UiO : Department of Informatics
University of Oslo

Fighting fire with fire: Eliminating standing queues with large UDP packet floods

Håkon Botnmark Jahre

Master's Thesis Spring 2015

Abstract

Excessive buffering in network equipment, sometimes called ‘buffer bloat’, can create very large delays for packets traversing the Internet. This degrades the user experience, especially when communication is latency critical, for example with interactive multimedia applications (Voice-over-IP etc).

This problem has recently become addressed via Active Queue Management mechanisms – however, such mechanisms need to be installed in the device where the problem occurs. Sometimes, large uncontrolled buffers cause delay in a device that is far away from the user who suffers from the problem, and therefore not under control of this user. This thesis presents a simple software-only solution that can at least partially improve the situation remotely: a ‘queue flusher’ that monitors the round-trip time of packets and, upon seeing a significant and sustained growth in delay, floods the network with a large bulk of packets that are sent in the direction where the problem is suspected. Evaluation results from a simple emulated network show that this mechanism can work surprisingly well, with a pronounced improvement in average delay at the expense of only a limited reduction in throughput.

Contents

List of Figures

List of Tables

Listings

Preface

It has been a lot of work to reach the finished thesis you are reading now. During the item I have been working with this thesis, I have had the chance to learn about a theme which I previously only had some knowledge of. Before starting this project, I did not know about the inner workings of active queue managers. Now, after working on the 'standing queue' problem, I have got a new understanding of how traffic in the internet works.

I would like to thank my friends and family for the support I have received during my master thesis. A special thanks to my supervisor is in order, as he has helped a great deal during my time at the university.

It has been a long road, and finally I am about to reach my destination.

Part I

Background

Chapter 1

Introduction

1.1 Motivation

For years, internet providers have pushed higher bandwidths to their customers. Higher bandwidth is only necessary for the power user who really have use for more bandwidth. A family where everyone is streaming video have a higher demand for bandwidth. For the average customer who spends most of their time browsing the web, however, a few megabits of bandwidth can be more than enough. In those cases, it is often not the bandwidth but the network delay that is the main reason for a 'slow internet'.

There are several reasons for the latency in a network. First, every device has a processing delay, which is the time it takes for the device to process a piece of data. This is often so low that it is almost nonexistent. At least non-detectable for humans. The other cause is the time it takes for the data to propagate along a link, which is near constant as it is dictated by the physical medium the link is made of. The third reason, and the one it is possible to do something about, is the size of the buffers in the internet. As each device has a buffer, which is a temporary storage for data before it is transferred to the next machine, data are queued. The buffer size depends on the amount of memory available in the device. As memory prices have declined, the amount of memory in the devices has increased [14].

The role of a buffer is the ability to absorb bursts of data, thus a buffer needs to be large enough. Big buffers (also called bloated buffers) however, due to the way most internet devices are set up, induce a large delay when filled. All data segments passing the buffer have to wait in order to be forwarded. The incoming data segment gets an additional delay added, namely the time every packet in front of it needs to get transferred. This delay may easily be several times larger than the original network latency.

As buffers full of data induce more latency, this is often the reason of an impression of a slow internet connection. As every connection may get the same delay, it is more noticeable on short connections than longer file transfers. If a file transfer take a second more to finish it is perhaps no big deal, but if every web page opened get an additional second in delay, it is a big deal.

As described in section 3.6, there are solutions to the problem of bloated buffers and the amount of data queued in them. The solutions need to be implemented and activated on every intermediate device, thus it may be difficult to do quickly even though the algorithms are thoroughly tested and provide reasonable results.

A possible solution could be to let the infrastructure in the internet be as it is and try to fix the buffer problems in another way. One other way that is presented in this thesis is a 'queue flusher', an application that monitors network traffic on a machine and tries to 'flush' the queue if a queue discovered. Because the flusher may reside on an arbitrary machine, it does not have direct control over the buffers as the other solutions have (as discussed in section 3.6); due to this, it treats the network as a 'black box'. It knows the input and output, because it is monitoring the data traffic to and from the device it is running on, but not what is going on inside. As long it is capable of monitoring input and output it should be able to do an approximation and hopefully get a better flowing network.

1.2 Structure

This thesis consists of 3 main parts; Background, Queue Flusher and Results.

The background section presents the necessary information to give the reader an overview of previous works and facts that are needed in order to fully understand why the 'queue flusher' works as it does.

This part describes how different algorithms used for data transfer work, the effects of large temporary storage and some solutions to the problem described in section 1.1.

The 'Queue Flusher' part discusses and describes the 'queue flusher'.

While the last part presents the environment in which the 'queue flusher' was run, and presents and discusses some test results. The environment is a homogeneous network where every single variable are set in order to get understandable results. In addition, the main results in presented and discussed.

1.3 Layers

This section contains a brief, abstract overview of how the network stack in an internet connected device works. The layers are non-transparent, meaning that an application sending data has to trust the following layers to forward the data to the correct recipient.

The layering of the stack also makes it easier to visualize the flow of data sent from an application to another.

1.3.1 Link layer

The link layer is the lowest level in the stack. It is responsible for transferring data to and from the physical network link. Data sent through

Table 1.1: IPv4 header format [26, 41]

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version				IHL				DSCP						ECN		Total Length															
4	32	Identification																Flags			Fragment Offset												
8	64	Time To Live								Protocol								Header Checksum															
12	96	Source IP Address																															
16	128	Destination IP Address																															
20	160	Options (if IHL > 5)																															

the link layer get a header (with information of the next node in the network) appended.

At the link layer, all data are sent to the next reachable node, which may be either the correct recipient or a network gateway.

All outgoing data from the link layer are temporarily stored in a buffer which ensures that bursts of data are formed into a steady stream of data.

1.3.2 Internet layer

The internet layer is where subnetwork routing occurs. When data arrive from the link layer, the link layer header (often an Ethernet header) is stripped off and the internet protocol (IP) header is parsed. If the recipient is not the current device, the packet is forwarded to the subnetwork assumed to know the recipient.

The IP header contains information about who the recipient and sender are, which protocol is used etc. The IPv4 header is shown in table 1.1. As the internet has been populated by devices, the number of unique addresses available is declining. A new version called IPv6 has been designed and it expands the number of available addresses from 2^{32} to 2^{64}, which should be enough for the foreseeable future.

