACK is used as a detection method, and because of this we need to make an active decision as to whether a packet may be deemed lost or not. To have some confidence that a packet actually has been lost, we need to define the grounds for making the decision. According to Jacobson and Karels one of two mechanisms may be used:

1. **Retransmission Time-Out (RTO)** The sender keeps a timer for the packet. Once the timer has run out, the packet may be considered lost.

[7] Van Jacobson and Karels define the window in terms of packets, and so does Linux

[8] `ssthresh` is the threshold indicating when slow start should stop. Initially this "SHOULD be set arbitrarily high"[4]

2. **Fast retransmit:** For each packet arriving receiver-side, an ACK is sent. The ACK contains information about the last in-order packet received. I.e. if packet 1, 3 and 4 are received, but packet 2 is missing, both packet 3 and 4 will result in an ACK indicating to the sender that 2 is missing (the next in-order packet missing is packet 2). Once the sender has received three[9] such DUPACKs, the packet may be considered lost and retransmitted. That way a retransmit may be done with some level of confidence, but without waiting for a timeout.[10]

RTO is calculated using the measured RTT. It must be set high enough to avoid spurious timeouts occurring (premature timeouts indicating loss when the packet is simply delayed or has a different RTT than expected). Because of this, the wait for an RTO can be comparably long, and an RTO must be set to have a proportionate impact on the sender. It should restart slow start all over with a `cwnd` reset to 1 and a new `ssthresh`.

Using fast retransmit is not that invasive. It simply halves `ssthresh` and tries to retransmit the missing packet. In either case (if we restart slow start or do fast retransmit), if another RTO occurs or another triple DUPACK is received, the same series of events take place, reducing `ssthresh` once more. Slow start ends either if we reach `ssthresh` or if we do fast retransmit.

Once slow start ends, we enter a new state called *congestion avoidance*. This is where we employ the second algorithm presented by Jacobson and Karels, working our way up much more slowly, trying to maximize the bandwidth utilization whilst reacting to any new changes in the network. More on this in section 4.4.

4.3.1 Queue overshoot

Slow start was created to reach a state of equilibrium in a fast way without causing major congestion. It is still a vital part of TCP today and has done the job it was set out to do. The algorithm gets a flow up and running, but it achieves its goal in a somewhat coarse manner. This coarse manner also exposes the flow to a risk of missing the target quite massively. As the authors state;

> ”overestimating the available bandwidth is costly. But an exponential, almost regardless of its time constant, increases so quickly that overestimates are inevitable”[24]

Overestimating the Bottleneck Bandwidth (BtlBw) by using a too large window results in the bottleneck queues filling up until they eventually overflow and we experience packet loss. It is not necessary

[9] In Linux, this number is adjustable. When routes are unstable, it may be better to increase the number.

[10] Fast Retransmit was introduced in [24], but formally standardized in RFC-1122[13].

to hit the forwarding rate of the bottleneck with pinpoint precision (that is why we have buffers), but in the case of slow start, we may overshoot substantially.

The bottleneck has ingress and egress queues, the main purpose of which is coping with some variance in flow rate, but the capacity has its limits. The queues will fill up if the sender sends at a higher rate than what the bottleneck can process. Any difference in rate will cause queue changes. If the sender rate is higher than the bottleneck limit, the queues will get more filled (limited by the size of the buffer), and any difference where the sender rate is lower than the bottleneck limit will cause the queue to drain (limited by the buffer becoming empty). The longer the difference sustains or the bigger the rate difference is, the faster the queue will either fill up or get drained.

Slow start does an exponential increase until loss is detected, and because of this rapid increase, we risk ending up with a sending rate even more than twice the rate manageable by the bottleneck. As Misund et.al explain; if the flow is sending at a rate equal to BtlBw, then the next time around it will send at double that rate.[28] Add in the effect of the delay from the time at which the packet gets dropped until the sender realizes it is dropped, and we have a lot of potentially overshot packets. This overshoot results in full queues, delay and packet loss.

The intention of Tahoe was to reach a state of equilibrium and keep it there. By doing this one wishes to fill the pipe,[11] not the queue. Filling the queue causes congestion and latency. The wider and longer the pipe, the more the pipe can hold.[12] When bandwidths and distances increase, slow start may potentially grow further, and the amount overshooting the target may get bigger. As will become apparent later on, increasing the buffers does not solve the issue.

The main issue is that the algorithm itself does the increase in a coarse manner and the impact increases as the BDP increases. These considerations have led to some algorithms seeking to do slow start in new ways. I have chosen to highlight two of them, the first one being HyStart which is the slow start algorithm used by the current TCP algorithm in e.g. Linux and Mac OS. The second one is Paced Chirping; a brand new experimental algorithm under development showing promising results for both speed and queue overshooting.

[11] A pipe is another word for the path between two endpoints. When we talk about the size of the pipe, we talk about the Bandwidth Delay Product (BDP) of the pipe.

[12] BDP is the term for how much data the pipe is able to have in transit at any one time. It is calculated as the product of bandwidth and RTT:

$$BDP = BW * RTT$$

Pipes with high bandwidth and big RTTs (in other words, big BDPs are nicknamed Long Fat Pipes.

4.3.2 HyStart

Traditional slow start may potentially overshoot by as much as twice the amount accepted by a bottleneck. To reduce the risk of severe queue overshoot, the creators of TCP Cubic (CUBIC)[13] created a new slow start algorithm named Hybrid Slow Start. By measuring variations in inter-packet gaps (ACK train length) and variations in RTT (delays), the intention is to determine whether congestion is starting to build up without having to wait for packets being dropped. As we will see in the later sections, this type of early detection through measurement is being implemented more and more in algorithms of today. The idea seems promising, but it is not done without challenges. The solutions and challenges seen in Hybrid Slow Start are related to bursts and measurements. It will be enlightening to give a brief walk through of it.

The principal idea behind Hybrid Slow Start is reasonable. Congestion results in increased delay because queues get filled (higher queue occupancy), and each packet needs to spend more and more time waiting in queue as queue lengths increase. This may be detected by monitoring the RTT or by observing inter-packet gaps between ACKs. The challenge with such a solution is that it may react too early (spurious delay), or the measurements may be unreliable due to congestion in the path of ACKs rather than the data. The authors suggest two ways to do congestion detection in Hybrid Slow Start: *Inter-Packet Arrival* and *Delay Increase*.[7]

Inter-Packet Arrival relies on the bursty behavior of slow start. Because all the packets in a burst are sent back-to-back, the idea is that one may detect congestion by measuring the length of an ACK train (the sum of the interpacket gaps between a certain number of packets) when it arrives at the end-system after traversing the whole path. One may detect when congestion is starting to build up or whether the packets in flight are approaching the BDP [21] by observing an increase in the train length. The advantage of using a train is that it does not require a high-resolution clock (because we measure a whole train and need only clock the start and end of the train), and it is not that sensitive to small variations in delay seen when using e.g. only 2 packets. To avoid the train being affected by the burst size itself filling up a queue, only some of the first packets of the burst are used.[14] However, this method is susceptible to ACK compression, delayed ACKs and SACKs. It is dependent on ACKs arriving without compression or receiver interference. According to Ha et.al, delayed ACKs are not that big an issue (at least on Linux) because Linux uses quick ACKs for up to 16 initial segments in slow start to ramp up the speed. Quick ACKs here meaning that ACKs are not delayed but sent as quick as possible. Even so, there seems to be quite a few challenges with the other types of ACK variations.

The Delay Increase algorithm seems perhaps a bit more promising

[13] see section 4.4.3

[14] CUBIC in the current Linux version (v5.11.11) uses the 8 first packets

because it uses RTTs and, more importantly, changes in RTT. The main idea is that a substantial change in RTT indicates that congestion is starting to build up. When the change in RTT surpasses a predetermined threshold, Hybrid Slow Start exits slow start and enters congestion avoidance. But this may also be affected by RTT variation not caused by congestion; e.g. idle periods or aggregation (seen in Wi-Fi), or path changes.

Even when Hybrid Slow Start works as intended (if we disregard the challenges with inter-packet measurement and assume the RTT measurements give a fairly descent indication, it is not flawless. If several flows are using the same link, they have to fight for capacity. If a flow uses Hybrid Slow Start, it risks not getting its fair share because it backs off when the link capacity is reaching its limit. Thus, the more aggressive flows retain a higher share of the link capacity.[15] It may also be that there is congestion downstream (the ACK stream) which may be interpreted as congestion upstream. All in all, hystart has a high risk of exiting too early. Because of this, Hystart++ was proposed in an RFC draft in 2019 [7] where Limited Slow-Start for TCP (LSS) is used to bring the flow from the slow start exit point up to the point of saturation faster than traditional congestion avoidance, but slower than slow start. In sum we then go from Hybrid Slow Start to LSS to congestion avoidance.

Hystart++ is very new, but has recently been implemented in the Windows 10 implementation of CUBIC. Hybrid Slow Start is used in the Linux implementation. Even as early as in 2014, 46.92% of web servers used TCP BIC (BIC) or CUBIC [42], and CUBIC is the Linux and MacOS standard today. Hybrid Slow Start may have some limitations, but it is in use, and the idea of reacting early and avoid packet loss has some appealing advantages which will be discussed in further detail in section 5.1.

4.3.3 Paced Chirping

The main goal of Paced Chirping is much the same as Hybrid Slow Start, but it uses active measurements as opposed to the passive measurements used in Hybrid Slow Start.

Paced Chirping uses several series of packets (chirps) sent at a gradually higher rate to measure the capacity of the bottleneck. By measuring the change in queueing delay, the idea is that we can infer the capacity of the bottleneck from the inter-send gaps and the queueing delay measurments. By making the chirps not just small bursts but a series of packets sent at an exponentially increasing rate, the algorithm is able to measure the capacity of a bottleneck queue quite well in experiments over fixed links. The gaps between the chirps

[15] The advantage of being aggressive and merciless is prominent in several aspects when working with internet protocols. In some cases the more fair or kind algorithms never get a foothold at all in the wilderness that is the internet. More examples will be mentioned along the way

allow the queues to relax, and by gradually reducing these gaps, one is able to increase the utilization of the link without overshooting. One interesting case for this thesis is whether pacing in hardware will contribute to Paced Chirping. Pacing in hardware may provide better granularity, and that is of interest to the implementation of Paced Chirping and other algorithms.

Paced Chirping difficulties

As with Hybrid Slow Start, Paced Chirping is a delay based version of the slow start algorithm (albeit different in methodology and persistence). To do proper measurements, we need to make some assumptions about the flow of ACKs. The ACKs and their pace are the grounds for the measurements performed. The sender sends out paced chirps and measures the change in pace of the ACKs as they arrive.

The measurement therefore is sensitive to ACK compression and other events downstream resulting in ACKs being delayed, reordered or lost. Some challenges also arise when faced with discontinuous links like Wi-Fi , docsis and mobile networks where aggregation and other deviant behavior is observed. This poses difficulties for algorithms relying on measurement rather than loss. More work is being done to mitigate these challenges in Paced Chirping.

4.4 Congestion avoidance

Slow start starts the ACK clock and ramps up the speed of a flow. In classical congestion control algorithms, this is done until we receive three DUPACKs and enter fast retransmit, or if we have RTO and restart slow start. If we restart, we repeat the same steps until we either reach `ssthresh` or enter fast retransmit. In more modern algorithms like Hybrid Slow Start, we detect congestion early and end slow start.

Because of the coarse nature of classical slow start, Jacobson et al. suggested that the final approximation for capacity (finding the equilibrium) needs to be done a lot slower so as to not overshoot again. After doing slow start, we have an approximate idea of what the bottleneck capacity is. It should reside somewhere between the size of our `cwnd` when we encountered a packet loss, and half of that. To find the saturation point somewhere between those to points, they proposed to do what they called an additive increase.[16] Jacobson and Karels named this state *congestion avoidance*, and it is the second central algorithm to the congestion control presented in [24]:

1) On any timeout, set `cwnd` to half the current window size (multiplicative decrease).

[16] Linear increase of the `cwnd`

2) On each ACK for new data, increase `cwnd` by $\frac{1}{\texttt{cwnd}}$ (additive increase).

3) When sending, send the minimum of the receiver's advertised window and `cwnd`.

The biggest difference between slow start and congestion avoidance is in 2). On each ACK we increase the window with 1/`cwnd` rather than increasing it by 1. By doing this, we effectively increase the window with 1 each time a whole window is ACKed.

This solution proved to work very well. It made it possible to get closer to a state of maximum utilization than what was possible with slow start alone. Slow start and congestion avoidance work together as a complementing pair where the first one does the dirty work of getting us into the same ballpark, while the other one seeks to utilize the last half of the potential as gently and widely as possible.

Tahoe solved the initial problems discovered in '86, but some obvious limitations and areas of improvement were discovered in the years to come, causing several different algorithms to spawn.

4.4.1 Fast recovery

According to Tahoe, slow start should restart every time an RTO occurs. This may result in under-utilizing the capacity because we need to ramp up from 1 to the new halved `ssthresh` before we can enter congestion avoidance. If we have an indication that the ACK clock is still ticking (packets leave the network and cause ACKs to be sent), reducing `ssthresh` to 1 seems a bit extreme.

These observations led to TCP Reno being introduced in 1990. Reno is very similar to Tahoe in many respects, but it reacts differently to DUPACKs. Tahoe restarts slow start with a reduced `ssthresh` upon receiving 3 DUPACKs. Reno also reduces `ssthresh`, but it does not set `cwnd` to 1. Instead it halves the `cwnd` and does fast retransmit. For each successive DUPACK it keeps increasing the `cwnd` with 1 because a DUPACK indicates that the receiver is still receiving packets with higher sequence numbers, allowing the `cwnd` to increase. Once the missing packet is ACKed, Reno enters congestion avoidance and does additive increase. This modification is called Fast Recovery and is associated with Fast Retransmit.[17] In short, both Reno and Tahoe do Fast Retransmit, retransmitting the lost packet without waiting for RTO, but Tahoe drops its `cwnd` to 1 and enters slow start while Reno does Fast Recovery with `cwnd` halved.

Reno enters Fast Recovery when it detects packet loss (through DUPACKs) . One problem with Reno is that it exits Fast Recovery when the missing packet is ACKed, and this may cause bad performance if there are multiple packets lost in the same window. We risk performing FRR several times before the packets lost in the same window are ACKed. This observation led to TCP New-Reno which handles multiple

[17] Often referred to as Fast Retransmit and Fast Recovery (FRR)

packet losses in one window by staying in Fast Recovery until all the packets in transit from the original window are successfully ACKed.

4.4.2 Improving New-Reno with SACK

One challenge with TCP New-Reno is that the loss-handling is quite slow. The sender needs to be informed about each of the lost packets individually. In other words a lot of RTTs are used to finish Fast Recovery. The introduction of SACK made it possible to convey information about several packets at the same time. A list of packets lost may be sent at one time, and the sender may retransmit them at once. The trouble with SACK is that it has some more overhead and this may impair the speed if the `cwnd` is very big or we have large bursts of packets lost.

4.4.3 BIC and CUBIC

As network capacity increases we need more time to ramp up to maximum capacity during slow start and congestion avoidance, and that makes us more likely to encounter packet loss along the way. Traditional loss based algorithms have some challenges when faced with big fat pipes because it can take a long time reaching equilibrium. Even if we have no congestion, we are likely to encounter packet loss due to packet error before we get up to speed using congestion avoidance. The congestion avoidance algorithm worked just fine in general for small, slow links, but today it poses a problem because it is too slow. We can easily end up with the risk of having issues even with the theoretical limit of network bit error rates.[18]

In addition to the need for a very long period of time with basically no packet loss, we can see that we spend a very long time moving from (theoretically) half capacity to full capacity. In other words, we have poor utilization all this time. Prior to BIC, several algorithms tried to mitigate this challenge by altering the rate at which the `cwnd` increases per RTT.[19] The idea was simply to ramp up faster, and it seemed to work well. The protocols adaptively increased their rates; e.g. STCP had a reduced multiplicative decrease and a defined ramp up of the `cwnd` of 1/100 ACKs instead of 1/`cwnd`. As the window increases in size, the size of the rate increase ramped up because the amount of `cwnd`-increases became a lot more than 1 per RTT as the window increased in size. By doing this, the time needed to fill a big fat pipe was greatly reduced. The solution was effective, but doing it this way would also turn out to favor flows with short RTTs rather than long, and it risked larger overshoots.

Shorter RTT means shorter time intervals between each increase. In the end we get an unfair distribution of bandwidth where the flows with short RTTs get a lot more bandwidth than those with long RTTs. It

[18] If RTT is 100ms we need 1 hour to ramp up from half to full utilization on a 10Gbps link with a loss rate of less than 1 per 2.6 Bn packets[27]

[19] SABUL, FAST, High Speed TCP (HSTCP), and Scalable TCP (STCP) [27]

may even lead to starvation. Another RTT unfairness stems from how synchronized loss has a higher probability of hitting flows with large windows, but it turns out that flows with shorter RTTs recover faster than flows with longer RTTs even though they may have much larger windows and greater loss than those with long RTTs.[27]

Binary Search Increase

Xu et al. sought to achieve better bandwidth utilization in high bandwidth networks while at the same time managing to avoid the RTT unfairness discovered in some previous solutions to the problem. Their solution (BIC) was to do what they called a binary search increase; Once congestion avoidance starts, we do a binary search for capacity rather than the traditional additive increase. thus significantly reducing the time needed to reach full utilization.

Binary search increase uses the `cwnd` present before halving as a maximum window size W_{max} and `cwnd` after halving as minimum window size W_{min}. Then it tests the capacity by setting `cwnd` to the midpoint between W_{max} and W_{min}. If another packet loss is encountered using the midpoint, the midpoint is used as the new W_{max}. If not, the midpoint is used as the new W_{min}. In either case, this midpoint testing continues until the distance between W_{min} and W_{max} is less than some predefined threshold S_{min}.

Doing this binary search reduces the impact of RTT on fairness because the number of RTTs needed to reach the desired capacity is reduced to log_2 of the bandwidth capacity rather than a linear increment. Furthermore, the increase function is logarithmic; it is halved in each step, thus reducing the potential overshoot as we approach the saturation point. The algorithm also combines the binary search increase with an additive increase called window clamping. If the distance to the midpoint is very large (larger than a preset threshold), it does an additive increase until the distance is within the threshold. By doing so it reduces the risk of adding too much stress on the network.

CUBIC

BIC ended up being the Linux standard because it proved to be very stable and efficient. Because the binary search increase is a logarithmic concave function, its overshooting potential is reduced for each step it narrows in towards the saturation point as opposed to a linear function (additive increase) which increases at the same rate the whole way, or (at worst) an exponential/convex function increasing the rate.

If the bottleneck capacity suddenly increases, BICs binary search will not find the new max just by using the previous max. Once it reaches the previous max without any packet loss, it shifts to a convex function called *max probing*, ramping up (much like a slow start from the previous max) to find the new point of saturation.

This combination of a convex and a concave function is replaced in CUBIC using a cubic rather than a binary function to find the saturation point. It is very much like BIC in having both a convex and a concave part,[20] thus maintaining the same stability with regards to treading gently around the saturation point while also being very efficient in finding the saturation point. The algorithm displays the same RTT fairness as BIC because the methodology is similar.

CUBIC removed the window clamping because testing proved it to be unnecessary. This reduced the complexity of the algorithm, as did shifting to a cubic function rather than two different functions. It also turned out that BIC was too aggressive in some situations. If the RTT was small, BIC would score badly on TCP friendliness.[21] CUBIC sought to improve on this by introducing TCP mode; a mode where the increase of the `cwnd` per ACK is equal to that found in standard TCP. To determine what mode the algorithm is in, CUBIC measures the time from when congestion avoidance was started (i.e. the time passed since the packet loss triggering the state). By knowing this time, it may calculate a classical additive increase to measure against. Three modes exist in total; the TCP mode if needed to be friendly, then the concave mode once we close in on the saturation point and are within TCP friendly bounds, and finally the convex mode if `cwnd` surpasses the W_{max}.

4.5 Limitations of loss based detection

According to Jacobson and Karels, packets may get lost due to damage in transit or congestion, but congestion is the main cause ($>$99% [24]), and thus seemed to be a fair parameter to use. Using dropped packets as the sole indicator for congestion (loss based congestion control) has worked just fine for a long time, but with the emergence of faster and longer links it has become more and more apparent that the loss based approach seems to fall short in some areas. Hybrid Slow Start and Paced Chirping (PaC) try to deal with exactly this recognizing that queue overshoot is very costly to all the flows sharing a bottleneck. It leads to higher latency and more packets getting dropped; which in turn leads to flows backing off and retransmitting packets that could have already been delivered had the queue not been put under too much pressure in the first place. As section 4.4.3 showed us, as the bandwidth increases, the time spent to grow linearly to the saturation point increases with the classical congestion control algorithms employing linear increase. The more time spent in this state, the higher the risk of packet loss leading to further decrease of flow rate. Thus, the bigger the BDP, the more important it seems to avoid packet loss in the first

[20] A cubic function (or any odd order function) has the property of having both a convex and concave part

[21] TCP friendliness is a measure for how greedy an algorithm is when competing with classical TCP algorithms

place. BIC and CUBIC tackle this by increasing the speed at which they ramp up, but we still have a situation where we spend a lot of the time under-utilizing the network capacity because we have a repetitive pattern where we constantly:

1. Fill the pipe until packets are lost
2. Back off to let the queues drain
3. Start increasing the rate again

This produces the classical saw-tooth pattern in TCP shown in figure 4.1, and it is not an ideal situation when it comes to utilization or latency. It is also worth noting that using loss as the primary indicator of congestion implies that congestion can not be detected before queues overflow and packets are dropped. Full queues and dropped packets are signs that the state of congestion is at its worst, and repairing the damage is thus at its hardest. Congestion detection may be done differently, and the following paragraphs will explore some of the alternative solutions.

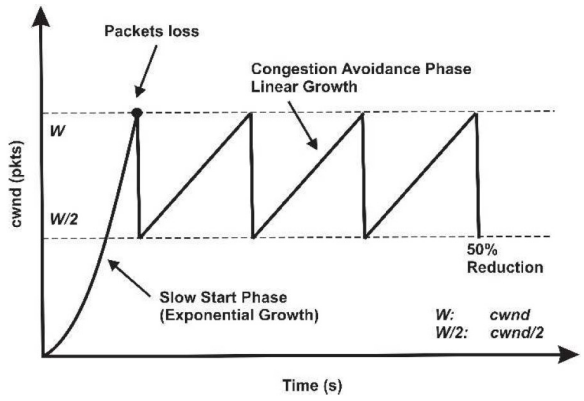

Figure 4.1: Dynamics of a classic TCP connection displaying saw-tooth pattern[2]

We must also remember that traditional loss detection is indirect loss detection. A lost packet can not let the sender know it is lost. The detection is done either through RTO where we simply ”give up” because too much time has passed, or through receiving three DUPACKs. These indicators are only implicit indications that a packet has been lost, and as such, they have an implicitly delayed notification capability. We use time to indicate that a loss has occurred (either waiting out the whole RTO or waiting for three DUPACKs). This delay is in itself a problem because it results in the sender reacting later than theoretically possible. Reacting later causes the sender to continue at a higher rate than wanted for a longer time, thus adding to the congestion problem at the bottleneck, potentially causing more packet drops and further delays.

Indirect detection is time consuming, and it is not a 100% exact science. A timeout makes us draw the conclusion that a packet is lost (which may not be the case), and from that we infer that the problem is congestion (which it need not be).

Changing the main principle for congestion detection from loss to some more explicit or direct sign of congestion paves the way for detecting congestion before the congestion is at its worst. It also makes the algorithms less sensitive to the occasional packet loss due to reasons other than congestion. One important addition to this is active queue management, and that will be the focus of the next chapter.

Chapter 5

Active Queue Management

Buffers are necessary to allow for fluctuations in flow-rates. Sudden bursts should not cause massive packet loss, but simply use the available buffer space. But as we have seen, the very nature of loss based congestion detection leads to the buffers gradually filling up if the flows live long enough, no matter how big the buffers are. Simply increasing the buffer size to avoid packet loss is a poor solution potentially as the very nature of classical congestion control is to add pressure to the system until the buffers fill up. Increasing buffer sizes may lead to even more latency without avoiding the potential packet loss anyway.[1]

5.1 Queue length is not the whole story

When we say that buffers are necessary to deal with fluctuations or bursts, this is not just bursts or fluctuations caused by e.g. a sender producing slow start bursts. Buffers are needed in all cases where a node has a higher ingress rate than the egress rate. As Jacobson described it; we often need to connect a fire hose to a soda straw, and the adapter for such plumbing is called a queue.[39]

According to Jacobson et al., queue length is not a good measurement for congestion because, at steady state, it tells nothing about the rate. The queue length resonates with the window size. If the window size is bigger than the BDP, this difference is present in a persistent queue which may not necessarily be detected by the queue manager or the sender as congestion. But it does result in unnecessary latency.

Some of these observations led to the idea of Active Queue Management (AQM). Simply put, AQM is the idea that the intermediate nodes actively monitor the buffer occupancy and take action before the buffers fill up completely. Three possible ways of acting are listed here and will be examined in the following sections:

- Random Early Detection (RED) (1993)

[1] Excess buffering to deal with increased packet loss is often called *bufferbloat*

- Explicit Congestion Notification (ECN) (2001)
- Controlled Delay (CoDel) (2012)

5.2 Random Early Detection (RED)

One of the first versions of AQM as we know it today was called Random Early Detection, and it was named this way because of the way it functioned: Select packets to get dropped randomly, and do it early. Early as in before the buffers are actually full, and random as in the queue manager selecting the connection randomly. The probability of a connection getting picked is simply ”proportional to that connection's share of the throughput through the gateway.”[19] [2]

When we say that a connection would get notified, this could happen in one of two ways; either explicitly with markings (see section 5.4), or implicitly by dropping packets like it would normally happen in full bottlenecks; the difference now being that the packets would get dropped before the buffers were actually full.

By doing early detection, the idea is that one may keep the latency low while still being able to handle bursts. By making flows back off earlier, the congestion is reduced before it reaches its maximum (as mentioned earlier).

Doing the marking (or dropping) with a random (probability) calculation avoids a separate problem with classical tail drops. In classical queue management, packets are simply dropped at the tail of the queue which is natural because the queue is full and the arriving packets must be dropped. This may cause what is called a synchronized drop with packets from several connections getting dropped, causing several connections to back off. This leads to the total throughput dropping and poor bandwidth utilization. In RED, the packets are marked based on a random selection. The randomization results in the connections with the highest throughput to be most likely to get marked, which in sum causes the connections affecting the queue the most to be most likely to back off. This reduces the chance of small bursts getting punished unfairly, and increases overall throughput/bandwidth utilization because it makes flows back of at different times.

5.2.1 Limitations

According to Feng et al., ”The inherent problem with these queue management algorithms is that they use queue lengths as the indicator of the severity of congestion.”[16] And ”while the presence of persistent

[2] Because of the early detection, this type of congestion control is often called *congestion avoidance*. We try to avoid congestion from happening in the first place as opposed to congestion control trying to react to and get control of congestion that has already occurred.

queue indicates congestion, its length gives very little information as to the severity of the congestion."

Because packets arrive queues in bursts and hence have an uneven distribution, the queue length is not an expression of the severity of the congestion. One flow filling the queue by sending one large burst is less severe than several flows sending at a too high rate, but this is not easily detected by the queue manager. Using only queue length makes it difficult for the queue manager to determine what the different parameters should be for optimal throughput and latency. The thresholds need to be set well, and the buffers should be deep enough. It has proven to be a complex matter to adjust the parameters optimally; "little guidance was available to set its configuration parameters and it functioned poorly in a number of cases."[32] This is a problem for many AQMs, not only RED. The can work great, but are difficult to optimize.

This observation has led to several alternative ways to do AQM like BLUE introduced by Feng et al., and CoDel introduced by Jacobson which we will discuss further in the next section.

5.3 Controlled Delay (CoDel)

CoDel came about based on observations made by Jacobson in 2006 [39] concerning bufferbloat and queue management. He stated that the traditional perception of bufferbloat used in e.g. RED was a misconception of what bufferbloat actually is. In traditional RED, bufferbloat is detected by calculating an average queue length, but Jacobson demonstrated that this is highly affected by the innate bursty behavior of TCP transmissions. The basic idea described with the plumbing adapter is that we need buffers to manage the transitions from links with high bandwidth to links with lower bandwidth. As previously mentioned this enables us to handle bursts without having to drop lots of packets. Thus, buffers are necessary to deal with bursts hitting links with lower bandwidth, and these bursts are part of several of the algorithms described earlier in this thesis.

Jacobson argued that bursts increase queue lengths (and should do that), and this triggers classical RED to mark or drop packets. But the average queue length increasing for a short amount of time due to a transient burst is not something one should react to; it is simply the intended behavior of a well-functioning buffer coping with burst. The problematic queue length one should actually be worried about is the persistent queue length. That is, a queue filled to some degree for a sustained period of time.

According to Jacobson, a sustained queue is an expression of one (or more) senders having more packets in transit than the bottleneck is capable of handling without draining the queue. The average queue length need not be very long, but it does not go away, and hence is unnecessary. If the sender has a window larger than the BDP, it will result in a persistent queue at the bottleneck. This persistent queue

may be below the threshold for RED to react, and therefore the window will just keep being too big, causing unnecessary latency and reduced capability to deal with incoming bursts. A transient queue may come about because a burst arrives at a bottleneck, but as long as the window is smaller than the BDP, it will be drained and disappear after one RTT.

Because of this, Jacobson and Nichols came up with the concept of *good* and *bad queues* where good queues are transient queues that come and go due to burst handling, and bad queues are persistent queues caused by too many packets being in the pipeline at any one time. Bad queues are what one should properly call bufferbloat (unnecessary queue occupancy), and good queues are just a necessary aspect of TCP flows having variations in rates from one time or place to another.

Figure 5.1 shows how a persistent bad queue comes about due to a constantly too big sender-window, while figure 5.2 shows how a good queue builds up and drains constantly.

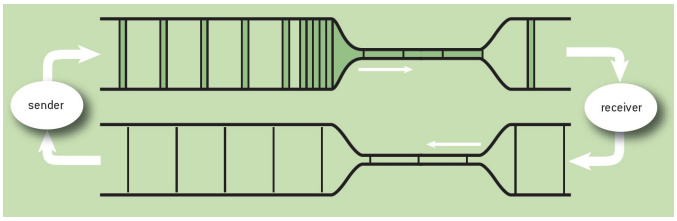

Figure 5.1: TCP connection with persistent queue after one RTT [32]

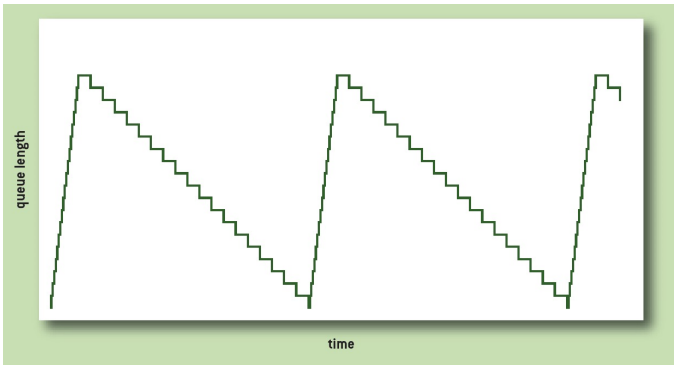

Figure 5.2: A good queue handling bursts [32]

To achieve an AQM where good queues are allowed and bad queues are detected and handled, Jacobson and Nichols created CoDel. CoDel does not measure queue length as an amount of bytes in queue, but rather measures the sojourn time of each packet. The threshold for queue length is set as a maximum sojourn time of a packet. Measuring sojourn time rather than number of bytes is an easier parameter to put in use. We avoid the need to to set different thresholds for different bandwidth links. The buffer for a link with a high bandwidth needs to be bigger (as previously discussed) than the buffer for a small bandwidth link, but the sojourn time may be the same. Further

more, calculating sojourn time is easily done by simply stamping each packet upon arrival. Not all packet sojourn times are compared to the threshold, only the minimum sojourn time of all packets passing through in a predefined interval of time is compared to the threshold. By doing this, the work done by the queue manager is simplified, and it allows for variations in sojourn time. According to the authors; ”Use of the actual delay experienced by each packet is independent of link rate, gives superior performance to use of buffer size, and is directly related to the user-visible performance.”[32]

If the minimum sojourn time is larger than the threshold when the last packet in the interval is processed, the packet gets dropped[3], and the interval length is reduced to increase the drop frequency until the minimum delay is within bounds. If the number of bytes in queue is less than the link MTU, no action is taken. If the minimum is within bounds, the interval length is reset.

The optimal thresholds and interval lengths are defined to be 5 ms for minimum and 100 ms for interval length: ”Below a target of 5 ms, utilization suffers for some conditions and traffic loads; above 5 ms there is very little or no improvement in utilization... ...A setting of 100 ms works well across a range of RTTs from 10 ms to 1 second (excellent performance is achieved in the range from 10 to 300 ms).” [32] But later works argue that minimum target ”should be tuned to be at least the transmission time of a single MTU-sized packet at the prevalent egress link speed”.[23]

Optimizing CoDel is simpler than RED and other classical AQMs, making it easier to implement it in a wide range of networks and hardware At the same time it shows very good performance compared to them.

5.3.1 FQ CoDel

Flow Queue CoDel is a variant of CoDel which combines CoDel with a fair queuing scheduler. Each flow is given its own separate queue, and each queue is monitored by CoDel AQM. The queues are picked with a variant of Round Robin[4] based on byte counting. The byte counting ensures that all flows get the same amount of bytes through, regardless of packet size.

New flows get a lot more leeway to start with in CoDel, and Flow Queue CoDel takes longer ”to converge towards an ideal drop rate for a given new flow but does so within fewer delivered packets from that flow.”[23] It is also has a more accurate drop from large flows, and it reacts faster to changes in a link. It is proven to be very effective,

[3]Dropped may be actual dropping or explicit notification (ECN) if the ECN bool is set

[4]Deficit Round Robin: ”the only difference from traditional round-robin is that if a queue was not able to send a packet in the previous round because its packet size was too large, the remainder from the previous quantum is added to the quantum for the next round. Thus deficits are kept track off, queues that were shortchanged in a round are compensated in the next round.”[37]

but also has some issues with specific conditions like the need for other fairness-criteria, or when combined with congestion control algorithms that need more clear delay-indications than what Flow Queue codel provides to detect congestion.

5.4 Explicit Congestion Notification

The queue manager may inform the sender in two ways; either implicitly by dropping packets as previously explained, or explicitly by passing information to the sender (via the receiver). The latter is done using ECN.

In ECN, once the AQM decides that a queue has built up enough and that the senders should reduce their sending rates, packets passing through will get marked by setting the ECN-bits, notifying the receiver. The receiver then passes that information to the sender through the ACKs. This explicit notification has its advantage in enabling the sender to respond more quickly because no DUPACKs or timeouts are needed. On top of that, the actual packet is not dropped, and that reduces the amount of retransmits. Considering the fact that the sender gets immediate notification, it does not need to wait for the triple DUPACKs, and it does not need to retransmit any packets. With such advantages one would believe it should be in everyone's interest to employ this everywhere, but it has not been used very much in the internet primarily due to two reasons mentioned in[26];

1. It requires changes to all intermediate nodes.
2. Early detection often starves when competing with more aggressive algorithms using loss based detection.

Agreeing on the use of ECN between sender and receiver is quite simple, but in order to ensure that the sender gets informed that congestion is building up, the bottleneck needs to employ ECN. This is not that easy to put together in the Internet. But of course, if the advantages of using it are as good at they seem to be, it can seem odd that it is not more widespread. The reason for this lies mostly in point 2. Because ECN is early and makes the sender respond quickly, it tends to back off before other more greedy, loss based schemes. This may result in unfairness or risk of starvation.

ECN has been put to good use in controlled environments like data centers. Once deployed in a controlled environment, it provides advantages to performance that are difficult to achieve with conventional loss based congestion detection because it may react preventive with less latency due to more shallow buffer needs.

5.4.1 Data Center TCP

One algorithm relying on ECN is Data Center TCP (DCTCP) which manages to combine low latency with high throughput. It came

about in 2010 because Alizadeh et al. observed that loss based congestion control struggled with too high latency for short messages and queries/aggregate in data centers where these types of flows are greatly mixed with ”long-lived, greedy TCP flows”.[3]

Ever since [40], the general rule of thumb has been that the buffer sizes of an intermediate node (e.g. a router) should be equal to the BDP. The reason for this is simply that the ”worst case data burst remains approximately equal to the”[40] BDP (assuming that the flows stay below the saturation point.

Some variations to this rule of thumb have been made[6]:

- dividing the BDP by the square root of the number of flows (assuming that the flows are asynchronous and independent of each other). This has proven to work well and is often used today.
- even smaller buffers (assuming that bursts are non-existent, which excludes a lot of networks)

Either way, it becomes apparent that the buffers will get deeper as the bandwidth increases. In data centers the RTT may be small, but the bandwidth may still get big for certain flows, and this poses a problem because of the variety of flow types present in a data center. As Alizadeh et al. put it, the greedy, long-lived flows running in the background ”will cause the length of the bottleneck queue to grow until packets are dropped, resulting in the familiar saw-tooth pattern”. This will happen over and over with congestion avoidance building it up and backing off when overflow happens. In a data center, this causes problems because short-lived, latency sensitive flows mix with the long-lived flows, and these flows experience an increase in latency because of the long-lived flows filling up the buffers.[3]

One way to deal with this would be to reduce the buffer sizes, but if the buffers were reduced, the heavy background flows would suffer because smaller buffers makes for reduced burst-tolerance resulting in an increased risk of packet loss and an unwanted reduction in throughput. Using ordinary TCP, the solutions to these challenges could be summed up in two separate options that both seem insufficient:

1. Keep the buffers large: Accommodate larger flows and provide burst tolerance, but increase the latency.
2. Reduce the buffer size: Accommodate the latency-sensitive flows (queries and short messages), but impact the throughput of the long-lived flows and increase the odds of overflowing when faced with bursts.

The solution that Alizadeh et al. created in DCTCP seeks to achieve the positives of both these options by using (or actually misusing[5]) ECN to provide early detection. Rather than dropping packets, the packets

[5]According to the original RFC (3168)[35], a ECN marked packet should cause ”the

are marked to inform the sender that action needs to be taken. The threshold for marking packets is set well below the buffer size, thus providing very early detection, resulting in a very low queue occupancy. By doing this, the buffer is more resilient to bursts arriving because there is a lot of empty buffer space to use. At the same time the average latency is kept low because the average occupancy is low. The larger bursts may surpass the threshold, thereby resulting in packets being marked, but the risk of packet loss is low because we are operating well below the buffer size limit. Once a burst increases the occupancy, the sending rates should be reduced, and the queue should get drained, thus re-establishing the low occupancy and low latency.

Seeing as the thresholds are set quite low relative to the buffer size, this results in a fair amount of markings. If marks would cause reactions equal to those of packet loss (reducing `cwnd` to half) this could cause major underflow. Therefore, to ensure good link usage and keep the thresholds low for low latency and high burst tolerance, the reaction to marks is calculated using the following function:

$$cwnd \leftarrow cwnd * (1 - \alpha/2)$$

where α is calculated using the number of marked packets per RTT like this:

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

"where F is the *fraction* of packets that were marked in the last window of data, and $0 < g < 1$ is the weight given to new samples against the past in the estimation of α "[3]. If congestion is small, only a small fraction of the packets are marked and `cwnd` is just slightly reduced, but if congestion is high, α is equal to 1, and `cwnd` is halved just like in ordinary TCP.

On paper, DCTCP seems to mitigate a lot of the problems raised by the special demands in data centers. It provides low latency and higher burst tolerance by keeping the queue occupancy low, and it is very resilient to packet loss. The main challenge with DCTCP is the difficulties related to implementing it outside of fully controlled environments and that it tends to starve against regular TCP.[6] Even though this might be the case, it is used in some form in a lot of data-centers today, and a lot of work is put into getting some version of DCTCP to the internet.[7]

transport layer to respond, in terms of congestion control, as it would to a packet drop." As will become clear, the idea of DCTCP is not to react to markings exactly like one would to a packet drop, but to react more mildly. The behavior defined in RFC 3168 has since been relaxed in RFC 8311

[6] Briscoe et al. in [26]; "no way has yet been found for DCTCP traffic to coexist with conventional TCP without being starved"

[7] See e.g. the L4S standard[41] or [26] on configuring a new AQM scheme to make DCTCP deployable in the internet

DCTCP difficulties

Huge amounts of bursts are one of the challenges DCTCP was constructed to handle, but batching of packets in Large Segment Offload (LSO) creates larger bursts than bursts caused by the algorithm itself. Tests show that the this results in very frequent queue-length oscillations, and it may cause buffer underflow and reduced throughput when LSO is used.[36]

DCTCP has an extensive use of markings, and with large segments bursting into the network, we have a high risk of single flows getting punished harder than others because a whole train of packets arrive the queue from the same flow. In DCTCP such a flow risks getting a lot of markings and could consequentially back off a lot more than actually needed.

Batching caused by offloading is problematic to several algorithms and it will be thoroughly examined in chapters 7 and 9, but the extensive marking-scheme of DCTCP makes it extra sensitive to such cases as the one described above.

Chapter 6

Measurement based congestion detection

AQM has the potential advantage of reducing latency and packet loss by reacting early and avoiding the big congestion buildups. When using ECN it is also possible to avoid the negative effect of dropping packets. This reduces the number of DUPACKs and RTOs, thus improving the performance of the flows.

The major challenge with AQM is that it requires active monitoring and configuration of the intermediate nodes. In the case of ECN, all entities need to be configured. The advantage of not using AQM is the plug-and-play effect. All configurations are done sender-side or may perhaps be negotiated with the receiver during the initial handshake. Avoiding computations and monitoring in the intermediate nodes also reduces overhead in the nodes.

AQM has some important advantages. It tackles the bufferbloat problem, and it enables us to use smaller buffer than would otherwise be possible without risking massive packet loss. The question is if it is possible to have some of these advantages in place without involving any other entities than the sender. The following sections investigate some of the proposed solutions. One common aspect to them all is that they try to react early by using measurements rather than reacting to packet loss.

6.1 TCP Vegas

TCP Vegas (Vegas) is not a modern algorithm, but it is worth mentioning here because it was one of the first TCP algorithms employing a delay based congestion detection. As early as 1994[14], Vegas introduced the notion of using delay to detect congestion before packets were lost.

The idea of Vegas is simply to keep track of the RTTs and measure them against a base RTT (an estimate for the smallest possible RTT of a path). Once the observed RTTs get too high compared to the base RTT,

congestion is assumed to be building up. The results of the algorithm were included in the abstract of the publication as achieving ”between 40 and 70% better throughput, with one-fifth to one-half the losses, as compared to the implementation of TCP in the Reno distribution of BSD Unix”[14].