If the data is sent to the current device (i.e. matching IP-addresses) the IP-header is stripped and the data is forwarded higher up the chain.

1.3.3 Transport layer

The transport layer handles the redirection of data from the internet layer to a specific application. A device may have several applications running concurrently. Each network enabled application gets one, or several port numbers assigned. A port number is a unique identifier that identifies the application and the protocol used for the lifetime of the application.

UDP (section 2.1) and TCP (section 2.2) are the most common internet protocols. UDP was designed for data which do not need reliable transfer, such as real time communication, streaming etc. while TCP was designed for reliable data delivery.

1.3.4 Application layer

The application layer is a common denominator for all applications running on a device, both network connected and not.

It contains protocols such as the hypertext transfer protocol (HTTP). The common HTTP port is port 80, meaning that an application listening for HTTP traffic has to listen to port 80 (unless the client knows about the other port). As only one application may be bound to a specific port, no other applications may listen to port 80 during the duration the web server is running.

Chapter 2

Transport protocols

2.1 UDP/IP

The User Datagram Protocol (UDP) is the preferred protocol for real time transfers and multicast traffic. UDP is non-responsive, i.e. the protocol does not have any built in mechanisms for discovering and handling congestion in the network. When data is sent using UDP, the application sends the data to the receiver without knowing if the data arrives in order or if the data arrived at all, making it the programmers task to ensure that the data arrives at the receiver if necessary.

As UDP is non responsive, wrongly configured applications may have a huge impact on experienced network speed as they might block data from other connections in the network, causing congestion.

2.2 TCP/IP

The protocol for reliable transfer of data through the internet is the TCP protocol. It is a full duplex protocol, allowing a steady stream of data both ways on a single connection. A full duplex protocol is a protocol where the data stream is going in both directions simultaneously, as opposed to half duplex connections where the communication is going in one direction at a time. TCP's strength lies in the ability to reliably transfer data through unreliable networks. The data may take several paths through the network, allowing packets to be unordered when received. TCP takes measures to ensure that the receiver can forward the ordered data to the receiving application.

TCP is byte stream based, the data it receives from the sender application is read as a continuous stream before converting it into TCP segments and sends them to the IP module/layer. Each TCP segment may be as large as 64kilobyte (KB) but is usually the size of the path maximum transmission unit (MTU) (path MTU) in order to prevent fragmentation. The path MTU is the MTU for that specific path. In the modern internet it is restricted by the MTU of the hardware, which typically is 1500 bytes for wired Ethernet connections.

Table 2.1: TCP/IP header format[45]

Offsets	Octet	0								1								2								3							
Octet	**Bit**	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	**0**	Source port																Destination port															
4	**32**	Sequence number																															
8	**64**	Acknowledgement number (if ACK set)																															
12	**96**	Data offset				Reserved			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" bytes if necessary.)																															

When an application sets up a TCP connection, the application opens a socket, an endpoint where the data is sent from and the received data is received. The socket is identified by the IP address and a port number. There may only be one socket connected to a port number at a given time, but a socket may support several TCP connections allowing applications to have several concurrent TCP connections.

2.2.1 TCP protocol

Each TCP segment has a header and payload. The header contains information about which port the segment is sent from and which port it is sent to in addition to a 32 bit *sequence number* and a 32 bit *acknowledgement number* (table 2.1). *Data offset* describes where the header ends and the data begin, it is necessary because the *header* may contain additional options. *Window size* describes the size of the senders *sliding window*.

2.2.2 Sliding Window

The sliding window is implemented as a way to control the flow between the sender and receiver. Both the sender and receiver have their own sliding window. The 'receive window' controls the amount of data being sent to the device, as it dictates the amount of data it is capable of receiving. Because the data may be received unordered, the unordered data are stored in the 'receive window' until the correct data is received, only then the ordered data is sent to the application and the correct amount of data are freed from the window. In order to know which data segments have been received, each segment has its own *sequence number*. The *sequence number* makes it possible to know the order of the received segments. For each segment received, the receiver replies with an acknowledgement (ACK). The ACK is marked with the sequence number of the next packet the receiver expects to receive.

Using the ACK the TCP protocol is able to calculate the network RTT and estimate the bandwidth delay product (BDP). The RTT is the time it takes from the TCP segment is sent until an ACK is received for that specific segment, and it is a measure of the network delay. If the RTT increases, the network is under load. If the RTT decreases, the transfer that caused the delay has stopped.

The BDP is a measure of the amount of bytes a link is able to transport at a given time. BDP is calculated as displayed in (2.1).

$$bdp = rtt \times bandwidth \tag{2.1}$$

Say 10 segments are sent, but only segments 1,2,3,5,6,8,9 makes it to the receiver. The receiver replies with 4, as it is the segment it needs before the first chunk of data can be sent to the application. When the sender gets the request for segment 4 it resends the segment. If segment 4 is received before the new segment come from the sender the new segment is discarded when it comes because the data is already accepted. Then the receiver follows the same procedure for segment 7 and 10.

If, however, none of the ACKs are received, a 'retransmit timeout' occurs and the sender sends the data again.

If the 'receive window' is full, a packet with window size = 0 is sent to the sender in order to tell the sender that the receiver can not receive more data before the current data is sent to the application for processing.

For the sender to know when it can resume sending packages to the receiver, it sends a *window probe*, a message with no data, in order to check whether the receiver is capable of receiving more data. As long as the *window size* return 0, the sender keeps probing using set time intervals. This is a response to the possible deadlock which may appear if a packet is lost on the way, either if it is the *window probe* itself or a segment containing a new, updated *window size* [7, 8, 45].

Data transfer, window probing and ACKing can in some cases lead to unnecessary overhead and load; this is especially bad when bandwidth is scarce. In order to try to decrease the load, TCP has delayed ACKs. Using delayed ACKs, the receiver waits for a maximum of 500 milliseconds before a proper response message is sent [8]. If a segment carrying data is detected before the timeout, the ACK is applied to that segment. The process of sending ACKs and other data merged with an existing segment is known as *piggybacking*. Piggybacking is necessary for lowering the overhead as the ACK is a number in the header. If an empty TCP packet is sent, the overhead is large as there is no payload. As soon as the TCP header is merged with payload, the overhead decreases.