With these positive figures it seems odd that Vegas or some other algorithm with the same logic has not prevailed as the dominant congestion control algorithm in most networks today. The reason is, simply put, that Vegas is not greedy enough to get implemented into a free market like the internet. When competing with e.g. Reno, Reno seems to get up to 50 percent more bandwidth than Vegas. This is a major issue for implementing and spreading the usage of an algorithm. If it is starved it does not help that it is ”kinder” to the bottleneck link. It will simply get pushed aside.

In addition to this issue, some other issues were revealed in the years after its creation. It proved to be very conservative during slow start, using an additive increase for a larger portion of the time, thus using a lot longer time to get up to speed. It also had an issue with handling route change. Routes may change in the network at any time, and for a protocol dependent on a base RTT, this posed a problem. As will be discussed in the following section, Bottleneck Bandwidth and Round-trip propagation time (BBR) also uses a base RTT but solves the issue by continuously re-estimating it.

All in all, Vegas was a very promising attempt at what some of our most modern algorithms seem to pick up on. The version presented by Brakmo et al. in 1994 had some shortcomings, but several modern algorithms emerging now use delay rather than loss to detect congestion. BBR is one of them, and it shows big promise to some of the challenges posed by both loss based congestion detection, and the shortcomings of the Vegas algorithm.

Interestingly for us, it uses pacing actively to achieve its results.

6.2 Bottleneck Bandwidth and Round-trip propagation time

BBR is an algorithm presented by Van Jacobson et al. as recent as 2016 as a direct response to the challenges posed by having loss based congestion detection:

> ”When bottleneck buffers are large, loss based congestion control keeps them full, causing bufferbloat. When bottleneck buffers are small, loss based congestion control misinterprets loss as a signal of congestion, leading to low throughput.”[15]

They argue that traditional, loss based congestion control does not operate in the optimal area of bandwidth-usage. Loss based congestion

control continuously pushes more data into the network until packets are lost. Once a packet is lost, the flows slow down before speeding back up until packets are lost again. Doing this results in the bottleneck queues constantly being filled up, and full queues result in both high latency and frequent packet loss. Furthermore, they argue that loss is not a good metric alone for detecting congestion because you can very well get packet loss before there is sustained congestion. Loss based congestion control is sensitive to this, and as a result we get poor throughput.

In short; if loss occurs before congestion, we get low throughput - and if loss occurs after congestion, we end up with bufferbloat with constantly full buffers because the flows will keep making the buffers overflow (they have no other way of knowing when to slow down).

BBR proposes a new approach to congestion control by attempting to measure the actual BtlBw. A flow at any time only has one bottleneck[1] with a specific BtlBw, and calculating that one variable is sufficient. If we are able to measure the actual BtlBw, we do not need to overshoot the capacity of the bottleneck, and we do not need to slow down more than necessary. In theory, we may get a more optimal bandwidth utilization with all the positives that implies.

The goal of BBR is to meet two conditions at the same time; sending packets at a rate equal to the BtlBw, and having a `cwnd` equal to the BDP. To achieve this, it is necessary to firstly pace out the transmission to send at an appropriate rate, and secondly to measure the BtlBw and optimum RTT. By not only adjusting the `cwnd`, but also adjusting the sending rate, they argue, it is possible to reduce the queue lengths because the bursts are removed. If we calculate the correct BDP, but do not pace the flow, we end up with the bottleneck buffer constantly filling up and draining due to the bursts. It may also lead to packet loss because the buffers are too small.

Pacing out the packets is easy as long as we know the `cwnd` and RTT. The main challenge is finding the actual BtlBw. BBR uses two separate probing techniques to find the minimal RTT and the BtlBw respectively:

1. ProbeRTT
2. ProbeBW

Probing for RTT is simply to try to determine the Round-Trip propagation time (RTprop). RTprop is the minimal amount of time a packet needs to traverse a path if there is no congestion (”noise introduced by the queues”[15]). The RTprop is physically determined by the length of the path and medium the signal has to traverse. In the ProbeRTT phase, the sending rate is reduced significantly so as to drain the queues over a period of at least 200ms and get an RTT as close to the RTprop as possible. This phase is repeated every 10 seconds to be able to detect changes in the RTprop (i.e. changes in the physical path).

[1] There must necessarily be one slowest link or hop in the path

In the ProbeBW phase, 8 RTprops are used to test an increased sending rate. The first round (RTprop) sends at a rate of 5/4 of the old sending rate. The next round reduces to 3/4 to ease the stress on the queues, and the last 6 rounds send with a sending rate equal to the old sending rate. By observing the ACKs, it is determined whether the increased sending rate of the first round was successful (i.e. did not increase the observed delay, indicating an increase in queue occupancy).

Every ACK contributes with an RTT, and combined with the sender monitoring the amount of data inflight, we get a delivery rate. ”Average delivery rate between send and ack is the ratio of data delivered to time elapsed”[15]. As long as the amount of data inflight is less than the BDP, the delivery rate should reflect the sending rate. Added delay indicates that the queues are getting filled.

The main idea of BBR is to use RTprop as a limit, and figure out what amount of delay/noise the flow experiences throughout the lifespan of the flow. RTprop is the minimal amount of time a packet needs to traverse a path if there is no congestion (”noise introduced by the queues”[15]). By measuring the delivery-rate found by analyzing the ACKs, the propagation-time and combining them with the amount of data inflight, Jacobson et al. argue it is possible to find the BtlBw as the maximum delivery-rate at that time. TCP measures RTTs all the time, so all information needed to do the calculation is to keep track of the amount of data inflight.

An important point here is that it is not possible to measure bandwidth and RTprop at the same time. To measure the latter, the pipe needs to be as empty as possible, and to to measure the former, we need to fill the pipe as much as possible.

Combining these different phases and constantly alternating between them provides us with the possibility of estimating the BDP quite efficiently and accurately. Once this is combined with pacing, we are able to maximize the `cwnd` without filling the queues.

The pacing in BBR is an interesting focus shift from earlier algorithms. ”BBR must match the bottleneck rate, which means pacing is integral to the design and fundamental to operation - pacing_rate is BBR’s primary control parameter”[15].

As will become apparent in chapter 8, pacing seems like a very promising way to maximize bandwidth utilization whilst reducing buffer sizes and latency. Jacobson created both RED and CoDel to tackle certain side-effects of packet bursts. With BBR it seems that we are shifting our focus to the sender and the way the sender is transmitting to remove some of the issues further down the path.

BBR is deployed in the Google B4 backbone and YouTube for testing, and it shows good promise. Vegas was also founded on estimating delay variations, but had trouble competing with loss based algorithms. Experiments show that BBR does not lose against loss based algorithms. In fact it is a lot more aggressive during its start-up phase than e.g. CUBIC. In addition to a more aggressive

probing for bandwidth during its slow start phase, it provides a faster convergence to equilibrium and fairness because it quickly calculates a more accurate BtlBw and adapts to BtlBw-changes[11].

6.2.1 BBR difficulties

The fast and aggressive convergence also has some negative effects because it results in the BBR flow using more than its fair share when it competes against other algorithms (e.g. CUBIC). Tests also indicate that BBR beats CUBIC with regards to throughput if the buffers are shallow, but that the results are opposite when the buffers are deep[11]. This may be caused by the loss based flows taking advantage of the empty space of deep buffers (left empty by BBR) to send more data. Another observation is that BBR appears to have more throughput oscillations due to its drainage phases. It is unclear as to whether this affects the overall throughput a lot, but it calls for further testing. There are also indications that throughput is better when we have just a single CUBIC flow than a single BBR flow.[11]

All in all there are several advantages to BBR if the buffers are shallow, and if loss is a major negative. As to the negative sides, it all depends on the competing flows, and what type of environment it is used in. Development is ongoing and BBRv2 is being rolled out with improvements to many detected weaknesses like e.g. coexistence with Reno and CUBIC with better fairness[44]. It also takes advantage of ECN signals where they can be found with results similar to DCTCP, and it may react to loss if needed.

Chapter 7

Bursts

Traditional TCP is window based rather than rate based. Because of this, bursts seem to be not only an inevitable, but a natural part of TCP flow- and congestion control. The previous chapter showed that a lot of effort has gone into dealing with bursts. Dealing with bursts may be done proactively to avoid bursts being created in the first place, or reactively by doing proper buffer sizing or AQM to make room for bursts hitting the networks.

The goal of this thesis is to shed light on and provide one way to reduce the formation of bursts, but to further argue that such a solution is even needed, it is important to clarify what bursts are, what events cause them, and what types of mitigations actually exist specifically for bursts.

There are many causes of bursts, and different states may also effect which possible effects they may have. To fully understand the effects of a burst or what type of mitigation is either possible or preferable when presented with it, it is important to list the most prominent ones before listing the known solutions to either reducing the bursts themselves or the effect of them.

7.1 Definition

Bursts are what we will call a collection of packets sent back-to-back as close together as possible.[1] It seems reasonable to define a burst as a sequence of at least 4 segments like it is done in ”On the Impact of Bursting on TCP Performance”[10] .[2] Where this line is drawn does not have an impact on the effects of bursts, but rather whether they may avoided at all.

We may split bursts into two categories:[3]

- **Micro-bursts**: Bursts sent as a response to a single event like an ACK resulting in a window slide or some other response.

[1] As close to each other as the link allows us to

[2] 2-3 segments at line rate is an intricate part of TCP algorithms

[3] This division is mentioned in several articles concerning bursts, e.g. [10] and [5]

- **Macro-bursts** The bursty behavior over time of a flow due to TCP behavior like slow start, ACK compression etc.

Bursts occur in both UDP and TCP, but we will only concern ourselves with TCP here.[4] As shown in chapter 2, a typical TCP flow has a lot of (mostly unintended) bursts, and there is a wide range of causes for them. In the following, the most prominent ones will be presented. We will not categorize the bursts into micro- or macro-bursts. The categorization does not add to the understanding of their impact or what type of mitigation we should consider for them, but it seems fair to state that larger bursts are more interesting to our implementation and analysis, as will become more clear later on.

7.2 Causes

7.2.1 Slow start

Most slow start implementations follow the same principal idea as the one first outlined in Tahoe. We do an exponential increase until packet loss is detected, and then react to the loss. The different parameters; initial window, reaction to loss and so on may vary, but the main idea is the same, and the formation of bursts takes place the same way in either case.

New packets are sent out when ACKs arrive. If there is little congestion,[5] we may expect two packets sent back-to-back to be ACKed back-to-back.[6] When those ACKs arrive at the sender, they will generate four new packets being sent back-to-back.[7] The next RTT, this is increased to eight, and this continues until we reach `ssthresh` or detect packet loss in some way.

The original (and subsequent variants) of slow start does not specify when or how packets should be sent. The idea is that new packets are introduced to the network as soon as possible. The ACKs may get stretched out due to bottleneck limitations, but each time around the density is doubled due to the increase in `cwnd`. Best case, we spread out the burst due to the ACKs being paced out by the bottleneck rate, but worst case, we may have delayed ACKs or ACK compression causing large bursts the size of `cwnd`.

7.2.2 ACK compression

ACK compression is a phenomenon discovered by Zhang et al. [43] which occurs when a cluster of ACK packets encounter a non-empty

[4] If we expand our implementation to the more general case of LSO, it will most likely facilitate handling of UDP bursts caused by UDP Segmentation Offloading

[5] Even with some competing traffic and congestion, the packets may end up back-to-back in the queue

[6] At least at bottleneck rate

[7] Two due to the ACKs leaving the network, and two due to the slow start phase causing `cwnd` increase

queue;

> ”their spacing in time upon leaving the queue is no longer the transmission time of a data packet but rather becomes the transmission time of an ACK packet. Since ACK packets are typically much smaller than data packets (in our simulations ACK packets are 1/10 the size of a data packet), this causes the ACK packets to arrive at the source much more closely spaced. The ACK packets are thus no longer reliable indicators of departures of data packets from the queue.”[43]

7.2.3 TCP Segmentation Offloading

TSO will be explained in more detail in chapter 9, but in short TSO results in burst in the network because larger chunks of data are sent from the kernel to the NIC, and those larger chunks may be sent onto the wire as a series of packets of size MTU sent back-to-back.

7.2.4 Fast Retransmit

If we use Fast Retransmit, we risk a hole in the window being filled if a retransmitted packet is successfully received. This filling of the hole may result in the whole window being moved, and all packets in the new window being sent back-to-back. The burst will often be sent at least within half an RTT[1]. SACKs contribute to this effect by efficiently acknowledging selected packets that may fill wholes in the window.

7.2.5 Delayed ACKs, Stretch ACKs and ACK thinning

The bursting effect is present in much the same way with delayed ACKs as with ordinary ACKs. The only difference is the number of bursts and their sizes. A delayed ACK simply worsens the effect, acknowledging several packets at the same time, resulting in the `cwnd` being increased several steps at the same time, and a larger burst being sent. The same may naturally be said about stretch ACKs and ACK thinning.

7.2.6 Multiplexing

Bursts from a single flow may hit a bottleneck back-to-back with bursts from other flows, causing the cumulative effect of a bigger burst hitting the bottleneck.

7.2.7 Unused `cwnd` increases

This variant is mentioned in [25] as seen e.g. in *scp*. In this example, a flow may send several control messages before sending a larger message. Every control message increases the `cwnd`, and when the final, large message is sent, it creates a burst.

This burst cause has similarities with any type of transmission starting after some idle time. Usually the idle time is a result of the

application waiting some period of time. Once the application tries to transmit again, this may result in a burst because as much as a whole window may get sent. To tackle this there is usually some idle time threshold where slow start is restarted if the idle time is out.

7.3 Effects

In general one may say that bursts have a negative effect on networks because they create oscillations in queue length, varying the load on the network from one time to another. Such effects are negative due to the unpredictability they create, but also because they have several secondary effects on the sender, receiver or other flows in the network. We will mention a couple of them in the following.

7.3.1 Packet loss

According to [10] the probability of losing a packet from a burst increases depending on how large the bursts are. At some point the ratio between buffer size and burst size is so strained that we get a massive increase in packet loss probability. The smaller the buffer size, the smaller the burst needs to be to have a massively negative impact on the probability of loss.

Packet loss is the worst consequence we may have because it may result in retransmit or a full RTO in which case the `cwnd` is severely reduced in classical TCP.

7.3.2 Buffer underflow and throughput loss

Shan et al. show that bursts caused by batching (interrupt coalescing/segmentation offloading) in DCTCP seems to cause queue oscillations that may lead to buffer underflow (14,3% throughput loss [36]) and 3 times larger oscillations compared to running without batching. Relative to the BDP in DCTCP we go from O($\sqrt{BDP}$) oscillations to O(BDP). To avoid throughput loss in DCTCP we need to adjust ECN threshold 61,6% higher (thereby increasing latency/queue length).

Even though the studies performed by Shan et al. are specific to DCTCP, the results are applicable to TCP in general concerning throughput. If we have huge queue oscillations, our throughput is likely to be reduced because it may result in packet loss and delay. Packet loss and delay cause flows to back off because they assume it is caused by congestion, and when the flows back up, they need to ramp up towards equilibrium again, and in the meantime throughput is reduced.

7.3.3 Slow Start effects

While in the slow start phase, a flow is sensitive to packet loss to get up to speed. If all packets in a `cwnd` are sent in one burst, and that burst encounters a queue which is almost full, we may experience packet

loss without being close to the potential BtlBw. The loss may cause overshoot and make the flow back off.

The problem is that we are aiming to make the flow run at a rate as close to the rate of the bottleneck as possible, but we try to do it without tuning into the rate. The classical TCP algorithms focus only on `cwnd` size, not actual rate. This makes us send big bursts of packets, and packet drops are affected not only by the BtlBw, but also the buffer capacity. Variations in buffer capacity directly impacts the experienced BtlBw because of both latency and packet loss. This limits us not to the forwarding rate of the bottleneck, but the ingress buffer capacity of the bottleneck. In addition to that, we know that the buffer capacity is expected to vary over time because of varying bandwidths (as mentioned in section 5.3). If we send out windows as fast as possible, worrying not about the forwarding rate, but only the size of bursts handled, we clearly see that we get bottlenecks with varying window-capacity over time as the buffer occupancy varies.

Ideally, we want our window to be as close to the BDP as possible, but as the discussions in 5.3 and 6.2 have showed us, we must also consider the buffer size of the bottleneck if we are to tackle the bursts innate in classical slow start.

The effect of a burst depends on several factors; the burst size, rate, bottleneck buffer size, and bottleneck buffer occupancy. With regards to packet loss, the worse the ratio between free queue space and burst size is, the worse the effect of the burst is. Regardless of packet loss, the effect of bursts is that we get an increase in queue length if the intra-gap of the packets in the bursts is lower than the forwarding rate of the bottleneck.

7.3.4 ACK compression effects

When ACK compression occurs, the spacing between the ACKs is reduced, and they arrive a lot closer than they departed from their origin. Some traffic caused them to get queued up together much like when there is a safety car in Formula 1. This may in turn result in a burst of data packets being sent into the network, filling the free space in the pipe communicated by the ACKs. The ACK clock is also disturbed. This poses a problem for several different algorithms. As mentioned in section 4.3.3, algorithms that use delay or intra-packet gap (like Hybrid Slow Start and Paced Chirping) to measure the level of congestion, ACK compression is problematic. ACK compression removes the value of measuring intra-gaps because the spacing between ACKs is not the same as the space created by the state and forwarding rate of the bottleneck.

Algorithms that are not dependent on the spacing between packets to detect congestion are still affected by ACK compression because it causes potentially larger bursts than would otherwise be the case. The compression causes the sender to send even more packets back to back than would otherwise be the case. Zhang et al. observed that two-

way traffic caused sharp increases in queue lengths because the ACK clock got disturbed. When the ACKs are not ”separated in time by at least the transmission time of a data packet”, [43] we get oscillations in the rate at which new packets are sent. This extra bursty behavior causes queue length increases which in turn causes delay and possible overflow. Because the packets are introduced without the spacing intended, we may get a premature congestion or even congestion that could otherwise have been avoided had the packets been spaced out as originally intended.

7.4 Mitigations

Mitigating these effects may be done using mainly one of these approaches or a combination of them:

1. Increasing buffer sizes
2. Using explicit signaling
3. Removing or reducing bursts by either limiting their potential max size or by using pacing.

7.4.1 Increasing buffer sizes

Increasing the buffer sizes may work to some extent, but as mentioned in section 5.3, this only masks the problem because the loss based algorithms keep increasing their `cwnd` until the buffers are full either way.[8] Increasing the buffer sizes may give more room for bursts, but queue management seems just as important.

7.4.2 Explicit signaling

Using explicit signaling is one way to do AQM and react before the packets actually need to get dropped. But if e.g. ECN is used the way it was intended to, the packet should be considered lost and resent once it is marked. This does not solve the issue of e.g. exiting slow start prematurely, but the lower queue occupancy at least betters the performance with regards to latency.

(Mis)using ECN needs some proper calibration, usually tuned specifically for a certain environment. No matter what solution or calibration is picked, tuning AQM is not trivial, and slow start is generally sensitive to shallow buffers or an aggressive AQM, but it is a valid solution to the problem.

[8] That is, if the buffers are used over time by either one or more flows

7.4.3 Alternative algorithms

Hybrid Slow Start, BBR and Paced Chirping use measurements to detect when we are approaching the saturation point during slow start, but they have some important differences. Hybrid Slow Start actually relies on the bursty behavior of slow start to measure inter-packet delay. The first packets of the burst are assumed to be sent back-to-back (as a burst), and their corresponding ACKs are measured. Paced Chirping uses active measurement through chirps to measures the available capacity. BBR also uses measurements, but it differs in one important aspect. It introduces pacing as a means to achieve the best possible utilization.

Hybrid Slow Start may provide us with the possibility of exiting slow start before packets actually get dropped, but as long as bursts are used, it makes itself sensitive to buffer sizes and capacity, not only the forwarding rate of the bottleneck. By measuring only the first (8) packets of the burst, the idea is that the size of the burst does not affect the measurement.

7.4.4 Pacing

In sum, the solutions presented above have some limitations. By not removing the bursts, our slow start exit risks being premature. Reducing the burst sizes or increasing the buffer sizes may postpone packet loss caused by bursts, but at some point the buffers will get filled. More so, even if we avoid packets getting lost, we introduce an increase in latency due to buffers filling up. Buffers were introduced to tackle variations in ingress rates, but having buffer capacity does not make it necessary to fill that capacity.

BBR uses rate rather than window size as the measure of bottleneck capacity, and it paces out the packets at that rate. This has the important effect of removing bursts sender-side. By pacing out the bursts, we may reduce the negative impact on the bottleneck queue, thus increasing the potential size of `cwnd` before buffers get filled up.

Zhang et al. introduced the concept of pacing to mitigate the effect of ACK compression. The idea was that the sender may pace out the packets to ease the pressure on the bottleneck, and restore some of the clocking effect. ACK compression may happen during any phase of a TCP algorithm, and using pacing should be considered a possible solution to the problem either way.

Pacing enables the bottleneck to potentially handle a larger amount of packets in one RTT than in the case where all the packets are sent in one burst. There is a significant difference for the bottleneck receiving 100 packets in 2 ms and receiving 100 packets in 100 ms.[9] Pacing out the packets over the RTT shows better promise to avoid filling up buffers prematurely and enable us to approach the actual forwarding rate of

[9] Off course, if the burst is within the buffer capacity of the bottleneck, we can still argue that getting all packets out at once might even be better.

the bottleneck in a controlled manner both with a single flow and with multiple flows competing for capacity.

Pacing may be introduced into several different phases of a TCP connection, slow start being one of them. Pacing is one of the main foci of this thesis, the whole next chapter is dedicated to it. Pacing as an augmentation of traditional slow start will be examined further there, but in short, the advantage of pacing seems to be that we may reduce pressure on bottleneck queues without reducing the `cwnd`, and this in turn reduces the probability of packet loss due to buffers filling up.

That being said, pacing is not error-free. As will be discussed in chapter 8, pacing has some limitations and challenges that need to be addressed if it is to be used in e.g. slow start. If we pace out very small windows, we get a slower slow start, and if we ease our way in with pacing, we risk maximizing our queue overshoot. In addition to this, pacing introduces problems that are not prominent in non-paced implementations; like delayed reaction, synchronized flows increasing pressure on bottlenecks, and some other issues. More on this in chapter 8.

An alternative approach

An alternative way to pace is used in Gallop-Vegas where slow start itself is paced [17]. The idea is to reduce the bursty behavior of slow start by reducing the exponential increase. Gallop-Vegas achieves this by increasing the `cwnd` more rarely (on every other ACK rather than on every ACK). In sum, this results in smaller bursts, but it does not remove them. If we have several ACKs arriving back-to-back, we preserve this block of packets and add on them (though the add-on is smaller than in traditional slow start). The solution tries to reduce the problem without actually removing it. Another problem with this solution is that it reduces the exponential increase of the slow start algorithm itself. We may postpone some negative effect, but at the cost of ramp-up speed.

Alternative ways to reduce the exponential growth of slow start exist. As mentioned in section 4.3.2, HyStart++ introduces LSS to be used when the window size becomes very large. LSS reduces the increase of the `cwnd` to avoid major overshoot. This reduces the potential overshoot at the cost of growth rate. Again, it seems plausible that tackling the bursty behavior of slow start could prolong the time spent in slow start before entering LSS or other similar growth reducing measures because a non-bursty flow may close in on the BDP with less risk of loss than a bursty flow will. This remains to be discussed and tested later on.

7.4.5 TCP Segmentation Offloading

TSO may result in as much as 64 KB being sent in one burst of MTU sized packets. The effect of TSO to a network is much like other

bursts, but as mentioned previously, Shan et al. showed that batching like TSO has a more negative effect on DCTCP than the other burst causes. Because of this, patches have been made to make it possible for the kernel to dynamically size the TSO-packets. In Linux this is called automatic sizing of TSO packets, and it is done per socket approximating the current sending rate and dynamically sizing the TSO packets so that at least one packet is sent every millisecond.

7.4.6 Summary

A lot of work has been put into avoiding the negative effects of bursts. The solutions may be divided into two kinds; either make room for the bursts or try to remove as much of the bursts as possible.

Making room for bursts by increasing the buffer sizes is a solution that has had a lasting legacy, but it has some shortcomings. It comes at a cost of increased latency because the queues are deep and need to get filled before traditional algorithms react to the congestion formed. At its worst, the deep buffers do not solve our initial problem of extensive packet loss because the loss based algorithms will keep filling the buffers no matter how deep they are. In the end we end up with bursts resulting in packet loss because the buffers are full (only now, we have also increased the overall latency of the path).

DCTCP tries to solve the issue by having buffers deep enough to cope with bursts, but avoid having full buffers by (mis)using ECN. But DCTCP still has problems with large bursts created by TSO.

It seems like a change in congestion detection is crucial to avoid the problematic behavior of loss based detection where we either get high latency due to deep buffers, or big losses and low throughput due to shallow buffers.

The other approach is the main focus of this thesis. Solutions like limiting burst sizes sender side, limiting the number of packets sent as a response to an ACK, or reducing the growth rate of the `cwnd` all try to reduce the sizes of bursts. They reduce the burst sizes and negative effects on queues, but they also limit the intended convergence speed of the algorithms. Pacing is also used to reduce or sometimes completely remove bursts, and on the surface of it pacing seems to achieve this without having to limit the algorithms and their convergence speed. There are some challenges, though, and the next chapter will delve into pacing as a concept, discussing both positives and negatives. After a more thorough walk through of pacing, we will move on to the Linux stack and TSO. There is an increasing desire to use pacing both as an innate part of the algorithm logic and as an augmentation to existing algorithms, but pacing and TSO do not interoperate well as of today.

Chapter 8

Pacing

8.1 The origins and definition of pacing

The initial idea of pacing was introduced to TCP by Zhang et al.[43] to tackle some effects of ACK compression. Since its first introduction, pacing has been used and tested with different parts of TCP in various different environments. Different implementations of pacing exist; from pacing with Proportional Rate Reduction (PRR) or rate-halving, to more recent pacing in hardware for end systems or routers.[1] Regardless of how pacing is done, we will define it as transmission where packets are deliberately spaced out in time.

8.2 Advantages of pacing

Since the first paper introduced pacing in 1991, a lot of both development and usage has followed. Pacing seems positive in a lot of different settings, some of which have already been mentioned.

8.2.1 Reducing queueing delay variation

Big oscillations in queue length is negative for the throughput if the buffers can not cope with it. If so it leads to more frequent loss and drop in throughput (low bandwidth utilization). It also may cause an increase in latency.

The unpredictability or variation in queue length can make flows back off more often than if the queues were more predictable. The more unpredictable the queue length is, the more likely flows will either overflow or underflow the queue.[2]

[1] PRR or rate-halving which seems more like spreading out than delaying, but it is a type of delaying/spacing of packets that could otherwise be sent at once after a pause. The whole intention of PRR or rate-halving is to better utilize capacity by removing the large delay followed by a burst imposed by fast recovery. It is done by pacing out small gaps rather than one big gap

[2] Variations in queue length may be interpreted as congestion and cause the flows to back off. The same thing happens if the intermediate nodes employ some sort of

By employing pacing, the queues build up or drain at a more steady pace. This creates less variation for flows trying to estimate the amount of congestion present in the network. Removing bursts may also remove the short term queue buildup entirely (assuming that the `cwnd` is less than the BDP and the paced rate is kept below the forwarding rate of the bottleneck).

8.2.2 Reducing loss rate

As mentioned, employing pacing reduces loss rate because of less queue variation. Less queue variation and a more smooth convergence to bottleneck rate may result in less packets getting dropped because queues fill up more slowly, allowing for more predictive action. The odds of a packet being dropped is less as the rate at which a queue builds up to its limit decreases. A sudden burst of packets filling up a queue may cause extensive loss because the queue went from a state of having available capacity to having no capacity in no time.

While this holds some merit, we also must comment that network environments and flow settings may impact the advantage. If the sending rate is higher than the bottleneck forwarding rate, queues will start to fill up. The bigger the difference, the faster the build. A burst is the outer extreme of this where most of the `cwnd` may end up filling the buffer rather than being forwarded as it hits the bottleneck. The larger the `cwnd` and the higher the sending rate relative to the capability of the bottleneck, the more likely we are to overflow the buffer and experience packet loss.

8.2.3 Worst-flow completion time is reduced

According to Wei et al. employing pacing reduces worst-flow completion time[18][3] Worst-flow completion time is used as ”a measure of fairness among a set of flows that start around the same time and have the same RTT. Simply put, a worst-flow latency is the latency of the slowest flow to finish the transfer”. Their measurements show a clear advantage to the paced flows when it comes to worst-flow completion time. The results are based on tests using various algorithms (Reno, NewReno, SACK, FACK, BIC and FAST). Aggregate throughput is also improved.[4]

8.2.4 More time spent in slow start

Tests show that paced flows (Reno) stay longer in slow start than non-paced flows because non-paced flows overflow buffers once the `cwnd` gets larger than the available buffer size.[1] Once this happens, slow start is

preemptive signalling like ECN. If not, the queue length oscillations may lead to packet loss

[3]Wei et al. call it worst-flow latency, but the term is interchangeable with worst-flow completion time.

[4]The experiments are run on various scenarios with TCP Reno and BIC-TCP using 1Gbps links and a variation of settings.

exited, and congestion avoidance begins. The larger the bandwidth, the larger the impact of this early exit. For paced flows with a rate lower than the forwarding rate of the bottleneck, the bottleneck buffers start to fill only when the pipe is saturated, i.e. when $\texttt{cwnd} = BDP$.[5]

One could argue that this increases the risk of a maximal overshoot in slow start because there is no indication to the bottleneck that we have reached a saturated point and are about to double our window. The effect of this depends on how large the buffers are and what type of AQM is used. But potentially a large portion of the doubled window could get dropped if the doubling fills the buffer. Unless we have way of making the sender back off before the buffers are completely full, this will lead to a massive drop. A way to reduce the consequence is using ECN or by actively measuring sender side (like BBR)).

8.2.5 Allows for smaller buffers

Both Wei et al. and Beheshti et al. [8] argue that pacing is necessary in shallow buffered networks if the link-speed is high. As previously mentioned, smaller buffers easily overflow when link-speed is high if bursts/unpaced flows arrive at the bottleneck. It also complements the comments made by Van Jacobson in [15] that higher link-speed results in deeper buffers because higher link-speed causes smaller buffers to more easily overflow due to rate variations. Remember that this is one of the principle ideas behing DCTCP; to ensure low latency (short queues), we either need to make the buffers smaller, or enforce some AQM to keep the buffer occupancy low. Using smaller buffers causes problems faced with the bursty behavior of data-center flows, and this resulted in the (mis)use of ECN. The main problem is combining the low latency (i.e. small buffers) with burst tolerance. Employing pacing seems promising as to avoid this issue.

On the face of it one may be lead to think that the BDP to buffer ratio is the main impact on what advantages we get from pacing; If the buffers are shallow, pacing shows promise when it comes to packet loss, and if the buffers are larger, the advantage seems more unclear. There is some merit to this if we do not factor in what type of algorithm we are employing. If we use e.g. DCTCP or something like BBR, the advantages of pacing proves prominent even for larger buffers. BBR reacts to the experienced latency (seeking to avoid bufferbloat), and thus attempts to keep the queue occupancy low even though the buffers may be deep.[6] Loss based algorithms on the other hand, may not see the same immediate advantages of pacing, but pacing will reduce queue length oscillations and provide more linear growth and thus more predictable states for the flows. The varying pressure on the bottleneck

[5] Given that they don't share the bottleneck with non-paced flows

[6] As mentioned earlier, bufferbloat is not just persistently full buffers, but persistently unnecessary occupied buffers like "a standing queue that cannot dissipate."[32] If the `cwnd` is larger than the BDP, we necessarily need to fill up the queue to have room for the whole window. The pipe is full, but we have more packets to spare.

is reduced.

8.2.6 Allows for network measurements

Pacing or rate control may be used to do active or passive measurements of network capacity (used in Paced Chirping, Hybrid Slow Start, BBR etc). As mentioned throughout chapter 7, network measurements used for congestion control with other congestion detection methods than packet loss, seem to need some sort of pacing. To be able to do active measurements, one may need to control the gap set in the first place.

Pacing helps to reduce the noise otherwise seen from bursty behaviors in the lower layers.

8.3 Challenges with pacing

Wei et al. argue that the main challenge with pacing is migration. Paced flows may get preyed on by non-paced flows, and this hinders migration into e.g. the Internet much like what happened to Vegas.

Upon further inspection, further challenges seem to emerge. We will try to list most of them here.

8.3.1 Migration

Migrating from a non-paced network to a paced network will not necessarily happen by itself by enabling pacing in some spot and waiting for the others to join. If one flow uses the same algorithm as other flows, but employs pacing, simulations show that ”the performance of paced flows is often lower than that of nonpaced flows when they share the same network”[18]. This is a natural effect from the fact that paced flows are less aggressive than non-paced flows. This is not to say that pacing is not better for the general performance of networks,[7] it only states that the migration from a non-paced network to a paced network does not happen by itself. Figure 8.1 illustrates this skewness. Even though the ”figure is not meant to be quantitatively accurate ”[18] I think it illustrates the issue well. It is an illustration of what the authors call ” the key message we have learnt from a comprehensive set of simulation experiments.”[18]

8.3.2 Delayed signaling

TCP as a flow- and congestion control mechanism needs to pass implicit or explicit information to the sender, causing increase or decrease in `cwnd` or rate. This information either comes from the receiver itself (flow control), or is sent from an intermediate node and relayed via the

[7] Quite the contrary according to Wei et al. Performance is improved in several areas, and they recommend it for both high-performacne computing and applications use. It seems to improve the performance not only of paced flows, but also concurrent non-paced flows

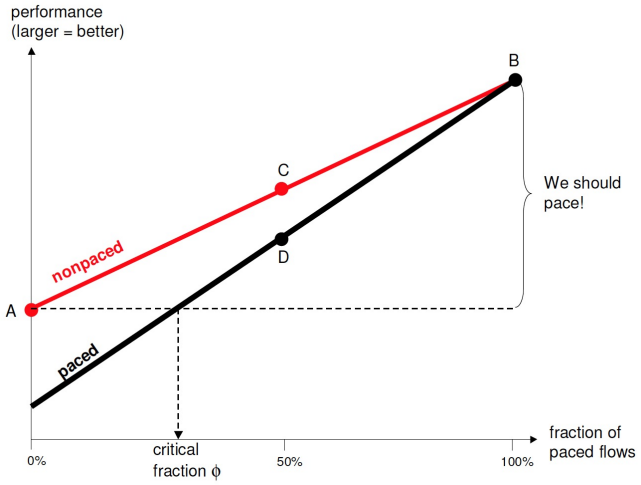

Figure 8.1: Graph displaying performance related to distribution of paced- vs nonpaced flows in a network[18]

receiver to the sender. Pacing out packets e.g. over an RTT causes the last packet to be sent close to 1 RTT after the first packet of the window. If the last packet is lost or causes an explicit notification, that information arrives at the sender no earlier than 1 RTT later than the ACK for the first packet in the window.

If no pacing was employed, that information could have been conveyed to the sender as much as 1 RTT earlier (given that the packets were sent back-to-back and no other latency was introduced. This difference may have a negative impact on the reaction time of the sender, which may cause further negative impacts on the network. Bear in mind that this is a worst case scenario, but any variance between the extremes is possible.

We must also be aware of the other effects that can occur due to our pacing in this example. Pacing makes the probability of the packet experiencing other traffic greater, but it will reduce the odds of other spurious signals like loss or other events caused by the burst that would otherwise be there.

8.3.3 Greater risk of overwhelming the network

According to [1], pacing increases the risk of overwhelming the network because pacing delays the signaling scheme of TCP. The idea is that we steadily fill up the buffer over time, allowing all flows to increase their windows. Once the buffer is full, we have a lot of flows with a larger window than would be the case if we did not have pacing, and this causes more pressure on the bottleneck. Aggarwal et al. argue that this causes more latency and may cause synchronized drops across several flows.

”TCP uses feedback from the network to detect congestion

> and adjust to it. With pacing, this feedback is delayed until the network is saturated, making it difficult for senders to ”avoid” overwhelming the network.”[1]

It is important to remember that this argument is created based on tests with Reno. Reno uses loss based congestion detection, which makes the argument valid. For algorithms more sensitive to latency and queue build-up, this argument seems to fall short. As the authors themselves comment, a good AQM may remove some of these issues. Using AQM with early detection has the same effect as e.g. the delay based algorithms reacting to changes in delay before packets are acutally dropped.

But this assumes that we have buffers with enough capacity to make the sender experience the increase in delay and react to it before we overflow them. If the buffers are shallow, this may not be possible. The signaling is delayed until the buffer starts filling up, and when it happens the buffer gets filled up before the sender notices anything (because we e.g. have a doubling of the `cwnd` taking full effect.

8.3.4 Costly pacing with small `cwnd`

In slow start, we may theoretically start with an initial `cwnd` set to 10. The next RTT, we increase this to 20, and then 40. If we employ pacing from the start, we quickly see that the performance of slow start is heavily reduced.[8] A non-paced slow start may send the 70 packets from the these rounds in 3 RTTs. A paced slow start will try to spread out each window over 1 RTT, causing the transmission of a window to take 2 RTTs (the last packet leaves the sender basically 1 RTT after the first, and the sender has to wait one more RTT for the ACK of the last packet to arrive back. In effect, we may spend close to double the time on each window.

Starting at an initial window of 10 rather than 1 contributes positively here, but the possible reduction in performance in the early stages of a flow still remains.

8.3.5 Multiple concurrent flows in high performance networks

Some recent research indicates that paced flows in data center networks with high bandwidth, shallow buffers, bursty behavior and short-lived flows[9] may perform worse than non-paced flows if the number of concurrent flows exceed what Ghobadi et al. call Point of Inflection [20]. The results merit that some considerations need to be taken into account in certain environments and setups.

[8] Theoretically we start at 1, but the standard today is to start at 10 because 1 so low

[9] Similar to the environments for which DCTCP was created

Point of inflection is the number of concurrent flows where non-paced TCP outperforms paced TCP. They define it with this lower bound:

$$\frac{LinkCapacity * RTT}{Buffersize}$$

In short, Ghobadi et al. found that:

> ”while the number of concurrent flows is below the PoI bound, pacing offers improvements on link utilization, drop rate, average and 99th percentile flow completion times. As the number of flows passes twice the PoI bound, however, these benefits are diminished.[20]

When several flows pace out their packets over an RTT, we risk some packets from different flows to end up back to back on the wire. According to Ghobadi et al., the probability of packets bundling together like this increases as the number of concurrent flows increases. They prove the lower boundary holds by comparing the worst case scenario of paced flows and the best case of non-paced flows. The proof states that the best case of non-paced flows is when the burst of all the concurrent flows are spread out throughout the RTT (hence not overlapping or back to back). The worst case for N concurrent paced flows is when all N flows have the same pace and send their packets at the same time (synchronized) such that we get bursts of size N sent at a the paced rate.

Critique

These findings are important to the discussion regarding whether to employ pacing or not in specific environments. Theoretically, it is not problematic to grasp. The question is how likely these two outer extremes are to happen, and what one may assume to be the most probable middle point.

We need to point out that the calculations in [20] are based on all the flows sharing the same path; i.e. having the same RTT. If the RTT is the same, the rate for two flows on the same link will have the same pacing rate if they both calculate their rate to be $\frac{cwnd}{RTT}$. In a specific datacenter environment, this might be probable, but for Internet situations, this would not hold. It is also important to note that if these presumptions hold, and the number of flows passes twice the bound of point of inflection, the benefit simply starts to diminish, not suddenly disappear.

Ghobadi et al. also claim that the bigger the buffers are (bigger buffer-to-BDP ratio), the lower the number of flows N are needed to make non-paced TCP outperform paced TCP. This may be an extra argument for using pacing in shallow-buffered networks. The tests and reasoning is performed with CUBIC. It is likely that the advantage of paced vs non-paced flows is bigger for BBR or a similar type of algorithm

even though the buffers are deep (simply because the algorithm does not encourage filling up the buffers or allow flows to be aggressive with the buffers).

The experiments supporting the argument are quite constrained in this regard. The results concerning bigger buffers stem from using buffer sizes of 6,8% of BDP where they observe that no packets are dropped even with non-paced traffic. This shows something about the general properties of these tests. There are some constraints in order for this to be a fact seeing as all buffers will overflow if flows in sum send at more than bottleneck rate, and the flows live long enough. Also, measuring by drop-rate is important, but does not exhaust the subject considering other types of algorithms developing today.

Synchronized drops due to concurrency

If concurrent flows cause bursty behavior because the packets from different flows bundle together, we also risk getting a massive degradation of performance for the bottleneck link. If a burst consisting of packets from several flows encounter a queue with high occupancy, we risk packets from several flows being dropped at the same time. These drops in turn cause several flows to back off at the same time, which causes us to under-utilize the link.

> ”increasing the number of concurrent flows sharing the bottleneck increases the inter-flow burstiness, and as a result, increases the chance of many flows experiencing the drop event.”[20]

Mitigations

Ghobadi et al. propose the following solution:

> ”applying per-host TCP pacing on top of per-flow pacing to reduce the inter-flow bursts and drop events. The objective of per-host pacing, or more precisely per-egress port pacing, is to smooth the aggregate traffic leaving an egress port”[20]

If we have a data center where some of the mentioned restrictions are in place (severel concurrent flows with the same path or RTT (most likely from the same end system), pacing out the aggregate seems like a fair solution. If not, it still seems fair to assume that the bursty behavior across concurrent flows seems more improbable to happen than not.

8.3.6 Synchronized drops and under-utilization

Synchronized drops due to inter-flow bursts is not the only synchronization challenge posed by pacing. If several flows share the same bottleneck and apply pacing, we are more likely to experience loss for several flows once the bottleneck buffer is full. If packets arrive unpaced, they

arrive as bursts from each flow. When the bottleneck queue is full, we will most likely drop a lot of packets coming from the same flow, rather than several flows. Spreading out the packets gives us a more fair distribution of packets entering the queues, but we also risk a more spread impact once the queues are full. The number of packets lost per flow is reduced, but it is important to remember that only one packet lost may cause significant impact to a flow. We may reduce the amount of packets to be retransmitted in e.g. fast recovery, but we still halve our window, exit slow start and so on. Spreading out the impact may actually increase the total impact across flows because we inflict packet loss on many flows at once.

Once again, the impact may be smaller for delay based algorithms, but it does seem as a relevant issue that needs to be evaluated. Once flows are in congestion avoidance phase they will inevitably reduce and increase their windows according to the saw tooth pattern. This happens with delay based as much as with loss based algorithms. If we end up with several flows reducing their window at the same time because of the fair distribution of packet entering the queue, we end up with an almost persistently under-utilized bottleneck. Employing an AQM with early detection constructed around the issue may be more relevant.

Hardware limitations and CPU overhead

As links get faster and faster, a new problem arises for pacing. If a link is fast enough, the end system may not be able to space out the small MTU sized packets with such small gaps. The timing may get too coarse. In a case where we have MTU of 1500 bytes and a 10 Gbps link, a packet would have to be generated every 1.2 μs.[10]

This is a problem not only for timers, but as will be clear in the next chapter, a problem for the CPU.