For transfer of big segments the link utilisation (throughput) can be good but if there is a consistent transfer of small segments the overhead per segment is so high that the throughput is minimised. As an optimisation *Nagle's algorithm* [22, 34] was proposed. The algorithm works by accumulating sent data until it fits in a single segment, only then the data is sent. This optimises the throughput by decreasing the header/packet ratio. If a segment is sent with 1 byte of data, the entire packet has a size of minimum 41 bytes (this is the minimum due to the fact that TCP headers may contain extra options). Sometimes Nagle's algorithm may cause problems. If an application has to send many small packets, as the case is in online gaming or other real time applications, Nagle's algorithm may cause unwanted delay. It is possible to disable the algorithm by using the *TCP_NODELAY* option.

2.2.3 Silly window syndrome

Silly window syndrome (SWS) occurs when the congestion window is in a state where the data segments sent are so small that the throughput and transfer efficiency decrease. It is most likely to occur in badly implemented TCP stacks, which does not take the latest improvements into consideration [10, 43].

The receiver announces the window size as the amount of data it is able to receive next, and the senders window size is the number of bytes waiting to be sent. The number of bytes to send is the window size subtracted by the number of bytes 'in flight'.

If a receiver announces a window of 1000 bytes, and the senders segment size is 200 bytes, the sender can send 5 segments. Suppose the segments are read when they arrive, the receiver still announces a window if 1000 bytes, but the sender knows that it may only send one packet. If a segment at the receiver gets read while a segment is received to the queue, the receiver may announce a window size of say 950 bytes as there are 50 unread bytes. Then the sender responds by sending a segment of 150 bytes. The next segment has a size of 50 bytes. Each subsequent segment will have a size of 50 bytes, thus decreasing performance [10]

SWS is not only applicable to the sender. The receiver may also be responsible for SWS if the receiving application slowly pulls data from the window. This can be solved by announcing a decrease in window size when the application is able to receive a lot of data at once [43].

The solution to the sender side SWS is to enable *Nagle's algorithm* which will delay the sending of segments until enough data have accumulated to send a large segment [43].

2.2.4 Congestion control

For TCP to react to an overloaded link it is using *congestion control* or, more correctly, *congestion avoidance* to manage and model the flow of segments between the sender and receiver.

Congestion occurs when there is more data arriving at a link than the link is able to transfer. If a link has a bandwidth of 10Mbps and a RTT of 100 *ms* the link, according to the *bandwidth delay product* (equation (2.1)), it has a maximum capacity, one way, of 6250 bytes.

$$\begin{aligned} bdp &= \frac{1}{2}100ms \times 1 \cdot 10^{7} bps && (2.2)\\ &= 5 \cdot 10^{-3} s \times 10^{7} bps && (2.3)\\ &= 5 \times 10^{7-3} bit && (2.4)\\ &= 5 \times 10^{4} && (2.5)\\ &= 50000 bit = 6250 bytes && (2.6)\\ &\approx 4 \cdot 1500 bytes && (2.7) \end{aligned}$$

This means that when 4 Ethernet frames are sent, the link is saturated, working at full capacity. If more segments were to be sent, they would be dropped. In reality, networks are much more fragmented, and each intermediate device (i.e. a router) has its own buffer. It is only when the buffer is full that the device starts to drop packets and congestion occur.

Congestion occurs at the *bottleneck* link, a place in the network where a network is connected to a slower network, with lower bandwidth and packets are filling the buffer to a point where packets are dropped.

When a packet is sent from a link with high bandwidth to a low bandwidth link, each packet takes longer to transfer. If an Ethernet frame were routed from a $10 Mbps$ link to a $1 Mbps$ link, the transfer would take 10 times longer on the slow link, assuming the RTT on both links is the same. When the ACKs are sent from the receiver, each ACK would be sent with a time difference equal to the transfer time of the slow link. This is known as the ACK clock and it is stable when the transfer is ongoing (unless the link to the first machine is congested). The ACK clock is an essential part of TCP, which helps smooth out traffic and avoid unnecessary queues at routers [24, 34].

In order for the sender to know which segments it has in flight (on the link, travelling to the receiver) each sender implements a congestion window which counts the amount of segments sent. The window is a counter that counts each sent segment until it has arrived at the receiver and an ACK is received. When an ACK is received, the congestion window is incremented.

The initial window size is at most 4 segments [6], this size has recently been raised to 10 segments in some hosts, e.g. Linux [9]. For each segment sent, the window size is enlarged by a segment. This makes the window size double for each RTT [34] and is known as *slow start*, even though it is exponentially increasing.

In order to control the *slow start*, a threshold known as the slow start threshold (ssthresh) is implemented. When the window size is at the threshold level or a packet drop occurs, the *slow start* is discontinued. To start with, the ssthresh is set arbitrary high to let the *congestion window* grow as large as possible. At any given time, when TCP discovers dropped packets the ssthresh is set to half the size of the current congestion window. Packet drops are detected by receiving the same ACK three or more times in a row [33].

Each time the window size reaches the size of the current threshold it starts growing linearly until packets are being dropped once more. The new ssthresh is set to half the congestion window size when a drop occurs.

The growth rate of the congestion window when the ssthresh is reached is roughly $1/$congestion window size. With this growth rate the congestion window size increases by one packet for each sent congestion window.

Chapter 3

Queues

3.1 What are queues?

All network connected devices have one or more network interface controllers(NICs), a physical connection to the network. While most devices only need one, some devices (e.g. routers) have multiple NICs which connect the device to different networks. As data is sent from an application, the outgoing packets are placed in a buffer before they are sent out through the NIC onto the network. Each NIC has its own internal buffer [32], and the *internet layer* (section 1.3.2 on page 5) also has its own [32].

A buffer is a temporary storage (with a predefined size) of data. The buffer's main function is to smooth a bursty data stream. In the early days of networking, when memory was scarce, the buffers were of a limited size due to the prices of memory. As the prices for memory decreased, additional memory was added to the devices, leading to overly large buffers that could hold a lot of data [14, 24].