Mitigations

One solution presented in [38] is simply to insert empty packets in between the ordinary packets, thus creating a time delay. These packets are often called *gap* packets or *PAUSE* packets, and their size may be variable depending on what gap is needed. The *gap* packets should be discarded at the first intermediate node (preferably the switch connected to the NIC. Even so, it does result in a lot of resource use.

Another solution is to have dedicated hardware do the timing or spacing. A dedicated NIC may be better suited for the job than a CPU. As networks are getting faster and faster it is all the more feasible that such solutions should be available.[11] Hardware timing or spacing also

[10] 1Gb / (12kb per packet) = 833 333 packets per second.

[11] TCP TIMELY has had a similar challenge because the timers provided by the CPU are to coarse to what TIMELY needs. Dedicated hardware has caused TIMELY to work much better and be a more relevant algorithm today.

saves a lot of CPU-time. The kernel may deliver the packets to the NIC, and the NIC may do the tedious work. E.g. Hany et al. have implemented pacing in hardware on routers egress queues in order to cope with optical switches‘ small buffers and high bandwidth [22].

Chapter 9

TCP Segmentation Offloading

The constant work on improving network performance has resulted in vastly increased bandwidths over bigger and bigger distances. NICs have evolved to cope with this development, but as the NICs evolved, the CPUs needed to keep up with the increasing demands from the NICs. Seeing as the congestion- and flow control is performed in the transport layer (L4), and the NIC is placed in the other end of the stack, each MTU-sized packet and each ACK needs to traverse the whole TCP/IP stack in the host. This requires a lot of CPU resources to when the bandwidth is in the gigabit range.

The communication is bidirectional with packets being sent up as well as down the stack, and the resources used in all stack-traversal are controlled and provided by the kernel. We can easily see that this poses a problem if the speed of CPUs does not keep up with (and indeed surpass[1]) the speed of NICs.

A more viable strategy is to not only increase the speed of the CPUs, but distribute the workload in a smarter and more efficient way. The most widespread approach in this regard is implementing different types of offloading to the NIC; one of them being TCP Segmentation Offloading which is the technique of most interest for this thesis.

9.1 Stateless vs stateful offloading

As previously described, a lot of work is done in TCP to keep track of packets, do error-checking, send ACKs and so on. A lot of these tasks may be handled lower down, and modern network cards facilitate just that. When talking about offloading in the sense of having work handed over to different components (e.g. the NIC), it is purposeful to split the tasks into `stateful` vs `stateless` tasks.

[1] Given that the number of instructions needed to pass packets up and down the stack and process the contents in the receiving layer is greater than receiving and forwarding packets in the NIC

9.1.1 Stateful offloading

In stateful offloading, the NIC needs to keep some sort of state table and be able to perform tasks depending on different states. One of these tasks may be responding to the sender with an ACK if that is the next desired step in the communication. In order to make the decision to send an ACK and create a proper ACK, the NIC needs to keep track of some of the state normally kept at L4. By letting the NIC keep some state tracking and act upon information along the way, the kernel may be spared for traversals and monitoring. Several different variants of stateful offloading exist, from small tasks all the way up to what is call full stack offload where the whole stack is offloaded to the NIC. This may seem like a very promising approach, but it puts a lot of constraints on the L4 possibilities as the NICs are hardware designed to specification. If one wants to make some adjustments to an algorithm it may require new hardware.

9.1.2 Stateless offloading

Stateless offloading tackles tasks not requiring knowledge exceeding the information contained in the packet at hand[2]. Typical tasks include checksum-calculation, TSO, LSO, Large Receive Offload (LRO), Receive Side Scaling (RSS) and TCP Support for Sensory nodes (TSS).[3] Some of these are specific to TCP, and some are more generic. A common aspect to all of them is that they reduce the pressure on the kernel of the host by either doing some labor that would otherwise be done by the CPU or by doing some sort of batch operations.

9.2 Segmentation Offloading

The batch operations found in segmentation offloading can be split into two; sender side and receiver side. The use of one does not imply the use of the other. Each end system handles this irrespective of the rest of the path.

9.2.1 Sender side

Sender side we have the aforementioned TSO. TSO is also referred to as LSO. LSO is the general term in TCP for offloading the CPU by bundling several packets into one larger packet before passing them down to lower layers. This type of offloading is not unique to TCP, and a completely generic term for offloading of this type regardless of protocol is Generic Segmentation Offload (GSO).

[2]That packet coming from the wire or from L4

[3]Letting intermediate sensor nodes cache TCP data segments and perform local retransmissions in case of errors

9.2.2 Receiver side

Receiver side we have LRO and Genereic Receive Offload (GRO) which are not more mysterious than the variants sender side. They are the TCP and generic variants of the bundling of packets in the lower layers before passing them up the stack.

9.3 TCP Segmentation Offloading

TSO is specific to TCP. As such, it is the main focus of this thesis. TSO is in widespread use and is the default on most systems where possible.

9.3.1 TCP/IP stack traversal

When TSO is enabled, the transport layer fills up a memory buffer of up to 64KB, and passes the whole buffer down to the driver. In the Linux kernel (which we use) this process is made possible and efficient by the help of a data structure called `sk_buff` . This data structure is a *struct* with pointers to the different parts of the buffer. At the transport layer, a TCP header is added to the payload, and this is done by writing the header to the area adjacent to the payload itself, and the pointer to the header start is moved to the header start of the TCP header.

A pointer to the `sk_buff` is passed down from one layer to the next, and each layer adds on the header as needed along the way as illustrated in figure 9.1. This way, the payload rests in the same memory location the whole way, and can be easily copied into the NIC once it reaches the NIC driver. Once inside the NIC, the payload is split up into MTU sized packets, headers are duplicated for each of them, and they are queued for transmit.

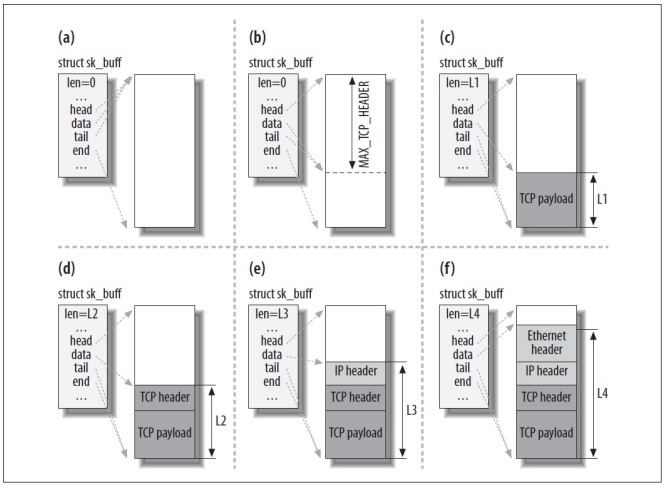

Figure 9.1: Buffer being filled in while traversing the stack from the TCP layer down to the link layer [9]

9.3.2 TSO automatic sizing

While using TSO has great advantages it has seen some adjustments along the way. At lower rates, using TSO can have a lot worse effect on burstyness than in networks with higher rates. The lower the rate, the more time is spent batching up each TSO-segment. This can result in noticeable periods of time with no packets before a new burst of packets are sent out. For higher rates, this is not problematic as it only contributes to keeping the link saturated, but on lower rates it could cause more fluctuations in the network than needed. In sum it could have more disadvantages than advantages.

Because of this, Eric Dumazed in proposed to add a patch to the kernel reducing the potential negative effect of TSO on such low rate flows. The patch allows the TSO-size to be dynamically adjusted so that we always send out a segment at least every 1 ms.

This reduced the bursty behavior noticeable when windows were small or speeds were low. It is a limitation one needs to keep in mind when working with TSO and pacing. TSO autosizing is enabled by default and is part of the main output file for TCP in the Linux kernel, `tcp_output.c`

9.3.3 Limitations to offloading in traditional NICs

TSO causes some headache for the algorithms employing pacing. By offloading the job of splitting payloads into smaller segments, we lose control of the departure time for each packet. Several algorithms want detailed control over every departure-time, and even for those algorithms that do not employ pacing, it is arguable that network throughput could be improved by spacing out packets rather than lumping them together the way TSO may end up doing. As Welzl et al. [34] summarize it:

> ”Because software timers do not have control over the individual packets that are created from such a larger data block, TSO is usually disabled when using pacing. As an alternative, Linux can dynamically change the size of the TSO data blocks. Naturally, reducing the size of TSO blocks comes at a cost”

Yet another issue with the various forms of offloading is the fact that the offloading is done using programmed hardware. The NICs are designed from the producer or on request from a customer to offer a specific set of offloading capabilities. Once the NIC is constructed, the capabilities are well defined and unchangeable.

The advantage of hardware processing is speed and reliability. The disadvantage is the lack of adaptation. As networks keep evolving as fast paced as we have seen over the last decades, such classical NICs may surely pose some limitations to the possibilities one would hope to have. Because of this, the last decade has given rise to a new form of

network hardware in what has been named Smart NICs and dynamic datapath programming.

9.4 Pacing in hardware

The challenge with doing pacing in hardware is that hardware traditionally has had a need to be customized by the manufacturer for specific tasks, but this is about to change with the emergence of programmable hardware. One programmable hardware variant that has been around for a while is FPGA-circuits. FPGA allows the programmer to manually configure the circuits to do what is wanted.

9.4.1 Hardware calendar

Many functioning FPGA implementations of various data path programming exist, one of which is a hardware calendar created by Welzl et al. [34] intended for use in TCP. The main idea is to be able to do pacing with high resolution in the NIC while keeping TSO enabled. That is, enable software timing at the same time as we pass larger TSO blocks to the NIC.

The hardware calendar shows that we are closing in on possibilities not seen before when it comes to precision, flexibility and programmability of hardware. Their solution uses FPGA and enables us to have a per packet pacing which may be infeasible for CPU controlled pacing due to a lack of enough resolution. They also make it possible to have a more dynamic hardware which can be quite easily changed compared to traditional hardware in need of manufacturing.

9.4.2 New SmartNICs and abstractions

The use of FPGA is very promising, but as we will see in the next part, more flexible and accessible APIs may become available with the new Smart NICs being developed these days. The more depth of control we get over the NICs, and the more speed, flexibility and adaptation, the more promising a complete offloading of pacing seems.

The second part of this thesis will investigate the possibilities provided in the Netronome Agilio SmartNICs. Our main aim has been to see if the abstractions in P4, C and MicroC provided by Netronome can enable us to employ pacing in hardware. If so, we could get the best of both worlds; a finely grained clock in hardware and TSO without creating big bursts in the network.

Part II

Solutions in a world of SmartNICs

Chapter 10

SmartNICs and data path programming

It has become a well known fact that Moore's law[1] is coming to an end as the laws of physics tell us that the more dense the chip is, the more heat it will produce, and the more energy will be needed to cool it down, thus reducing the efficiency. Moore's law has in principle been dead for a while, but the claim on computing power has prevailed thanks to Dennard scaling and Amdahl's law. Now they too are facing a challenge to prevail.

One of the issues of today is the workload put on the CPUs by the more and more demanding network processing. Distributed processing and cloud solutions are pushing the limits of networks, and private end systems are also catching up. The CPUs are at a stretch to keep up with the amount of data and processing needed to maintain high speed networks. Especially for demanding protocols like TCP.

10.1 NICs to the rescue

While the rate evolution of CPUs is flattening, the NIC development is still steady. NICs are getting more and more capacity and more and more capabilities. Thankfully these capabilities can help the CPU out in doing more work.

10.1.1 Offloadable NICs

A CPU is designed to be versatile, handling anything needed by the kernel, while a NIC can have highly specified processing finely tuned for fast and efficient processing of a smaller set of operations. Because of this it has become natural to make the NIC offload the CPU to do tasks that are easily done further down the stack.

[1] Moore stated that he number of transistors per square inch would double roughly every 18 months. This implied that CPUs would become increasingly more efficient as time went by, and it is a ”law” that has been quite accurate for a long time.

The offloading mentioned in the previous chapter has been present for a while in what we may call offloadable NICs. The simplest offloadable NICs have a processor capable of doing predetermined tasks like TSO or the like. They may also assist in checksum calculation and other simple tasks.

10.1.2 SmartNICs

If one wants more advanced or custom offloading, the way to go has been to order customized hardware from the manufacturer. As long as we have a long term perspective on what we want customized, that can be a viable solution. But customized hardware is costly and can take years to deliver.

As fast as algorithms and use cases develop today, this sort of cost and delivery-time is not necessarily good enough. Spending time and money to put custom solutions into silicon is simply not always worth it in systems of today. It is also less versatile for the customer because the level of customization can only go thus far. On top of that it forces the customer to share private information with the hardware manufacturer which is not always ideal.

Enabling the customer to manipulate the hardware itself has emerged as a good way to create customized solutions for faster networks and saving more CPU usage than ever before. The hardware is broadly nicknamed SmartNICs because they are ”smart” enough to be modified, but their exact capabilities vary about as much as the number of variants of producers and cards.

One variant of SmartNICs is FPGA. FPGAs have been around for a long time and are well known for their customization capabilities and speed. They enable the user to change the logic blocks on the card in such a way that they can perform highly customized operations like the hardware calendar in section 9.4.1. The process for this is quite cumbersome, but at least it offers a highly customizable piece of hardware.

Some new SmartNICs take this a step further and add the possibility for customization through an even simpler API, and may in some cases offer more computational power too.

We will define a SmartNIC as a NIC capable of offloading, having computational power and capability of reconfiguration of the hardware. Most SmartNICs have all the offloading capabilities seen in standard offloadable NICs.

10.1.3 Hardware acceleration

Using dedicated hardware to offload specific tasks from the general purpose CPU is broadly called hardware acceleration. The use of hardware specialized for the tasks at hand is a lot more efficient than using the general purpose hardware found running the kernel.

The term hardware acceleration can be used for a lot of different applications such as GPUs handling graphics processing, machine learning/AI, sound or signal processing and a lot more. It can also be a type of offloading internally on a NIC as we will see in the next section.

One could argue that there is a distinction between hardware `offloading` and hardware `acceleration`. We will not make that distinction in the following, but it does not seem relevant to the term as we use it either.

10.2 Netronome Agilio SmartNICs

The Netronome Agilio family of SmartNICs is a series of fully programmable NICs. The firmware of these cards can be downloaded and uploaded just like any other application. It can even be uploaded while the NIC is running. This opens up the possibility of having them reprogrammed remotely using IP, possibly enabling a user to dynamically change the hardware of a whole path without having to do any physical modifications.

The Agilio SmartNICs have a lot of offloading capabilities, and they offer programmability through P4, C, a hybrid of them both and MicroC. This section will create an overview of the hardware architecture before the next sections examine the actual possibilities this offers using the different programming techniques.

10.2.1 NFP-4000 Flow Processor

In our project we are using Netronome 4000 2x10GbE cards. As such, they fall in under a group called 4000 for short. The 4000 family comes in a variety of specifications (2x10GbE, 2x25GbE, 50GbE and more still), but they all share the same basic architecture with the same amount of cores and functionality.

The 6000 family is built up in much the same way, just more of everything.

10.2.2 Micro Engines

One of the most important components of the Netronome 4000 NICs is their 108 Micro Engines (ME).[2] The MEs are divided into two types, Flow Processing Core (FPC) or Packet Processing Core (PPC).

There are 60 programmable FPCs and 48 PPCs. Each core has 8 threads, enabling the NIC to process up to 480 packets simultaneously.[3]

The packet processing is performed by the FPCs. The PPCs facilitate and aid the FPCs in their work. More on this later.

A key takeaway with the cores on the NIC as opposed to CPU cores is not only their optimized packet processing, but also the high level

[2] Netronome themselves use the term ME and Processing Core interchangeably

[3] Each thread handles one packet and we have at most 60 FPCs handling packets

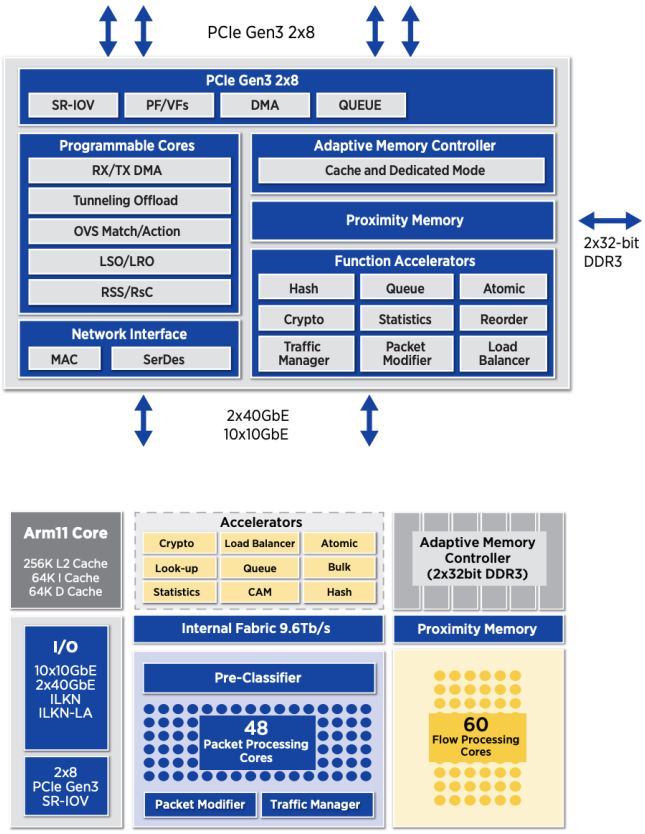

Figure 10.1: NFP-4000 Flow Processor Block Diagram [31]

of concurrency and fast context switching. A CPU core has a different purpose and needs to support more long lived processes and complex instruction sets.[4] It therefore has more invasive switches.

The whole NIC is built around making the cores work as efficient as possible with packet processing and nothing else. This includes internal hardware acceleration, fast memory and very fast thread switching (2 cycles).

10.2.3 Hardware organization

To help out in enabling the cores to do their work as fast as possible, the NIC has a lot of hardware function accelerators that offload the FPCs to keep the cores focused on packet processing and making this as simple as possible.

The hardware function accelerators include functions like statistics collection, traffic management, load balancing and lookup. Their functions are mostly atomic and lockless, which lets the threads work without having to worry about multi-thread issues. There is also a load balancer that takes care of picking the next available thread or the one

[4]The Netronome Agilio NICs have an instruction set optimized for networking whereas a standard CPU needs to support a full general purpose instruction set

with the least amount of backlog at all times. This way, the threads mostly need only concern themselves about themselves.

It also makes it a lot easier to the programmer writing code for the FPCs. The actual programs for the FPCs can be simplified a lot, less concern can be put on thinking about message passing and synchronization, and tasks one would otherwise need to write out can be omitted as the hardware accelerator functions take care of it.

The FPCs and PPCs are organized into what Netronome refers to as islands. An island consists of 12 cores, and the Netronome 4000 NIC family has a total of 5 FPC islands and 4 PPC islands.

In addition to the accelerator functions and MEs, we have various memory areas, memory management, and a distributed switching fabric connecting it all together.

10.2.4 Scheduling

When software is uploaded to the NIC and runs, we have a thread pool handling packets picked from the queue. All the threads are scheduled with a ”run to completion” style where every thread runs until finishing or yielding.

The scheduling is handled by the PPCs and is not part of what the programmer will handle in standard programming of the NICs using the Netronome abstractions.

10.2.5 Memory

In standard data plane processing we have match/action handling where actions are performed based on packets matching some criteria defined in a table. These types of actions (part of the FPC actions) are performed in on-chip memory for best performance. The on-chip memory is a single cycle memory directly accessible to the core, making the actions very efficient.

Host memory is also accessible if wanted. We have multiple PCIe buses which allow us to connect to different host sockets directly so that we can have software running with the memory of a specific socket if needed.

Packets are placed in a work queue designed as a ring. Depending on what processing is needed, this queue can be accessed by several MEs as well. This makes it possible to offload specific work to other MEs so that a thread can have a thread on a whole different ME assist with some specific custom function being done.

When passing a packet off to another thread, one can use signaling through signal-pairs to have the packet passed back to the original thread if needed. If not, the new thread can handle passing the packet off to the next step in the pipeline. More on the pipeline and ingress and egress queues in section 10.3.

The memory on the Agilio 4000 NICs can be divided into these main regions:

- Local Memory
- SRAM (deprecated)
- MU (Memory Unit)
- CLS (Cluster Local Scratch)

The MU consists of Internal Memory (IMEM), External Memory (EMEM formerly known as DRAM) and Cluster Target Memory (CTM). Each memory area has its set of capabilities suitable for different purposes.

When programming in MicroC or C on these cards you need to specify which memory region allocations are done to. This is a requirement and one should make considerations as to what area of memory would best suit the purpose. In addition to specifying what memory region we want to allocate from, we can also specify where in memory the pointer should reside:

```
__declspec(mem) buffer* __declspec(cls) buf_ptr_1;
```

10.2.6 Code and variables

Code is uploaded per ME, making all threads in one ME share the same code, just with their own copies of variables and data structures in IMEM. Each ME has 256 general purpose registers (GPR) and 512 transfer registers used for communication with I/O and memory (256 in and 256 out). They also have 128 next neighbor registers (NN) which act as both in- and out-registers for communication between neighboring MEs. These registers are either in receiving mode or sending mode, allowing for either read or write to or from the closest neighboring ME.[5]

As we can see the whole structure of registers is focused on concurrency and fast communication between threads and MEs.

Every ME has a local memory the size of 4KB private to the ME. This memory can be shared via registers between contexts or kept completely local. By default, all variables and data are kept local to the ME.

The system also allows for easy declaration of variables shared among all threads on the ME with the parameter ”shared” and a specifier for what memory region it should reside in. Note that variables in local memory cannot be exported to other MEs.

Declaration of variables on global or island scope is just as easily done using the ”export” and ”import” arguments and defining the scope to be either ”global” or ”island”.

The EMEM is slower, and access is asynchronous. The threads can either wait for a signal for completion or swap themselves out while

[5] Specified by setting a bit

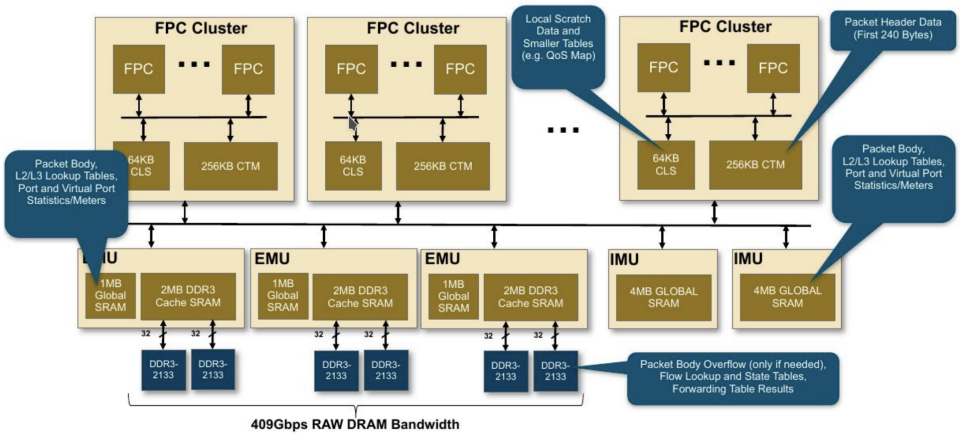

Figure 10.2: Overview of all important memory-types[33]

waiting to allow other threads to run. The manuals do not specify with code how the scheduling is handled for threads waiting on I/O, but it is mentioned in several places that the thread model found on these cards is designed to prioritize thread-swapping when waiting on I/O to maximize throughput.

	Thread Local	Shared	Export	Remote/Visible
GP Register	Yes	Yes	No	No
Transfer Register	Yes	No	No	Yes
Signal Register	Yes	No	No	Yes
Next Neighbor	Yes	No	No	Yes
Local Memory	Yes	Yes	No	No
SRAM (legacy)	Yes	Yes	Yes	No
MEM	Yes	Yes	Yes	No
Cluster Local Scratch	Yes	Yes	Yes	No

Figure 10.3: Allowed combinations of attributes on data [29]

10.2.7 The power of threads

The architecture is what we can call a thread-based architecture, not a cache-based architecture. Multithreading gives a better effect than simple cache-usage when it comes to networking because the NIC gets constantly bombarded with packets, and each packet may need completely different information or treatment.

One packet may need looking up into one table for routing, while the next one may need something completely different. This would require very much memory to be efficient the same way threads may make work done, having one thread do one thing on one packet while another one is doing something else with a different packet.

The high number of cores and large number of threads and helper

functions, combined with their specialized instruction sets, is the secret to the efficiency of these NICs and why they can enable us to save CPU time.

10.3 Packet flow

The flow for a packet depends on whether the packet is going from host to network or arriving from network to host. The flow for host-to-network is a bit simpler than the one for network-to-host.

10.3.1 Network-to-host

In the case of a network-to-host packet, the journey starts with the load balancer.

Each island has its own ring buffer used for ingress packets. The egress buffer is shared among all the islands. In a standard scenario all the packets share all the threads to achieve the highest bandwidth possible.

With one work queue accessible for the load balancer, it can make sure to pick an available thread or the thread with the least amount of backlog and pass the packet to the ring buffer of the suitable island.

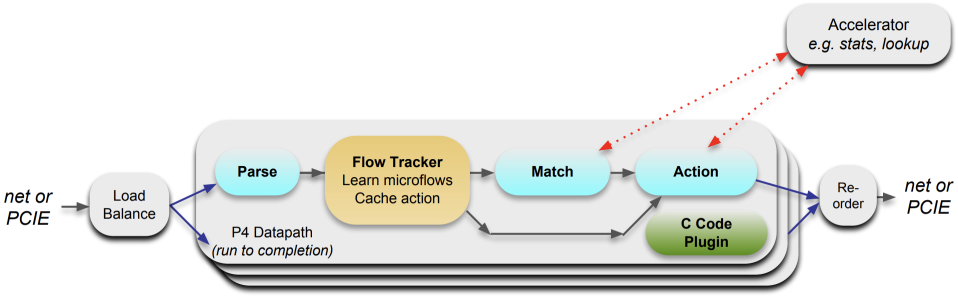

Figure 10.4: Packet Flow through chip [33]

The packets get parsed by the parser (done by the 48 PPCs), adding on metadata if needed, and saving the headers in CTM-memory of the designated FPC island destined to process the packet. All threads in the island can access CTM and pick a packet. This process is software controlled.

Once the packets are parsed and get picked up by a thread, they are run through a set of match table lookups. This matching can consist of several lookups, and the process can be further improved by using a flow tracker which can match up with a cached result table. A hit in this table will make the flow bypass the matching step and go straight to action.

Due to the concurrent processing with a pool of threads, order is not guaranteed. Threads may spend a different length of time processing a packet, and we therefore need to take care of reordering afterwards before the packets are transmitted.

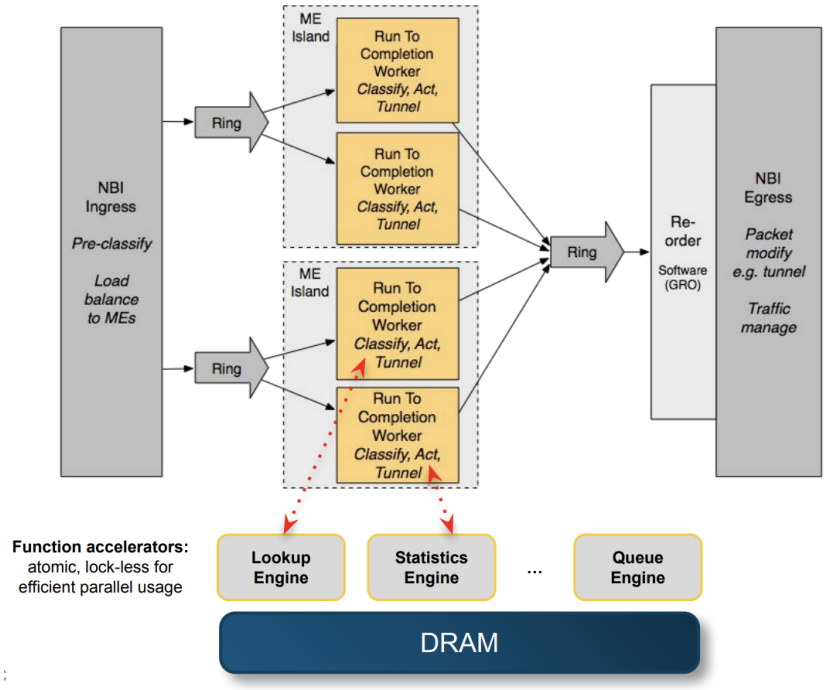

Figure 10.5: Packet Flow through chip detailed [33]

All the packets from all the MEs are placed in one ring buffer and the chip has built in reordering working on that buffer ensuring that we maintain order. The parallelization is handled in its entirety by the NIC by providing load balancing, function accelerators and reordering.

10.3.2 Host-to-network

In the case of a host-to-network path, the flow is a bit different. The packets are delivered to a PCI-island using the tx-descriptor from the driver. The packets are stored in the PCI-island before being moved to the appropriate next handlers (e.g. the FPCs).

Once the packets are passed on to the FPCs the process is much the same as for Network-to-host processing. See figure 10.6. The hardware accelerator functions provided by the egress PPC[6] are said to be configurable by appending metadata to the packet in the FPC, but the documentation here is not very clear.

10.3.3 Data Path offloading

The flow of packets through the chip is a concurrent effort to process packets according to a table of actions. Looking up and performing actions based on various criteria is standard, but these NICs allow us to program this match-action into the chips. It can also be done with a

[6]Traffic Manager, Packet Modification Engine and Reordering Engine

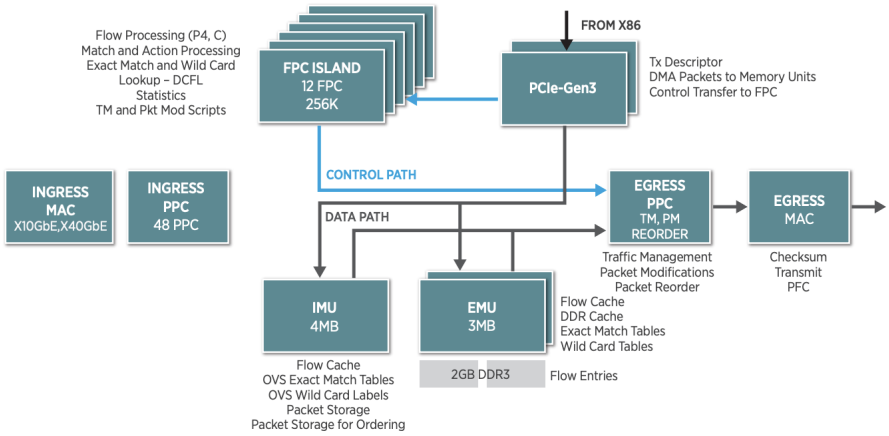

Figure 10.6: Host-to-network flow [30]

lot of customization where we program the data path for the NIC using P4, eBPF or C.

The topology shown in the figure 10.4 is the standard variant, but it is possible to change this to have a different flow with e.g. some processing done by other cores. This can be done using the built in areas of communication and it can be implemented using P4 or some hybrid variant with a P4 data path with embedded C functions that can do custom packet handling. The C functions can have fully customized parsing, and the control/action table can be programmed with P4.

Chapter 11

Programming Agilio SmartNICs

11.1 eBPF/XDP

This is perhaps the simplest way of offloading data paths. The NICs simply offer the possibility to offload eBPF or XDP directly to the NIC rather than having the CPU do the work. As such it works as a really good hardware accelerator for data path processing that can be written as eBPF or XDP.

We will not go into much detail with the use of eBPF because eBPF has a lot of constraints, one of which is the lack of support for conditional loops. If we are to introduce pacing in some way, it is obvious that we will need to have conditional loops or some other ways to wait out specified lengths of time, and that length of time needs to be dynamically adjustable and not pre-programmed as a set number of cycles or something similar.

11.2 P4

The programming method prioritized by Netronome on the Agilio SmartNICs is standalone P4 or P4 in some combination with C.

P4 is an open source, domain-specific programming language designed specifically for data plane programming and packet processing. It is optimized for hardware implementation and aimed at making the construction of match-action tables as custom and easy as possible. It is a relatively new language, originating in 2014 in ”P4: programming protocol-independent packet processors”[12].

P4 is intended to be as versatile as possible and is based on a Protocol Independent Switch Architecture (PISA). The PISA-architecture can roughly be defined with these three parts:

- A programmable parser
- The programmable match-action pipeline

- A programmable deparser

The parser is a state machine which parses the packets into a parsed representation. The headers are stored one place in memory and can be manipulated along their way, while the payload resides in a different place in memory.

The match-action pipeline can consist of various control actions like table lookup. We can have several stages of this pipeline, each with a specific match-action.

The deparser is a simple state machine which puts the whole packet back together.

With P4 we can further improve on this architecture by having more complex pipelines and communicate or transfer packets between islands, but the base architecture is based on PISA.

The P4 compiler exists for many different devices (targets), making it versatile for the programmer. The pipeline and architecture can be handled as a black box where the hardware itself fits into the interface needed for the compiler. Compilers exist for many manufacturers and many types of devises, from Netronome Agilio to Altera with FPGA.

11.2.1 Limitations

P4 is a quite pure data plane language, and as such it is mostly stateless. The packets are processed using lookups and match table logic. As we are working with hardware, it is highly optimized to do these types of operations.

The P4 pipeline can be dynamically programmed, but it has not got unlimited capabilities. Scheduling and buffer management is also harder to control, which will prove to be a important to achieve our goal. More on this in the next chapter.

More advanced functions can sometimes be better handled using C or MicroC, and the P4 programming environment from Netronome supports extended functionality using C.

11.2.2 Netronome Programmer Studio IDE

The programmer studio created by Netronome offers a full GUI for programming the NICs using P4 and C. It is only available on Windows, but works perfectly fine using Wine or a VM running Windows. We used Wine in our experiments and it worked fine. The programming environment offers all the standard functionality found in a modern IDE with a more traditional high level programming language like Java.

Below we show a P4 program as basic as they get where we can see all actions from parser to deparser. Normally the program starts out with defining how to parse headers so that they get put in local memory (1 cycle access) before moving on to the control flow where we state what to do with the packets.

A basic P4 program dropping packets as shown in the NFP-lab 1 :

```
header_type eth_hdr {
    fields {
    dst : 48;
    src : 48;
    etype : 16;
    }
}

header eth_hdr eth;

parser start {
    return eth_parse;
}

parser eth_parse {
    extract(eth);
    return ingress;
}

control ingress {
    apply(encap_tbl);
}

table encap_tbl{
   actions {
        drop_act;
    }
}

action drop_act(){
    drop();
}
```

The programmer studio also provides a debugger which lets the programmer inspect specific MEs and step through the code line by line while it executes on a thread on an actual Netronome SmartNIC on.

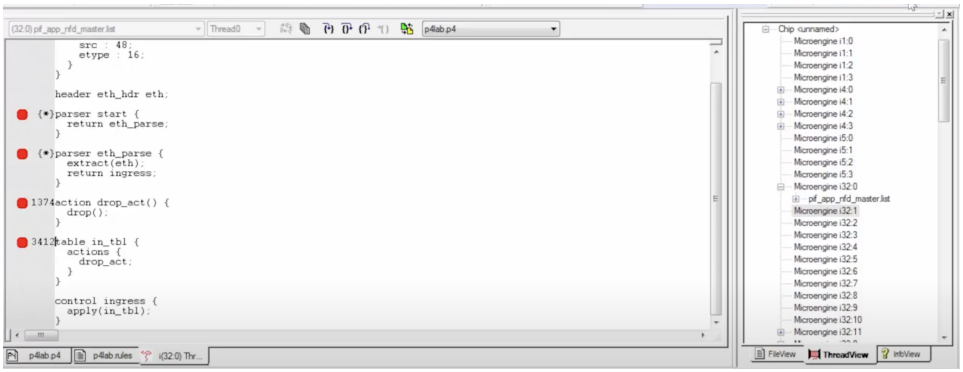

Figure 11.1: P4 debugger as shown in P4CDevCon Lab1 by Open-NFP https://youtu.be/f8b9y2P6Ib8

11.2.3 Actions and helper functions

The actions are defined in separate rule-files defining the tables. Setting them up is quite straight forward, but as previously mentioned it has some limitations. P4 is evolving, but if you want to have as many possibilities as possible, injecting C code is the way to go.

When defining the actions, you can use standalone C functions. These C functions have access to everything that the P4 space can access (headers, metadata, memory, co-processors and libraries). In addition to this, creating functions in C gives us a greater liberty to create more custom handling with more statefull processing and more complex logic because C provides us with better data structures. It is also possible to do advanced filtering and handling with deep packet inspection with access to the packet payloads in our C code.

The functions are written in standard C style in their own files. The compiler handles all the linking of the C code to the P4 code. All we need to do is call the functions using predefined keywords. Everything else is handled by the compiler. This code also supports the use of timers. Using a combination of `me_tsc_read()` and `sleep()` one can loop and make things happen with a delay.[1]

11.3 MicroC programming

Sometimes we want to control the flow in a more bare bones way than using P4 through the IDE. When using MicroC it is possible to write small bits of code and compile and install them without all the P4 programming. In a lot of cases it can be a lot simpler to simply write a small snippet in MicroC than solving it using P4 and e.g. some callback in C. With MicroC you can program just the part you need, e.g. some custom packet matcher.

MicroC is perhaps more familiar to a lot of programmers with its C syntax than P4.

As with P4 in the IDE we can also specify details for the compiler as to how many MEs we will have execute the code (as all MEs have their separate code).

11.3.1 The C compiler

The C compiler allows for most types of functions, but there are some limitations worth noting:

- Recursion
- Variable length argument lists
- Pointers to functions

[1] This is essential for what we are trying to achieve, and we will explain in more detail later, but as will become clear in the next few sections, we had to abandon this path for P4 due to other obstacles.

- Passing aggregates larger than 64 bytes[2] as function arguments or return value.

The compiler is based on C89, and most of the C99 additions are not supported. There is extensive support for inline assembly. Various flags for compiler optimization are available (like loop unrolling)

Again, we have the possibility of doing time-based actions like `sleep()`, and we have done tests using this functionality in section 12.3.1.

Netronome provides good descriptions and tutorials on their websites for programming in MicroC on the Agilio NICs, and we will not go into all details here. Some more help concerning troubleshooting, pitfall avoidance and usage can be found in the GitHub repository for this thesis.[3]

11.4 Compiling and uploading - our main challenge

When compiling and uploading P4 or hybrid C code we can use the IDE. When compiling and running MicroC code, we compile and install the firmware much like a standard C program. In order to enable firmware modifications, though, we must first enable the Run Time Environment (RTE).

The appropriate package needs to be downloaded, and in our case this was the `nfp-sdk-6-rte` binary for Ubuntu. The necessary files are available through login to Netronome support pages which will be provided by contacting Netronome with information about the NIC you have purchased.

Once the files are obtained, the RTE-service can be installed using the shell-script in the folder. At this point you may run into some issues. We recommend following a great guide provided by Mauricio Tavares UnixWars installing Netronome drivers.[4]

In addition to the challenges listed by Tavares, it is not unusual that there will be a conflict between the upstream driver and the RTE-installer. After reaching out to Netronome, we were told we needed to manually remove the `agilio-nfp-driver-dkms` before continuing:

```
sudo apt-get remove agilio-nfp-driver-dkms
```

With the driver removed, we could install the SDK and run the RTE-service, and once the RTE-service was running, code could be dumped onto the cards using IP and the port number of the service for P4, and

[2] 128 bytes in 4-context mode. In NFP Enhanced Mode, 128 bytes in 8-context mode, 256 bytes in 4-context mode.

[3] Permki PacedLinux https://github.com/Permki/PacedLinux

[4] http://unixwars.blogspot.com/2019/05/programming-netronome-network-card.html

by using the provided Makefile for the MicroC program.[5] This is also how we can run the debugger.[6]

11.4.1 Missing host-to-network functionality

Installing and running the RTE-service worked fine, but running our custom P4 or MicroC firmware created a whole different problem.

The intended use for P4 this way is to have it loaded onto the NICs the RTE service is running. When trying to run a simple packet handler with P4 running the RTE-service, the interfaces formerly associated with the NIC disappeared from the list of interfaces on the host.

We could run our P4 programs, but only using the host basically as a switch, not as a host sending out packets. This made us attempt a simple program in MicroC instead, but it had the same result. We could install our new firmware, but that resulted in the NIC not being accessible to the host.

We contacted Netronome support and discovered that this is the intended result according to Netronome. When installing our own firmware (P4 or C), the upstream firmware recognized by the OS will not be present, and hence the normal functionality as a host NIC will not be available.[7]

Our intended use of the NIC as a host to network card clearly is not the same as the intended use from the manufacturer. The main target group for these NICs is accelerating networks by dynamically programming switches, firewalling and similar functions throughout a network, not to offload the sender or receiver. This is problematic for our use case. Without being able to use the NICs as host, the whole idea of offloading and pacing in union with L4 is not doable.

This finding is worse than other challenges as it actually excludes the use of the abstractions. There is no need to investigate modifications to the abstracted methods unless there is some way of combining the upstream firmware with the modified code provided either with MicroC or P4.

Even though this seemed quite hopeless to our project, we still did tests and other modifications needed before further investigating the issue and looking at possible workarounds. The workarounds and optional solutions will be discussed at the end of the next chapter after further laying out our implementations.

[5] The default port numbers are 20206 for the RTE-service and 20406 for the debugger

[6] Note that the debugger can require as much as 8GB RAM on the node and can be quite straining if the machine is of a smaller size.

[7] See Problems starting RTE-service
https://help.netronome.com/support/solutions/articles/36000152964-problems-starting-rte-service
and Interfaces not present
https://help.netronome.com/support/solutions/articles/36000152961-smartnic-interfaces-are-not-present- for more information

11.5 Traffic Manager

The traffic manager is not part of the data plane constructed using P4. It resides in the control plane. The same goes for the buffer manager. There is limited control over the traffic manager offered to us using P4 at the time of this writing. According to David George in Netronome (2017) they are planning on implementing the feature in a later release.[8]

There seems to have been released some version later with support for some control of priority queues, but the documentation is sparse. The issue definitely could need some further examination in future work as it could prove to be very useful for a working solution using P4 or C.

[8]Claim found in the NFP forum (Google Group https://groups.google.com/g/open-nfp/c/GVZ9GzEkMNw/m/UcGzx2kEDwAJ)

Chapter 12

Outlining a solution to pacing in SmartNICs

The overall plan for this project has been to have L4 calculate an ideal rate for packets given a `cwnd` and RTT and pass that information down to the NIC so that the individual packets created from a TSO-segment could be spaced out with a delay causing the actual rate of packets going out on the wire to be the same as the calculated rate from L4.

Classical offloadable NICs are not capable of doing such work, but on paper our SmartNICs from Netronome might be able to do it. Using the abstractions from Netronome is not viable for our intent of use, but they reveal that some needed underlying functionality is present, like clocks and timers. Before we can look into possible workarounds for the missing interface challenge, we must first make sure we have some other elements in place.