The data is queued in the buffer before being relayed further. All queues induce a certain amount of delay as it takes time for the queue to drain. This time induces delay on every connection using the buffer. Usually the queues disappear quickly as packets are sent to the next node in the network, but if a queue forms at the slowest point, the bottleneck, it takes time before it is gone. If the bandwidth for the incoming packets is higher than the outgoing bandwidth, the queue length will keep increasing until the buffer overflows.

3.2 How queues form

Queues form when the rate of incoming data is higher than the rate of outgoing data. When the output rate is lower than the input rate, packets accumulate and create a temporary queue.

$$L = \lambda W \tag{3.1}$$

A definition of queue behaviour is seen in Little's law (equation (3.1)), which defines the queue length, L, by multiplying the arrival rate, λ, with the average time, W, each item stay in the queue.

$$W = \frac{L}{\lambda} \tag{3.2}$$

The average time an item stays in a queue is defined in equation (3.2).

According to [24] there are two types of queues, 'good' queues and 'bad' queues. They differ in how the queue impacts network performance. A good queue is a queue that is emptied in one RTT. After one RTT the queue is gone, and the buffer is ready to get more data. A good queue does not impact the RTT negatively as the queue does not create a significant delay for other connections whether it is real time traffic or large file transfers, and disappears after a single RTT.

A bad queue is a queue that lingers after one RTT. When a bad queue forms, there are more packets coming before the queue is emptied. Thus, the queue restricts traffic and creates delay. A reason bad queues form can be a wrongly calculated congestion window in the TCP stream that sends data. If the window is calculated to be 10 packets larger than the link is able to transfer, a queue of length 10 forms in a bottleneck buffer, creating delay for other traffic. If the packets linger in the buffer after several RTTs it is defined as a bad queue. 'Standing queue' is another name for 'bad queue'. A standing queue is a queue which is not emptied when a packet drop occur, thus inducing an additional delay in the network.

A bad queue can occur when the TCP congestion window is larger than the actual 'pipe size', the amount of data a link is capable of transporting. Due to the way the TCP congestion avoidance algorithm works, a bad queue is not necessarily discovered as it creates delay, not indicating congestion, as the following example shows.

Figure 3.1 on the facing page displays a path with a bottleneck. The fastest link has a bandwidth of 100 Mbps and a one way delay of 1 ms, while the slower link has a bandwidth of 10 Mbps. At connection startup the congestion window at the sender is negotiated to be 25 packets, not 20 packets which is the link pipe size and the first congestion window is sent and queued in the bottleneck buffer.

In figure 3.2 on the next page the packets are sent through the bottleneck towards the receiver. After one RTT the sender receives the first ACK and the TCP slow start is halted, this causes the congestion window growth rate to decrease as described in section 2.2 on page 7. The ACKs arrive with a delay of 9 ms as the packets are transferred from the buffer. As new packets arrives each millisecond, the packet has to wait for 9 ms at the bottleneck in order for the packet in the front to be transfered. Each queued packet has to wait 9 ms longer than the packet in front in the buffer. When the ACK is received, the TCP algorithm sends the next packet in the congestion window on to the link. The time between packets is long enough for a packet to leave the buffer. As the congestion window is larger than the 'pipe size', a queue is accumulated in the bottleneck buffer. As new packets arrive as others leave, the buffer will have a persistent queue which will continue to grow because of the continually increasing congestion window size. This is also known as a 'standing queue'.

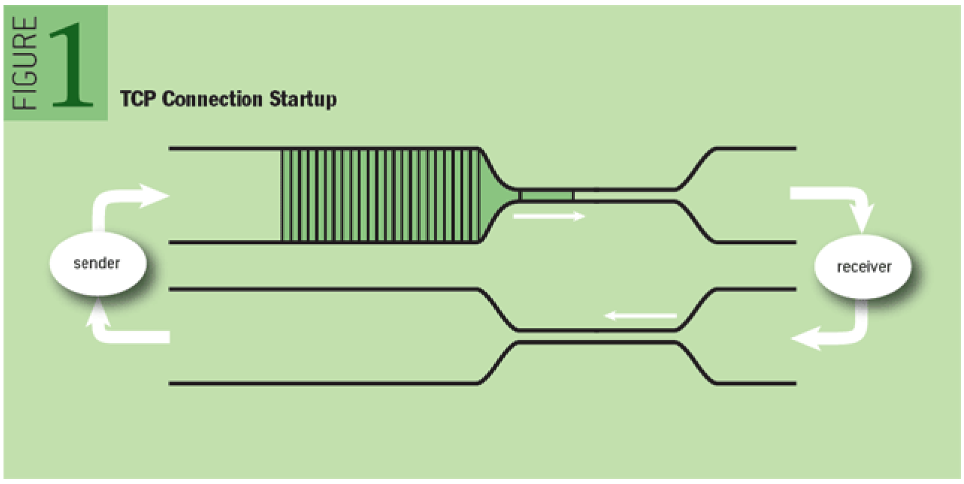


Figure 3.1: TCP connection startup [24]

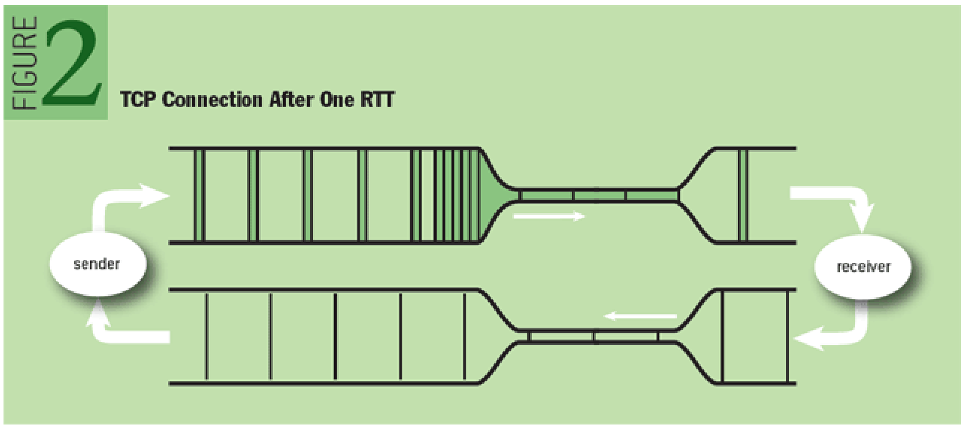


Figure 3.2: Connection after one RTT [24]

Figure 3.3 on the next page shows how the queue length at the buffer and the TCP congestion window size change as the queue persists. The queue length is the difference between the congestion window size and the 'pipe size', which in extreme cases may create a massive delay. The small variations in queue length are due to small time variations in the sending and receiving of packets.