Mainly, we can split this problem into two parts:

1. Message passing from L4 to the NIC
2. Make the NIC pace out packets using the information passed down

The hardware implementation in number 2 is the main focus of this thesis as it is the main area of concern, but the decisions made concerning communication from L4 will affect the solutions to be discussed for the hardware implementation. Therefore we will first delve into our thoughts and implementations of message passing from the upper layers before exploring the hardware solutions.

12.1 Means of communication

As the Linux kernel uses the `sk_buff` both to pass packets up and down the stack and to collate information for TSO, the `sk_buff` would seem like an obvious data structure for message passing from L4 to the NIC.

The `sk_buff` has an area called `skb_shared_info` which is accessible for all layers. Here we find information about GSO segments, length

and other relevant information for the bundled packets contained in the buffer. A possible solution to the communication challenge is to expand the `skb_shared_info` struct with the information necessary to calculate a pace-rate. It seems like an obvious solution, but it is not unproblematic.

In the transition to the driver, things become more complicated. The buffers get moved to `tx-ring` buffers, and it is unclear where, how or if this information can actually be picked up by our custom code further down in the NIC. To make sure the relevant information gets passed all the way down we could risk having to modify the drivers themselves, which is error prone and difficult in itself.[1]

The appeal of using `sk_buff` lies in its simplicity. Modifying a struct makes it easy to implement and (in theory) easy to parse out. Our challenge is that we do not have full control of how the packets are passed further down before they eventually end up parsed in memory ready to be processed.

One thing we know for certain is that the information in the actual packets is persistent throughout the packets lifespan from one end system to another (or until it is wholly lost and gone). Because of this, we opted for using the packets themselves. It may not be the optimal way for a final solution, but it is a very safe way to do initial implementation and testing. It also makes it easier to modify the content along the way if needed.

12.1.1 Message content

Pacing out packets is the same as waiting a specified amount of time (a gap) before sending the next packet in line. Seeing as the classical TCP algorithms operate with a max send rate of 1 `cwnd` per RTT, the gap may, in its simplest version, be expressed as:

$$\frac{RTT}{cwnd}$$

This of course assumes that we want an equal distribution of all the packets in one TSO segment. If we were to further explore more complex solutions here it would be very interesting to look at a more smoothing function. One could have an average gap for the segment, but distribute it unevenly either for use in an algorithm like Paced Chirping or some other algorithm focusing on e.g. the slow start phase.

To convey this information we can either send the rate, the desired delta-time, or pass the variables themselves; RTT and `cwnd`. We decided on using the delta-time as it is simple to handle and, most importantly; unambiguous. The delta-time passed down is the desired amount of delay between two consecutive packets in a TSO segment. It is important to note that this is the desired delay of one packet relative

[1] Professor Andreas J. Kassler from the University of Karlstad, Sweden, joined in on some fruitful discussions on this topic with his experience using the Netronome SmartNICs.

to the previous packet in the original line. We will discuss this later on, but it is not necessarily trivial to control the order of the packets and how to ensure that the delays are added when wanted, so we may need some extra calculations here.

With all these unkowns, we decided on uing using the desired gap between the packet in question and the previous one. It seemed like the most versatile solution and simple solution.

12.2 TCP option modification

The TCP-header consists of 5 words (20 bytes) of standard information followed by a maximum of 10 words (40 bytes) worth of optional information. The optional information (`options`) is a series of small collections of bytes, each containing different relevant information. Adding an extra option is something which is quite common, safe and controllable. We know the headers are parsed into memory and made accessible to the MEs in CTM, so it seems reasonable to have this as a first solution to the communication-challenge.

There may of course be better ways to handle the communication, but the important thing here is to just have a means of communication that could work and enable us to do start tackling the bigger challenge of programming the hardware itself.

The following paragraphs will give some details to how we have implemented this for anyone who wants to do the same.

TCP segment header

Offsets	Octet	0								1								2								3							
Octet	Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	0	Source port																Destination port																
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0				NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size														
16	128	Checksum																	Urgent pointer (if URG set)														
20	160	Options (if *data offset* > 5. Padded at the end with "0" bits if necessary.)																															
⋮	⋮																																
60	480																																

Figure 12.1: TCP header in full

The TCP-header information is added in tcp_output.c,[2] and specifically the `options` are added in the function tcp_options_write.

A simple solution for initial testing could be to hard-code a value into the write-function, but that would be very limiting to realizing a solution where L4 can dynamically adjust the rate according to state.

To enable functions on L4 to provide relevant information to the function in `tcp_output.c`, we need to create a callback function. The

[2]The names are hyperlinks to the respective files at bootlin.com for those who are interested

Linux source code makes this quite easy by adding the function to the tcp_congestion_ops-struct. By adding a callback function like this, we can have any TCP-algorithm make use of the new option in the header. We can make any kernel module use this function to add information of the same type to this option.

In our case we add a u32 value for the time-gap to be conveyed. This should provide enough space to be able to test a great variance in gaps.

The callback function and some identificators and sizes are added to `tcp.h`.

Modified tcp.h[3]:

```
#define TCPOPT_PACEOFFLOAD                200
#define TCPOLEN_PACEOFFLOAD               6
#define TCPOLEN_PACEOFFLOAD_ALIGNED       8

struct tcp_congestion_ops {
        ...
    u32 (*pace_offload)(struct tcp_sock *tp);
        ...
};
```

To add the function in L4, we add it to the list of functions in `tcp_congestion_ops` found in the L4 file, and make an implementation of it. In our case we have made a copy of CUBIC with only that modification.

tcp_cubic_paced.c:

```
static u32 bictcp_pace_offload(struct tcp_sock *tp){
        return tp->srtt_us / tp->snd_cwnd;
}

static struct tcp_congestion_ops cubictcp __read_mostly = {
        .init           = bictcp_init,
        .ssthresh       = bictcp_recalc_ssthresh,
        .cong_avoid     = bictcp_cong_avoid,
        .set_state      = bictcp_state,
        .undo_cwnd      = tcp_reno_undo_cwnd,
        .cwnd_event     = bictcp_cwnd_event,
        .pkts_acked     = bictcp_acked,
        .pace_offload   = bictcp_pace_offload,
        .owner          = THIS_MODULE,
        .name           = "cubic_paced",
};
```

Finally we need to actually add the option to the header run-time if needed. That is, if the callback is actually used. This is done in `tcp_output.c` after all other options are written to the header. Once that is done, we need to make sure that the size-calculation of the header takes the new option type into consideration.

[3]Reference to the whole file in my Github repoository

Modified tcp_output.c:

```
#define OPTION_PACE_OFFLOAD                (1 << 4)

static void tcp_options_write(__be32 *ptr, struct tcp_sock *tp,
                              struct tcp_out_options *opts){
        ...

        if (tp == NULL)
                return;

        if (unlikely(OPTION_PACE_OFFLOAD & options)){
                *ptr++ = htonl((TCPOPT_NOP << 24)
                | (TCPOPT_NOP << 16)
                | (TCPOPT_PACEOFFLOAD << 8)
                | TCPOLEN_PACEOFFLOAD);
                *ptr++ = inet_csk((struct sock *)tp)->
                icsk_ca_ops->pace_offload(tp);
        }
}


static unsigned int tcp_established_options(struct sock *sk,
        struct sk_buff *skb, struct tcp_out_options *opts,
        struct tcp_md5sig_key **md5){
        ...

        if (inet_csk(sk)->icsk_ca_ops->pace_offload){
                const unsigned int remaining =
                        MAX_TCP_OPTION_SPACE - size;
                if (remaining >= TCPOLEN_PACEOFFLOAD_ALIGNED){
                        size += TCPOLEN_PACEOFFLOAD_ALIGNED;
                        opts->options |= OPTION_PACE_OFFLOAD;
                }
        }
        ...
}
```

The complete implementation can be found in the repo of this thesis Paced Linux https://github.com/Permki/PacedLinux

12.2.1 Compiling and loading

Once all files are modified, they need to be added to a kernel. `tcp_output.c` and `tcp.h` need to be compiled into the kernel while the L4 code can be compiled and loaded into a running kernel using the `insmod`-function.

Before compiling the kernel it is important to note that all functionality of the Agilio NICs not necessarily runs on all kernel versions. In our case we were advised by Netronome to use a kernel version no higher than v4.x.[4] We opted for version 4.15 which was the newest supported kernel for Ubuntu 18 when we started working on it.

[4] See Tested Linux versions for Netronome software

The kernel version is important if all software from Netronome (RTE-service and so on) is wanted. It is not necessary just for using the upstream driver of the NIC.

When the kernel is compiled and installed with the modified kernel code, we can insert our custom L4 module (in our case the `tcp_cubic_paced.c` using `insmod`. With the module inserted, the congestion control algorithm needs to be selected by adding it to `/etc/sysctl.conf` and running `sysctl -p`.

At this point we have a functioning kernel which inserts the value for packet gaps run-time into the `options` of every TCP-header. We did this and verified it both by adding a kernel print inside `tcp_output.c` and by running `tcpdump` on a neighboring machine receiving data from our host.

All that is left at this point then is to enable TSO in the NIC and start working on the hardware implementation.

```
sudo ethtool -K [interface-name] tso on
```

The commands and aliases are all in the repository for this thesis.

12.3 Hardware implementation

Once we have a means of communication in place, we need to be able to parse this information out in the NIC and act upon it. All packets should be parsed automatically by the parser in the NIC, putting the headers in CTM for the FPCs. At that point we need to be able to parse out the option value for the delay and execute the delay.

12.3.1 Timing and sleep

Doing delays in P4 can be problematic. We can recirculate packets in some way, but it will be highly inefficient as it will fill up the queues and be quite demanding to the card to constantly be putting stuff back in the ingress queue just to have it happen again and again. It did not seem like a plausible solution as it would be highly inefficient and exhaust the card at some point.

There are sleep-functions available using C for P4, but we know we most likely will be needing a lot of customization. Because of that we moved over to working on a solution using MicroC. MicroC appears to be more straight forward with easily accessible functions for adding delay and more intricate parsing.

The `sleep`-function available in C for P4 is also available in MicroC. It can be found in `lib/nfp/_c/me.c`. This is available both in the MicroC-programs provided by Netronome, and in the NFP firmware. `sleep` sets an alarm and waits to be signaled. It is not the most efficient solution,[5] but it does do a delay.

[5] If we are to look a bit further than the scope of this thesis, it could seem inefficient

```
__intrinsic void
sleep(unsigned int cycles)
{
    unsigned int batch;
    SIGNAL sig;

    do {
        batch = (cycles > 0xfffff) ? 0xfffff : cycles;
        __implicit_write(&sig);
        set_alarm(batch, &sig);
        wait_for_all(&sig);
        cycles -= batch;
    } while (cycles);
}
```

With this functionality, we can add delays on packets. We wrote a simple program which parses packets, reads out the delay from the `options` and induces a delay before transmitting the packet.

Here is an extract of the most relevant part of the code:

```
int main(void){
  /* The packet header received by the thread */
  struct pkt_rxed pkt_rxed;
  /* The packet in the CTM */
  __mem40 struct pkt_hdr *pkt_hdr;

  for (;;) {
    pkt_hdr = receive_packet(&pkt_rxed, sizeof(pkt_rxed));
    if (pkt_hdr->tcp_hdr.off > MIN_TCP_HEADER_LEN) {
        parse_tcp_options(pkt_hdr);
        send_packet(&pkt_rxed.nbi_meta, pkt_hdr);
    }
  }
  return 0;
}

void parse_tcp_options(__mem40 struct pkt_hdr *pkt_hdr){
  __mem40 char *opt_ptr = pkt_hdr->tcp_options;
  int length = (pkt_hdr->tcp_hdr.off
    - MIN_TCP_HEADER_LEN)
    * WORDSIZE;

  while (length > 0) {
    int opcode = *opt_ptr++;
    int opsize;

    switch (opcode) {
        case TCPOPT_EOL:
            return;
        /* Ref: RFC 793 section 3.1 */
        case TCPOPT_NOP:
            length--;
            continue;
```

to have a lot of threads spinning like this, waiting to continue. With multiple flows, this could create a bottleneck in the NIC. A better solution would be to have more centralized delay on a queue or dispatcher for a specific flow. More on this in the sections to come.

```
        default:
            if (length < 2)
                return;
            opsize = *opt_ptr++;
                /* "silly options" */
            if (opsize < 2)
                return;
            /* don't parse partial options */
            if (opsize > length)
                return;
            if (opcode ==
                TCPOPT_PACEOFFLOAD){
                opt_ptr += 2;
                timing_loop(*opt_ptr);
                return;
            }
            opt_ptr += opsize-2;
            length -= opsize;
    }
  }
}

void timing_loop(long microsecs){
  long period = microsecs*633 ;
  while (period > MAX_SLEEP_TIME){
    sleep(MAX_SLEEP_TIME);
    period -= MAX_SLEEP_TIME;
  }
  sleep(period);
}
```

When testing this, we connected a machine directly to our SmartNIC and manually sent single packets to the NIC. The program receives a packet, sleeps and returns the packet back.[6]

12.4 Testing and workarounds

The first version of the program simply did a 500ms delay before returning the packet. With tcpdump and Wireshark on the connected machine, we could see that the packet was delayed when our modified code was set up to sleep. When we removed the sleep, we could see that the gap disappeared. In other words, we know that the `sleep`-functionality found in `me.c` works as wanted. We then experimented on sending packets using a Python-program with options modified, having the MicroC-program parse the options and do delay based on the value in the option. It worked as expected, which made us optimistic about making this work.

We then had to delve into how this could be used when we did not have access to the NIC from our host machine. The test itself was a workaround due to not having access to the NIC host-side.

We discussed some other options to see if we could work around the issue of having to use the NIC remotely. One could have one machine

[6]The rest of the code is available in the repository of this thesis.

send bursts of packets to the running SmartNIC much like we did with the Python program but ramping it up to create bursts. The SmartNIC could then pace and return the packets to the connected machine.

If the SmartNIC was successful in pacing out the packets, it could be measured at the connected system. The issue with this thought experiment is that we get a lot of unknowns. We do not know what egress rate the connected system has, and we lose control over the segments.

As will be clear in the next section, we cannot just delay packets a specified amount of time. We need to take the order into consideration. If we have 10 packets in one segment, we will have to delay the last packet 9 times longer than the second packet if we have an even distribution. That information needs to be conveyed. If we simply write it to the `option` when the TSO-segment is created, we still have the issue of the packets arriving at varying times at the SmartNIC because it is on a different macine.

We would need to ensure some ordering, and we would need to know which segments the packets belong to. The segments themselves would be theoretical. And the higher the speeds, the longer the distance and the more saturated the path is, the more this issue affects the experiment. We lose control of the path. On top of that, we know that the packet flow for host-to-network and network-to-host is not the same.

12.5 One queue to rule them all

The considerations concerning the actual spacing is what remains as the second big challenge with implementing pacing on these NICs. The concurrent processing of the threads must be taken into account.

If we wish to space out packets with inter-packet gaps, we need to manage the relative gap between the packets in a TSO-segment. Simply making a thread spin for a certain amount of time will not do. Imagine 42 packets arriving the NIC as a TSO-segment. The segment gets split into individual packets before being dispatched to the MEs CTM. We risk several threads picking up packets at the same time. They will all spin (or yield waiting for a signal depending on the final solution) until the requested delay-time has passed. The packets are then forwarded to the egress queue basically at the same time, causing them to enter the network without the inter-packet delay.

Doing this will just create a delay on the TSO-segment as a whole or in part, not the individual packets. We need a way to make sure that the packets are delayed relative to the previous packet in the segment. There are a couple of ways we can approach this challenge.

We either stamp each packet with a delay relative to the first packet in the segment when it is created on L4, or we create our own queue system on an ME responsible for managing how much delay each packet has relative to a start. This can be combined with the first solution.

Yet another solution could be to calculate the accumulative delay

when the TSO-splitting is happening and pass it on to the MEs assigned.

12.5.1 Queue management

All the viable solutions mentioned require a lot more than just adding a sleep when a thread is processing a packet based on a fixed value sent down from L4. We need some management in the form of a queue manager responsible of organizing the delay being executed by the threads. The queue manager can perhaps be as simple as the ME doing the TSO splitting, or it could be a designated queue manager for several MEs or threads.

This and the challenge of our missing interfaces is the focus of the last chapter of this thesis. We ended up realizing that our intended use of the NICs was not compatible with how the abstractions were designed, but in the following we will propose a path to pursue to make the pacing happen. We hope our thoughts and ideas will be of help.

Chapter 13

Outlining future work with the CoreNIC

13.1 A solution without abstractions

The abstractions provided by Netronome have proved to be designed to fit a very different need than ours. Unless Netronome changes the compilers to enable us to compile custom code into the existing firmware code, we are at a loss with solving this problem using either P4 or the vanilla MicroC approach.

The positive discovery thus far is that we now know that it is possible to do timing on the cards, and we know that there are a lot of possibilities when it comes to inter-thread and -core communication with both global, semi-global and local memory sharing.

Another positive is that the functioning upstream firmware can be downloaded and installed at will, and we have some ideas as to how a working solution could be. We want to retain the existing firmware and add on some customized handling or packets with the pace-option enabled.

13.1.1 Firmware modifiction

As we have access to the upstream firmware and know the NICs are capable of timing, we propose that future research can focus on trying to modify the standard firmware. The firmware code is available and modifiable, and it can be compiled and installed just like any other application. By modifying the firmware itself, rather than creating a whole new one, we can keep the interfaces and have the cards do all the things they usually do, and just create a special handling for packets with the pace-option enabled.

We have done some initial attempts at modifications and started exploring the code, and the following sections will discuss some of our findings.

13.2 CoreNIC

The Netronome NIC firmware implementation for the Agilio SmartNiCs (nicknamed CoreNIC) is available through the Netronome Repository (https://github.com/Netronome/nic-firmware).

The repository has extensive information about the cards, the firmware and how to compile the code. Commands and aliases for easily compiling and installing the firmware can also be found in the repository for this thesis.

13.3 TSO split

As we know that the TSO-segments need to be split by the NIC before any processing is done, we considered this to be the best place to start looking.

The code for the splitting can be found in `blocks/vnic/pci_in/issue_dma.c`. The more generic name LSO is used, but that is where it is processed. Initially we believed we could simply add a delay in the loop that splits the packets, but that will not work, for two reasons.

Firstly, it seems like this function parses the packets and puts them all into a queue before finally signaling the queue handler(s) that the split is done.

Secondly, we have 8 threads working here, and we need to make sure they all have the same baseline for the clock. If we only have one thread it is unproblematic to have it wait for a specified amount of time between the packets, but when we have several threads handling different packets, we must make sure that they all use the same clock and delay with a delay calculated for the specific packet from time 0.

We tested it and found that we could stall the whole process by adding an infinite loop, but we could not delay on a packet by packet basis by simply sleeping for a specified amount of time.

Making the split-function do the work could be a potential solution, but it would require adding an option-parser and change the flow so that packets can be signaled for transmit when needed.

A challenge needed to be taken into consideration when rewriting this code is that the compiler is strict about how many instructions the resulting code will have. Some efficiency is needed in how the code is performed. A lot of instructions can be saved by e.g. avoiding unnecessary nesting or assignments.

13.3.1 Clocking

If this is to work, one would need to figure out how the queues are managed and how this code is executed. One TSO-segment seems to be handled and split by one ME. Looking at the code, it seems that we need to take into consideration that each thread is unaware of which

number in the line the packet is. As the code is shared among all threads of the ME, it would be necessary to make sure there is some shared information about time.

Having an absolute delay calculated at L4 rather than a relative delay, could seem helpful as each thread would not need to know which packet in the segment it is handling. But all the threads handling the same segment need have access to the same clock with some starting point.

Either we make sure that only one thread works on a segment (which seems unreasonable), or we need some centralized clock or queue manager.

Adding a shared variable for an ME should be unproblematic given the opportunities we have for both local and non-local variables. It would seem viable to have the code issue a variable shared by the threads of the ME holding e.g. the timestamp of the start of the TSO segment. Each thread would then issue a sleep waiting for the ME clock to reach the desired value.

In `me.c` where we find `sleep()`, we also find `me_tsc_read()` which seems to read off the current time from `local_csr_timestamp_low` and `local_csr_timestamp_hihg`. This would likely be the information needed to read time shared among the threads, and the threads could run a loop on `sleep` checking the value of the clock every given interval of time.[1]

This solution seems very plausible as long as one is able to find out where it can be done and how to make sure that each packet gets put up for transmit when needed. It also has the advantage of all the threads sharing the same clock as they belong to the same ME.[2]

13.4 Queue management

Doing work inside the TSO split is not necessarily the best solution. The splitting is not just a simple for-loop. It is close to 800 lines of code, doing a lot of different things to the packets being processed. Adding a delay to this processing seems easiest, but it would cause a lot of stalling, and it would be necessary to be absolutely sure when to perform the delay in the loop (if possible), and make sure that other parts of the flow would not be affected by it.

As the actual code splitting up the packet is complex it could be wise to look into what happens after the split is done. The packets are split up and most likely put into a queue ready for further processing or transmit. By looking into this part of the flow, one could have the advantage of doing the delay at a stage where all other processing is done.

[1] It is not possible for one thread to set a timer at a specific time. It will need to sleep a defined amount of time and then check the clock, repeating this in a loop until the needed amount of time has elapsed.

[2] Assuming that one TSO-segment is handled by one ME and not several.

To make this work, one could either look at the queue used today or create a new queue for the packets to be delayed. We know we will need some sort of queue management in the form of the threads signaling packets for transmit at specific times. Having a simple priority queue using a timestamp relative to a global clock could be a good solution. Having one queue manager handling the queue and ordering the packets based on timestamps would be how we would solve this in higher level programming.