Figure 3.4 on the following page shows a good queue, the queue drains completely before new data arrives. In the figure the receiver sends an ACK for each window size, which is a fragile method, but it works as the sender sends the whole congestion window when the ACK arrives.

3.3 Scheduling

Scheduling is a different part of queue handling. The scheduler moves incoming data to the right queue. The simplest version of scheduling is to move every packet into the same queue.

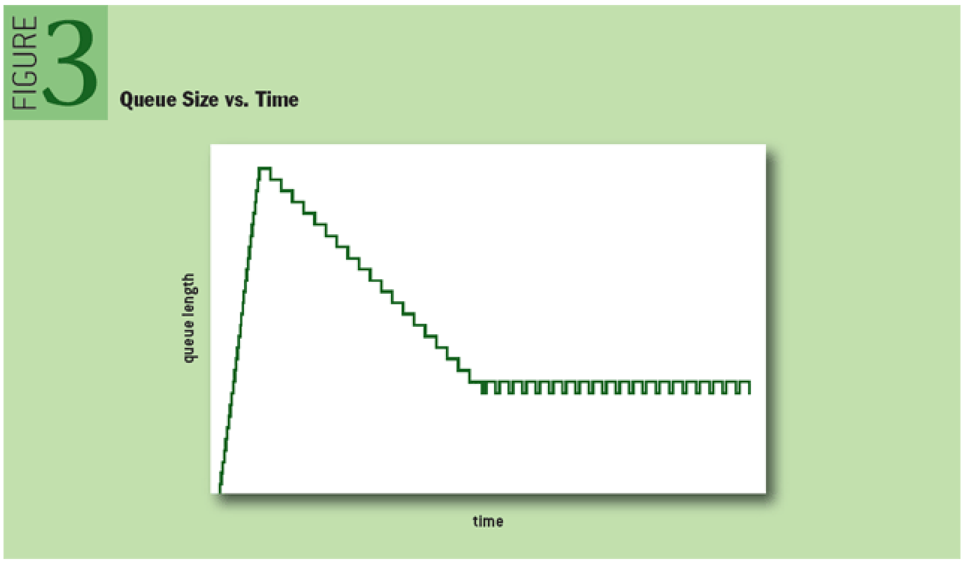


Figure 3.3: Queue length vs time [24]

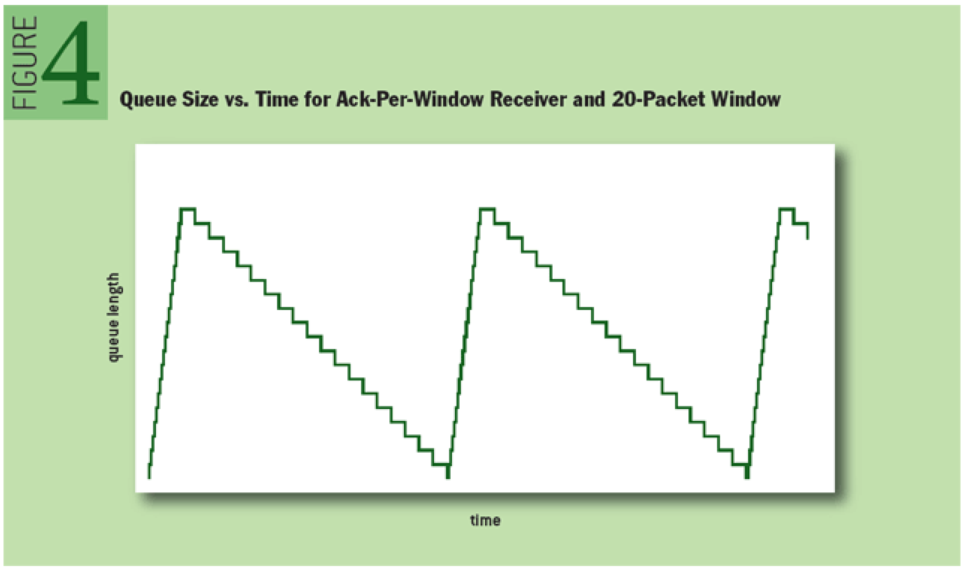


Figure 3.4: Queue length when sending one ACK per congestion window [24]

Using scheduling it is possible to make advanced queuing mechanisms where different types of traffic gets individual treatment, for instance ensuring *quality of service (QoS)* for VoIP and other streams that need low response time.

The job of a scheduler is to organise data into queues in a buffer. There are two main groups of schedulers, passive and active. Passive scheduling is achieved when the scheduler does not implement much logic. Most of the passive schedulers use either a counter, which determines which queue the data are going to, or implement no form for logic. Counter based scheduling may be based on round robin where the queues are traversed in circular order. Each time a new packet arrives, the active queue is switched.

3.4 Tail-drop discipline

Drop-tail queues are the standard queue limitation method in modern computer networks. They are a first in, first out (FIFO) based implementation which drops data when the queue is full, and thus it is easy to implement on devices with limited processing power and a limited amount of available memory. Modern devices create queues by allocating available memory. Because it is the outgoing bandwidth that limits the amount of sent data at a given time interval, network devices needs to queue outgoing data.

Using a drop-tail queue, when the queue is full, the overflowing data is discarded. This forces the responsive connections to reduce their sending rates. Responsive connections discover when a packet drop occurs. They may be implemented using UDP, but the most common algorithms are TCP based, which allows the connection to take appropriate action when congestion occurs.

With TCP the tail-drop queue causes some connections to get more resources than others. With a lower RTT the transfer protocol is able to push more packets onto the link because of each ACK arrive within a short time of the previous. As TCP sends data only when it gets an ACK, the amount of allocated bandwidth is higher for connections with a low RTT.

The tail-drop queue disregards real time applications with a low latency goal. Real time applications send small amounts of data at regular intervals and they are usually non-responsive. Real time transfers are dependant on low delay and low send rate variation in order to provide the best user experience possible. Because drop-tail queues drop data when the queue is full and real time transfers can be non responsive, the recipient may experience a drop in performance or frequent lag when the queue is full due to the amount of packets delaying the ones behind.