Each thread splitting up packets could add on metadata to the packet in the form of a timestamp relative to the clock used by the queue-administrator. By retrieving the timestamp from the queue-managing-ME, all MEs splitting segments could calculate a timestamp by adding on the delay found in the options. We would then need to have the delay-value in the packet option be an absolute delay set in L4, not a relative one.

Once the timestamp for departure time has been added to the packet, the packet can be given to the queue manager for further handling.

One big advantage with this solution is that it would be efficient if we have multiple flows. The FPCs would finish in much the same time as usual. The only delay would be in the priority queue, and as long as the queue is based only on timestamp, we could have multiple flows in the same queue. The queue manager wold simply handle the ordering and signaling for transmit when needed.

13.4.1 Buffer capacity and management

One of the challenges by adding delay and have packets wait in queue is that we start hogging resources. Rather than transmitting the whole segment at once, we make the NIC hold on to it while waiting for a clock.

Depending on capacity one could be forced to have one queue per segment, and one could get issues with capacity either way, but it also depends on how the queue is constructed. The queue manager can be implemented using only header pointers and a timestamp. The headers themselves would need to be moved out of the ME CTM if we are to be able to process more packets while waiting. If we can not move the headers to some other memory, we would risk running out of threads quite fast as all threads would have to wait for their packet to be picked for transmit by the queue manager.

Given all the different areas of memory on these cards, it would be pessimistic to assume that we could not make it work at least for a limited scope. Either way all of these thoughts are mostly theoretical reflections on the matter. They all need to be rooted in actual code, and there is more work to be done to do that.

Chapter 14

Final reflections

Employing pacing to improve network performance is something which is very relevant today, and so is the use of TSO.

For high speed networks, turning off TSO can simply be too costly or too inefficient to achieve the desired performance. At the same time, it is getting increasingly more relevant to let L4 control the actual transmission rate from the NICs. At higher speeds even having a clock with enough granularity is difficult without the help of specialized hardware.

All of these considerations have been mapped out, and we have argued that a possible solution to the apparent dichotomy of these two aspects of networking may be found with pacing in hardware.

14.1 Netronome Agilio abstractions

The main intention of this thesis has been to investigate whether it could be possible to implement pacing in hardware using Netronome Agilio SmartNICs with the abstractions provided by Netronome.

We have concluded that this is not possible simply by using the provided abstractions the way they are set up today. If Netronome at some point makes it possible to make the custom code coexist with the upstream firmware, it would be highly relevant to explore further capabilities both for P4 and MicroC. Until then we believe it is best to explore other options with these cards.

14.2 CoreNIC

Even though we did not find a clear cut way to pacing in hardware using P4 or MicroC, we have been positively surprised by the sheer capabilities of these cards and have high beliefs in the possibility of making pacing work by doing more research into the CoreNIC firmware.

A substantial amount of work remains to make it work, but given the abilities and organization of these cards, it seems highly plausible to us that a working solution is obtainable and is worth looking into.

14.3 The future of pacing

As we see it, making TSO harmonize with pacing seems important to try to achieve. TSO is here to stay, and having L4 lose control of the actual transmission rate seems like a loss to all present and future TCP algorithms. If future research into the firmware is successful, who knows what more can be done. Perhaps even more functionality could be offloaded from the transport layer.

We at least hope someone will continue to look into it. Hopefully this thesis and its resources can be of help to anyone attempting to make it work.

Acronyms

ACK Acknowledgement. *Glossary:* ACK, 9, 10, 12, 13, 14, 16, 17, 18, 19, 20, 22, 30, 37, 39, 40, 41, 43, 44, 45, 46, 47, 52, 53, 58, 59

ACK compression ACK compression. *Glossary:* ACK compression

AQM Active Queue Management. *Glossary:* AQM, 25, 26, 27, 28, 29, 30, 32, 34, 39, 44, 50, 53, 56

BBR Bottleneck Bandwidth and Round-trip propagation time. *Glossary:* BBR, 35, 36, 37, 38, 45, 50, 51, 54

BDP Bandwidth Delay Product. *Glossary:* BDP, 15, 16, 22, 25, 27, 28, 31, 36, 37, 42, 43, 46, 49, 50, 55

BIC TCP BIC. *Glossary:* BIC, 17, 20, 21, 22, 23

BtlBw Bottleneck Bandwidth. *Glossary:* BtlBw, 14, 15, 36, 37, 38, 43

CoDel Controlled Delay. *Glossary:* CoDel, 26, 27, 28, 29, 37

CUBIC TCP Cubic. *Glossary:* CUBIC, 16, 17, 22, 23, 37, 38, 54, 84

`cwnd` Congestion Window. *Glossary:* cwnd, 13, 14, 18, 19, 20, 21, 22, 32, 36, 37, 40, 41, 42, 43, 44, 45, 46, 47, 49, 50, 51, 53, 81, 82

DCTCP Data Center TCP. *Glossary:* DCTCP, 30, 31, 32, 33, 38, 42, 47, 50, 53

DUPACK Duplicate ACK. *Glossary:* DUPACK, 9, 14, 18, 19, 23, 30, 34

ECN Explicit Congestion Notification. *Glossary:* ECN, 26, 29, 30, 31, 34, 38, 42, 44, 47, 49, 50

FPC Flow Processing Core. *Glossary:* FPC, 66, 67, 68, 71, 72, 94

FRR Fast Retransmit and Fast Recovery. *Glossary:* FRR, 19

GRO Genereic Receive Offload. *Glossary:* GRO, 60

GSO Generic Segmentation Offload. *Glossary:* GSO, 59, 81

IP Internet Protocol. *Glossary:* IP

LRO Large Receive Offload. *Glossary:* LRO, 59, 60

LSO Large Segment Offload. *Glossary:* LSO, 33, 40, 59, 92

LSS Limited Slow-Start for TCP. *Glossary:* LSS, 17, 46

MTU Maximum Transmission Unit. *Glossary:* MTU, 2, 29, 41, 46, 56, 58, 60

NIC Network Interface Controller. *Glossary:* NIC, 2, 3, 41, 56, 57, 58, 59, 60, 61, 62, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 78, 79, 81, 82, 85, 86, 87, 88, 89, 90, 91, 92, 94, 95

PaC Paced Chirping. *Glossary:* PaC, 22

PPC Packet Processing Core. *Glossary:* PPC, 66, 68, 71, 72

PRR Proportional Rate Reduction. *Glossary:* PRR, 48

RED Random Early Detection. *Glossary:* RED, 25, 26, 27, 28, 29, 37

RSS Receive Side Scaling. *Glossary:* RSS, 59

RTO Retransmission Time-Out. *Glossary:* RTO, 13, 14, 18, 19, 23, 34, 42

RTprop Round-Trip propagation time. *Glossary:* RTprop, 36, 37

RTT Round-trip time. *Glossary:* RTT, 13, 14, 15, 16, 17, 20, 21, 22, 28, 31, 32, 34, 35, 36, 37, 40, 41, 45, 52, 53, 54, 55, 81, 82

SACK Selective Acknowledgement. *Glossary:* SACK, 9, 16, 20, 41

`ssthresh` Slow Start Threshold. *Glossary:* ssthresh, 13, 14, 18, 19, 40

TCP Transmission Control Protocol. *Glossary:* TCP, 1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15, 19, 20, 22, 23, 27, 28, 31, 32, 34, 37, 39, 40, 42, 43, 45, 46, 48, 51, 52, 54, 56, 58, 59, 60, 61, 62, 64, 82, 83, 84, 86, 96

TOE TCP stack Offload Engine. *Glossary:* TOE

TSO TCP Segmentation Offloading. *Glossary:* TSO, 2, 3, 41, 46, 47, 59, 60, 61, 62, 65, 81, 82, 86, 89, 90, 92, 93, 95, 96

UDP User Datagram Protocol. *Glossary:* UDP, 6, 7, 40

Vegas TCP Vegas. *Glossary:* Vegas, 34, 35, 51

Bibliography

[1] A. Aggarwal, S. Savage, and T. Anderson. "Understanding the performance of TCP pacing". en. In: *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)*. Vol. 3. Tel Aviv, Israel: IEEE, 2000, pp. 1157–1165. ISBN: 978-0-7803-5880-5. DOI: 10.1109/INFCOM.2000.832483. URL: http://ieeexplore.ieee.org/document/832483/ (visited on 02/24/2021).

[2] Mudassar Ahmad, Asri Ngadi, and Mohd Murtadha Mohamad. "EXPERIMENTAL EVALUATION OF TCP CONGESTION CONTORL MECHANISMS IN SHORT AND LONG DISTANCE NETWORKS". en. In: . *Vol.* (2005), p. 15.

[3] Mohammad Alizadeh et al. "Data center TCP (DCTCP)". In: *Proceedings of the ACM SIGCOMM 2010 conference*. SIGCOMM '10. New York, NY, USA: Association for Computing Machinery, Aug. 2010, pp. 63–74. ISBN: 978-1-4503-0201-2. DOI: 10.1145/1851182.1851192. URL: https://doi.org/10.1145/1851182.1851192 (visited on 03/09/2021).

[4] M. Allman, V. Paxson, and E. Blanton. *TCP Congestion Control*. en. Tech. rep. RFC5681. RFC Editor, Sept. 2009, RFC5681. DOI: 10.17487/rfc5681. URL: https://www.rfc-editor.org/info/rfc5681 (visited on 04/17/2021).

[5] Mark Allman and Ethan Blanton. "Notes on burst mitigation for transport protocols". en. In: *ACM SIGCOMM Computer Communication Review* 35.2 (Apr. 2005), pp. 53–60. ISSN: 0146-4833. DOI: 10.1145/1064413.1064419. URL: https://dl.acm.org/doi/10.1145/1064413.1064419 (visited on 03/17/2021).

[6] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. "Sizing router buffers". en. In: *ACM SIGCOMM Computer Communication Review* 34.4 (Aug. 2004), pp. 281–292. ISSN: 0146-4833. DOI: 10.1145/1030194.1015499. URL: https://dl.acm.org/doi/10.1145/1030194.1015499 (visited on 04/06/2021).

[7] P. Balasubramanian. *HyStart++: Modified Slow Start for TCP*. en. 2019. URL: http://www.watersprings.org/pub/id/

draft-balasubramanian-tcpm-hystartplusplus-01.html (visited on 04/04/2021).

[8] Neda Beheshti, Yashar Ganjali, and Monia Ghobadi. "Experimental Study of Router Buffer Sizing £ Ý". en. In: (), p. 14.

[9] Christian Benvenuti. *Understanding Linux network internals*. OCLC: ocm63674509. Sebastapol, CA: O'Reilly, 2006. ISBN: 978-0-596-00255-8.

[10] Ethan Blanton and Mark Allman. "On the Impact of Bursting on TCP Performance". en. In: *Passive and Active Network Measurement*. Ed. by David Hutchison et al. Vol. 3431. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 1–12. ISBN: 978-3-540-25520-8 978-3-540-31966-5. DOI: 10.1007/978-3-540-31966-5_1. URL: http://link.springer.com/10.1007/978-3-540-31966-5_1 (visited on 03/15/2021).

[11] Rune Johan Borgli and Joakim Misund. "Comparing BBR and CUBIC Congestion Controls". en. In: (), p. 4.

[12] Pat Bosshart et al. "P4: programming protocol-independent packet processors". en. In: *ACM SIGCOMM Computer Communication Review* 44.3 (July 2014), pp. 87–95. ISSN: 0146-4833. DOI: 10.1145/2656877.2656890. URL: https://dl.acm.org/doi/10.1145/2656877.2656890 (visited on 11/05/2022).

[13] R. Braden. *Requirements for Internet Hosts - Communication Layers*. en. Tech. rep. RFC1122. RFC Editor, Oct. 1989, RFC1122. DOI: 10.17487/rfc1122. URL: https://www.rfc-editor.org/info/rfc1122 (visited on 04/01/2021).

[14] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. "TCP Vegas: new techniques for congestion detection and avoidance". en. In: *Proceedings of the conference on Communications architectures, protocols and applications - SIGCOMM '94*. London, United Kingdom: ACM Press, 1994, pp. 24–35. ISBN: 978-0-89791-682-0. DOI: 10.1145/190314.190317. URL: http://portal.acm.org/citation.cfm?doid=190314.190317 (visited on 03/22/2021).

[15] Neal Cardwell et al. "BBR Congestion-Based Congestion Control". en. In: *ACM Queue* 14, September-October (2016), pp. 20–53. URL: http://queue.acm.org/detail.cfm?id=3022184.

[16] Wu-chang Feng et al. "The BLUE active queue management algorithms". en. In: *IEEE/ACM Transactions on Networking* 10.4 (Aug. 2002), pp. 513–528. ISSN: 1063-6692. DOI: 10.1109/TNET.2002.801399. URL: http://ieeexplore.ieee.org/document/1026008/ (visited on 04/10/2021).

[17] Cheng-Yuan Ho, Yi-Cheng Chan, and Yaw-Chung Chen. “An Enhanced Slow-Start Mechanism for TCP Vegas”. In: *11th International Conference on Parallel and Distributed Systems (ICPADS'05)*. Vol. 1. Fukuoka, Japan: IEEE, 2005, pp. 405–411. ISBN: 978-0-7695-2281-4. DOI: 10.1109/ICPADS.2005.86. URL: http://ieeexplore.ieee.org/document/1531157/ (visited on 03/22/2021).

[18] David X. Wei, Pei Cao, and Steven H. Low. “TCP Pacing Revisited”. In: *IEEE INFOCOM* ().

[19] S. Floyd and V. Jacobson. “Random early detection gateways for congestion avoidance”. In: *IEEE/ACM Transactions on Networking* 1.4 (Aug. 1993), pp. 397–413. ISSN: 10636692. DOI: 10.1109/90.251892. URL: http://ieeexplore.ieee.org/document/251892/ (visited on 04/09/2021).

[20] Monia Ghobadi and Yashar Ganjali. “TCP Pacing in Data Center Networks”. en. In: *2013 IEEE 21st Annual Symposium on High-Performance Interconnects*. San Jose, CA, USA: IEEE, Aug. 2013, pp. 25–32. ISBN: 978-0-7695-5103-6. DOI: 10.1109/HOTI.2013.18. URL: http://ieeexplore.ieee.org/document/6627732/ (visited on 02/24/2021).

[21] Sangtae Ha and Injong Rhee. “Taming the elephants: New TCP slow start”. en. In: *Computer Networks* 55.9 (June 2011), pp. 2092–2110. ISSN: 13891286. DOI: 10.1016/j.comnet.2011.01.014. URL: https://linkinghub.elsevier.com/retrieve/pii/S1389128611000363 (visited on 04/03/2021).

[22] Y. Sinan Hanay, Abhishek Dwaraki, and Tilman Wolf. “High-performance implementation of in-network traffic pacing”. en. In: *2011 IEEE 12th International Conference on High Performance Switching and Routing*. Cartagena, Spain: IEEE, July 2011, pp. 9–15. ISBN: 978-1-4244-8454-6. DOI: 10.1109/HPSR.2011.5985997. URL: http://ieeexplore.ieee.org/document/5985997/ (visited on 03/01/2021).

[23] T. Hoeiland-Joergensen et al. *The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm*. en. Tech. rep. RFC8290. RFC Editor, Jan. 2018, RFC8290. DOI: 10.17487/RFC8290. URL: https://www.rfc-editor.org/info/rfc8290 (visited on 04/11/2021).

[24] Jacbson, Van and Karels, Michael J. “Congestion Avoidance and Control”. en. In: *Proceedings of SIGCOMM '88* (Aug. 1988).

[25] Hao Jiang and Constantinos Dovrolis. “Source-Level IP Packet Bursts: Causes and Effects”. en. In: (), p. 6.

[26] Mirja Kuhlewind et al. “Using data center TCP (DCTCP) in the Internet”. en. In: *2014 IEEE Globecom Workshops (GC Wkshps)*. Austin, TX, USA: IEEE, Dec. 2014, pp. 583–588. ISBN: 978-1-4799-7470-2. DOI: 10.1109/GLOCOMW.2014.7063495. URL: http://ieeexplore.ieee.org/document/7063495/ (visited on 04/06/2021).

[27] Lisong Xu, K. Harfoush, and Injong Rhee. “Binary increase congestion control (BIC) for fast long-distance networks”. In: *IEEE INFOCOM 2004*. Vol. 4. Hong Kong, China: IEEE, 2004, pp. 2514–2524. ISBN: 978-0-7803-8355-5. DOI: 10.1109/INFCOM.2004.1354672. URL: http://ieeexplore.ieee.org/document/1354672/ (visited on 04/02/2021).

[28] Joakim Misund and Bob Briscoe. “Paced Chirping: Rapid flow start with very low queuing delay”. en. In: *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. Paris, France: IEEE, Apr. 2019, pp. 798–804. ISBN: 978-1-72811-878-9. DOI: 10.1109/INFCOMW.2019.8845072. URL: https://ieeexplore.ieee.org/document/8845072/ (visited on 02/19/2021).

[29] NFP Netronome. “Netronome Network Flow Processor 6xxx Network Flow C Compiler User’s Guide”. English. In: (2018). URL: https://github.com/Permki/PacedLinux/blob/main/Manuals/Firmware%20and%20FlowProcessor/UG_nfp6000_nfcc.pdf.

[30] NFP Netronome. “NFP-4000 Theory of Operation”. In: URL: https://www.netronome.com/static/app/img/products/silicon-solutions/WP_NFP4000_TOO.pdf.

[31] NFP Netronome. *Product Brief Netronome NFP-4000 Flow Processor*. Tech. rep. URL: https://www.netronome.com/media/documents/PB_NFP-4000-7-20.pdf.

[32] Kathleen Nichols and Van Jacobson. “Controlling Queue Delay: A modern AQM is just one piece of the solution to bufferbloat.” en. In: *Queue* 10.5 (May 2012). Citation Key=codel, pp. 20–34. ISSN: 1542-7730, 1542-7749. DOI: 10.1145/2208917.2209336. URL: https://dl.acm.org/doi/10.1145/2208917.2209336 (visited on 04/10/2021).

[33] Edwin Peer. *Mapping P4 to SmartNICs*. May 2017. URL: https://opennetworking.org/wp-content/uploads/2020/12/p4_d2_2017_nfp_architecture.pdf.

[34] Salvatore Pontarelli, Giuseppe Bianchi, and Michael Welzl. “A Programmable Hardware Calendar for High Resolution Pacing”. en. In: *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. Bucharest, Romania: IEEE, June 2018, pp. 1–6. ISBN: 978-1-5386-7801-5. DOI: 10.

1109/HPSR.2018.8850731. URL: https://ieeexplore.ieee.org/document/8850731/ (visited on 02/19/2021).

[35] K. Ramakrishnan, S. Floyd, and D. Black. *The Addition of Explicit Congestion Notification (ECN) to IP*. en. Tech. rep. RFC3168. RFC Editor, Sept. 2001, RFC3168. DOI: 10.17487/rfc3168. URL: https://www.rfc-editor.org/info/rfc3168 (visited on 04/09/2021).

[36] Danfeng Shan and Fengyuan Ren. "Improving ECN marking scheme with micro-burst traffic in data center networks". en. In: *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. Atlanta, GA, USA: IEEE, May 2017, pp. 1–9. ISBN: 978-1-5090-5336-0. DOI: 10.1109/INFOCOM.2017.8057181. URL: http://ieeexplore.ieee.org/document/8057181/ (visited on 03/09/2021).

[37] M. Shreedhar and George Varghese. "Efficient fair queueing using deficit round robin". en. In: *ACM SIGCOMM Computer Communication Review* 25.4 (Oct. 1995), pp. 231–242. ISSN: 0146-4833. DOI: 10.1145/217391.217453. URL: https://dl.acm.org/doi/10.1145/217391.217453 (visited on 04/11/2021).

[38] Ryousei Takano et al. "Design and Evaluation of Precise Software Pacing Mechanisms for Fast Long-Distance Networks". en. In: *Proceedings of PFLDNet* (2005), p. 7.

[39] Van Jacobson. *A rant on queues*. 2006. URL: //www.pollere.net/Pdfdocs/QrantJul06.pdf.

[40] Curtis Villamizar and Cheng Song. "High performance TCP in ANSNET". en. In: *ACM SIGCOMM Computer Communication Review* 24.5 (Oct. 1994), pp. 45–60. ISSN: 0146-4833. DOI: 10.1145/205511.205520. URL: https://dl.acm.org/doi/10.1145/205511.205520 (visited on 04/06/2021).

[41] Greg White, Koen De Schepper, and Bob Briscoe. *Low Latency, Low Loss, Scalable Throughput (L4S) Internet Service: Architecture*. en. URL: https://tools.ietf.org/html/draft-ietf-tsvwg-l4s-arch-03 (visited on 03/16/2021).

[42] Peng Yang et al. "TCP Congestion Avoidance Algorithm Identification". In: *IEEE/ACM Transactions on Networking* 22.4 (Aug. 2014), pp. 1311–1324. ISSN: 1063-6692, 1558-2566. DOI: 10.1109/TNET.2013.2278271. URL: http://ieeexplore.ieee.org/document/6594906/ (visited on 04/01/2021).

[43] Lixia Zhang, Scott Shenker, and David D Clark. "Observations on the Dynamics of a Congestion Control Alorithm The effects of two-way traffic". en. In: *Proceedings of the ACM SIGCOMM 1991 Conference on CommunicationsArchitectures and Protocols* (), pp. 133–147.

[44] Songyang Zhang. “An Evaluation of BBR and its variants”. In: *arXiv:1909.03673 [cs]* (Sept. 2019). arXiv: 1909.03673. URL: http://arxiv.org/abs/1909.03673 (visited on 03/16/2021).