A queue with a maximum size of 50 packets will induce a delay of 60 ms on a 10 Mbps connection when the queue is full. The human brain is able to process a continuous stream of images into fluid animations when the images shift at least 24 times per second [29]. A movie at the cinema has a lower frequency of 24 hertz, the inter-image delay is about 40 ms. If a packet is able to contain an image, a full queue would induce a 60 ms frame delay, which is about 16 hertz or 16 frames per second and the stream would be perceived as choppy.

Using IP telephony, the target delay is 20 ms with a maximum of 150 ms [42]. A 60 ms delay may not be noticeable, but it should be avoided for the perceived quality of the real time stream.

3.4.1 Problems with queues using tail-drop

Lockout

One of the problems with drop-tail queues is lockout. Lockout occurs when the incoming flows are synchronised. Synchronisation occurs when the queue is full and packet drops have occurred. As packets are dropped, the flows reacts by decreasing their send rates. The flows with low RTTs may get an unfair proportion of the queue as the flows with high RTTs may get a lot of packets dropped due to a nearly full queue.

Global synchronisation

The global synchronisation problem affects all drop-tail queues, as the drops are more frequent when the queue is full. Each connection that encounters a full queue drops packets and reduces its send rate. As almost all packets are dropped, every connection reduces its send rate simultaneously, making the throughput plummet as no data is transfered through the link. Because of the TCP designed, each connection increases their sending rate simultaneously, leading to a full queue and dropped packets. Thus alternating between empty and full queues, magnifying the problem.

The solution to global synchronisation is to either drop packets randomly from the queue or enforce a front drop policy. The front drop policy drops packets from the front of the queue rather than from the end. Thus, not leading to global synchronisation as not every connection gets dropped packets simultaneously [24].

Full queues

Another problem with tail-drop queues is the *full queues* problem, it is a problem for all algorithms using the tail-drop discipline. A queue should be able to handle bursts in traffic as packets have a tendency to come in bursts to the router. A nearly empty queue is able to handle the bursts without dropping a lot of packets, while a full queue, or a nearly full queue, has to drop packets as they arrive. This leads to a lot of dropped packets, lower utilisation of bandwidth, global synchronisation and lockout.

A queue in a non-full state is a well working queue due to its ability to absorb bursts of data. The queue length should not represent the amount of data it can hold, but rather the size of the bursts it is able to absorb. The queue size should be proportional with the outgoing bandwidth from a device. Higher outgoing bandwidth should translate to a queue which could absorb more data [24, 34]. But the full queue problem would still persist.

3.5 Queueing discipline

The combination of a scheduler and drop policy is often called a queueing discipline (qdisc). There are several qdiscs available, the most common is FIFO, which implements a single queue with tail-drop. There are two classes of qdiscs;

- Classless queueing discipline
- Classful queueing discipline

Classless qdiscs do not implement a lot of logic in the scheduling.

First in, first out

The FIFO qdisc implement a single queue using simple tail-drop. The first packets which are queued are the first packets to be sent. When the queue is full, the incoming packets are dropped.

Stochastic Fair Queueing

Stochastic fair queueing is a more advanced qdisc [18]. It has a scheduler which orders packets into queues based on the connection information (e.g. sender port and IP address and receiver port and IP address). The queue a specific packet is sent to depend on the output of the hashing function.

Each queue is a basic FIFO queue with tail-drop. When the queue is full, new incoming packets are dropped.

Outgoing packets are multiplexed into a stream using round robin, which selects each queue in succession. For each queue, the first packet is removed from the queue and multiplexed into the outgoing stream of data [18].

3.6 Queue management

Queue management is an easy way to restrict and reduce traffic flow. There are many ways to restrict flows, ranging from *drop-tail queues* to *active queue management*. The method used depends on the hardware deployed in the network. In the early days of networking, the philosophy was that the intermediate hardware should do as little processing as possible as the available resources were limited, such as processing power and memory capacity.

As the years progressed and processing power and memory have become cheap, the routers and intermediate network devices have become more powerful, thus the queue sizes have grown and are delaying data at bottlenecks. TCP will always make queues grow (equation (3.1) on page 13) and the challenge is how to limit them most efficiently.

3.7 Active queue managers

The drop-tail queue is a well known queue manager that works on low performance systems with limited memory. Every device using it has the possibility of becoming the point of delay in the network. The outgoing queue from devices should not be the cause of excess delay. When a queue enforces a delay of several seconds, the performance will suffer. Overly large queues trick the TCP congestion avoidance algorithm into thinking there is no congestion on the link, thus the window keeps growing and more data are sent as the ACKs keep coming.

Active queue management is a countermeasure to the rapidly growing queues. As modern routers and intermediate network devices have more processing power than ever before, they are able to run more complex algorithms in order to limit queue length.

The earliest active queue manager (AQM) versions had a lot of poorly documented parameters which had to be statically configured in order to get the AQM to work correctly. If a parameter configuration worked perfectly for some cases, other cases may have caused the AQM to perform poorly, thus making the job difficult for anybody wanting to correctly configure the AQM for several use cases.

3.7.1 RED

After the congestion collapse of 1986, alternative methods were researched. In an attempt to fix congested networks and optimize performance, the random early detection (RED) AQM was created. The RED algorithm was designed to run on intermediate routers, as opposed to the congestion avoidance algorithms which is implemented at the end points and reacts when the link has become congested.

RED uses the average queue length to determine the drop probability. The drop probability increases when the queue length has reached a set point. When drops occur the queue works as a drop-tail queue, dropping packets from the end of the queue. Due to the drop-tail discipline, RED shares the problems of other drop-tail queues. Connections with low RTT have an advantage over the ones with higher RTT as they would get a bigger cut of the queue, effectively choking the high RTT connections.

As the queue gets filled, the drop probability rises. When a packet is received, the algorithm determines whether the packet should be dropped or not using the drop probability and a drop threshold. The drop threshold determines whether the drop probability rises. A queue that is filled 50% when the drop threshold is set to 0, has a 50% drop probability. This translates to half of the packets received being dropped and the sender having to transfer them once more when the transmission timer times out, or when ACKs are received stating that some packets are missing.

When RED drops a packet it either truly drops the packet or sends an explicit congestion notification (ECN). ECN is sent to the sender from the receiver in order to get it to reduce its rate. The ECN is suggested as a replacement for packet drops [17, 19, 28]. Because it has the same effect as packet drops, ECN is heavily discussed [19].

RED has many parameters and every one of them need to be configured manually. This raises difficulties using the algorithm, and requires users to know every detail about the links which are connected to the device running RED. Every time some parameters change, whether it is RTT, bandwidth or queue length, the queue manager has to be reconfigured.

3.8 Components of an Active queue manager

3.8.1 Congestion indicator

Every AQM has a congestion indicator. The function of the congestion indicator is to discover possible congestion. Different AQMs have different congestion indicators.

Rate-based AQMs use the current flow rate as the congestion indicator. If the output rate drops below a certain level or the arrival rate increases, which in turn allows the queue to grow if $rate_{out} < rate_{in}$. The congestion indicator for a rate-based AQM could keep the arrival rate at a certain percentage of the link capacity [1].

Queue-based AQMs use the queue length as the determining factor when drops occur. When the queue length is close to a defined target, the drop probability is higher than for lower queue lengths. The early iterations of AQMs were queue-based. Queue based congestion indicators will induce a large drop probability when the queue is almost filled, even when the incoming rate is lower than the drain rate. This causes a high drop rate even though the queue length is decreasing, thus leading to a suboptimal utilisation of the link capacity. In the opposite case, a queue based AQM drops a fraction of the queue size when the queue length is small even though the incoming rate is higher than the drain rate, thus leading to congestion.

Delay-based congestion indicators use the total queue delay when determining the drop probability. They implicitly control the queue length by monitoring the amount of time a packet has spent in the queue. Some AQMs use a predetermined amount of time in order to control packet drops. For example, some AQMs use 5 ms as a congestion indicator [1, 3, 11]. If a packet spends more than 5 ms in the queue, the packet is dropped.

3.8.2 Congestion control function

The congestion control function decides what should be done when the congestion indicator changes. It works as a mapping function where the congestion indicator value is the input and output is the drop probability. The marking of packets to be dropped is closely connected to the congestion control function. The packets to be dropped are determined by the probability output. Even though the drop-algorithm is often overlooked (it is common to always only drop the newly arriving packet), different algorithms can have different outcomes for the same dropping probability causing variations in queue length, loss and delay.

3.8.3 Feedback mechanism

The feedback mechanism is responsible for sending feedback when a packet is not forwarded to its sender when the queue increase over a set point. Feedback can be sent by dropping packets, in which case the TCP algorithm detects packet drops (triple-duplicate ACKs or timeout) or by sending an explicit congestion notification (ECN). When an ECN is sent, the packet is forwarded to the receiver, and the sender receives a notification telling it to lower the transmission rate and shrink its congestion window.

Packet drop probability

The packet drop probability is a central component in the AQM feedback mechanism. The probability controls which packets are lost, and at what rate. A packet drop probability is calculated differently from AQM to AQM. Rate based AQMs use the incoming and outgoing rate to calculate the drop probability, while queue based AQMs determine the drop probability by measuring the queue length against a set point. With delay based AQMs, the drop probability is determined by calculating the time packets used to traverse the queue. If the time calculated, and/or measured, exceeds a set point, the probability is raised.

Packet drop

Packet drop is the usual feedback mechanism used in computer networks. When a queue is overflowing packets are dropped and the sender responds to the loss.

Explicit congestion notification

Explicit congestion notification (ECN) is an alternative to packet drop. The ECN is a marked packet that is replied to the sender which cause the congestion, in an attempt to reduce the window size. Due to the marked packet, the connection algorithm does not need to wait for packet loss and the window size should decrease faster without transmission of unnecessary packets (which would probably be dropped anyway).

The downside with ECN is that it is not implemented in all TCP algorithms, as some may chose to ignore the ECN.

Even though ECN is supposed to help congested links, it has proved to be less effective than dropping of packets [19] when used in networks with high congestion due to the amount of packets still in the network. During a congested state, the network cannot handle an increase in packets in transmission and the best solution may be to drop the packets in order to clear the congestion.

3.8.4 Types of Active queue managers

There are several types of AQM, depending on which criteria the AQM uses for discovering a potential queue. The AQM types are

Queue based This type of AQM uses queue length as its congestion criterion. The AQM starts dropping packets when the queue length is growing towards it maximum length; as the queue grows the dropping probability increases.

Rate based Rate based AQMs use the rate of the incoming flow to determine when data should be dropped.

Load based Load based AQMs use the device load as the congestion indicator. The drop probability is given by $P(drop) = {}^{rate_{in}}/_{rate_{out}}$, if $P(drop)$ is higher than 1 packets are queueing, thus the drop probability should increase.

3.9 Congestion control

Standard congestion control algorithms are implemented at the connection end points. The connections using end point congestion control are reactive, thus the controller reacts after the link has become congested. The reaction time corresponds to the queue length at the bottleneck. If the maximum queue length is $1\ MB$ and the bottleneck has a bandwidth of 10 Mbps, the congestion control would use up to 8 seconds to react.

Because the AQM is running on the intermediate devices, it could discover a possible queue in advance, being proactive. If a queue is about to reach a level where the network performance would suffer, the AQM could signal the sender to reduce its rate. A signal could be sent in different ways, either by dropping packets or by sending an ECN if it is supported by the congestion control algorithm. If the AQM has several queues for the active connections, it could easily restrict some queues when they become large while others remain untouched.

AQMs drop packets with a given probability. If the link becomes congested, the drop probability increases, if the link has more capacity the drop probability decreases.

3.10 CODEL

CoDel is a queue based AQM designed with the following goals in mind [3]:

- is parameterless – has no knobs for operators, users or implementers to adjust
- treats ‘good queue’ and ‘bad queue’ differently, that is, keeps delay low while permitting necessary bursts of traffic
- controls delay while insensitive (or nearly so) to round trip delays, link rates and traffic loads; the goal is to ‘do no harm’ to network traffic while controlling delay
- adapts to dynamically changing link rates with no negative impact on utilisation
- is simple and efficient (can easily span the spectrum from low-end, Linux-based access points and home routers up to high-end commercial router silicon)

The queue length measure is implicit as the CoDel algorithm measures the queue delay. If the queue delay reaches a point which is higher than the current set point, it drops a packet.

CoDel differentiates between 'good queues' and 'bad queues' [3]. A 'good queue' is a queue which is gone after one RTT when a drop occurs, a 'bad queue' is a queue which is persistent even after a packet drop has occurred.

Like other AQMs CoDel has three basic components in order to control the queue length:

Estimator figure out what we have got

Setpoint know what we want

Control loop if what we have got is not what we want, we need a way to move it there

3.10.1 Estimator

In order to know the current queue length in the router, an estimation is needed. The estimator works by tracking the time a packet spent in the queue. If there is no queue, an incoming packet is sent immediately and the time spent in the queue is 0. If there is a queue, the time a packet spent in the queue increases.

In order to determine if a queue is forming, a local minimum is computed. This local minimum is the minimum amount of time a packet spent in the queue. The minimum is updated at frequent intervals in order to assure that the value is not going stale.

3.10.2 Setpoint

The setpoint is the point where the AQM takes action and drops a packet in order to limit the queue length.

During tests, the optimal setpoint was measured to between 5% and 10% of the RTT while having a near 100% bandwidth utilisation [3]. The high bandwidth utilisation is the result of a small standing queue. If a queue is completely empty, the bandwidth utilisation decreases as there are no more data to send. Thus, a very small standing queue is a good thing.

3.10.3 Control loop

The control loop is the algorithm which determines when packets are dropped from the queue. Most of the classical control theory deals with the control of linear, time-invariant, single-input-single-output (SISO) systems. Control loops for these systems generally come from (well understood) proportional-integral-derivative (PID) controllers [3]. The fact is that queues are not a SISO system as they do not operate with a single connection, but multiple. Thus, an AQM is a multiple-input-multiple-output

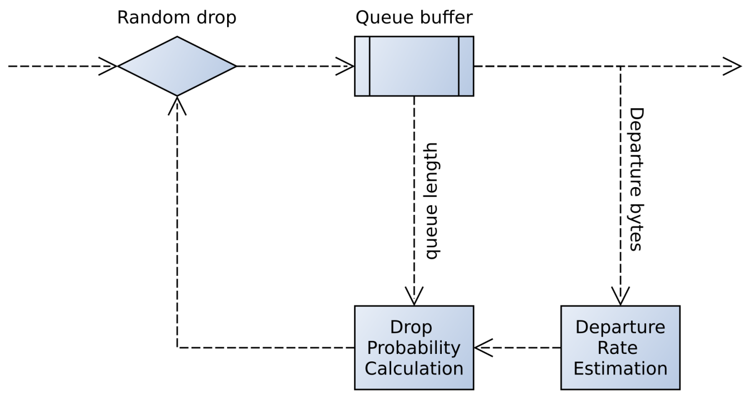


Figure 3.5: PIE Structure [4]

(MIMO) system which handle both input and output of different connections [3]. As the MIMO works on multiple streams, a single packet drop from a stream will fall short as only the stream which dropped a packet reacts. In a MIMO system there will be a need of dropping one packet from each stream that should react.

As the different type of connections in a MIMO system range from non-responsive UDP streams to responsive TCP streams, the AQM needs a drop policy that works on both types.

When a persistent, bad queue occurs a packet drop from the stream will remove a bit of the queue in an RTT due to the halving of congestion window which occurs after a packet drop (section 2.2 on page 7) [3, 34, 45]. Because of this, a drop of less than one packet per RTT with increasing packet drop probability is the way to go. This gradient drop policy reacts to the load at the bottleneck. The drop probability increases until the persistent, bad queue is shrinking before reseting when a new bad queue occurs [3]. The interval between two drops is decreasing in inverse proportion to the square root of the number of drops since the dropping state was entered [3]. Thus, packet drops get a slow start, but the packet drops become more aggressive as the time goes. This is the heart of the CoDel AQM.

Because the best rate to start dropping is at slightly more than one packet per RTT, the controller's initial drop rate can be directly derived from the estimator's interval [3].

3.11 PIE

Unlike CoDel, PIE is an AQM which use the dequeue rate to determine the drop probability for incoming packets.

As shown in Figure 3.5, PIE has the following components:

3.11.1 Random drop

For each packet arriving PIE randomly decides on dropping in an attempt to limit the queue length. The probability, p, is obtained from the 'Drop probability calculation' component [4].

When PIE decides to drop a packet, it drops the packet from the end of the queue.

3.11.2 Drop probability calculation

PIE drops packets with a certain probability. This probability is defined by equation (3.4).

$$est\_del = qlen / depart\_rate \tag{3.3}$$

$$p = p + \alpha \times (est\_del - target\_del) + \beta \times (est\_del - est\_del\_old) \tag{3.4}$$

$$est\_del\_old = est\_del \tag{3.5}$$

In equations (3.3) to (3.5) $qlen$ is the current queue length and the departure rate is determined by $depart\_rate$ which is obtained from the 'Departure rate estimation' block.

The est_del and est_del_old are variables which represent the current estimation and old estimation of queueing delay, while the target latency value is represented by $target\_del$.

3.11.3 Departure rate estimation

Queue draining in a network often varies because of varying conditions like other queues sharing the same link, or the link capacity fluctuates. Rate fluctuation is a common problem in wireless networks.

PIE measures the departure rate in the following way:

- If there are enough data in the queue, we are in a measurement cycle.
- If in a measurement cycle, update the current dequeue count.
- If the dequeue count is higher than a given threshold, calculate the departure rate.

The PIE departure rate is recommended to be set to 16KB assuming a typical packet size between 1KB and 1.5KB [4]. This threshold is long enough to calculate an average draining rate but also fast enough to reflect sudden changes in the draining rate.

3.12 FQ CoDeL

FQ CoDel is an AQM which mixes the properties of CoDel with the FairQueue discipline (section 3.5 on page 19), that queues packets in different queues based on the result of an hashing algorithm. The FQ CoDel algorithm is renamed FlowQueue CoDel, not FairQueue CoDel, due to its impact on flows [5].

Each of the queues is a CoDel-queue which tries to limit the queue length by limiting the maximum amount of time any packet use through the queue. If a packet uses more than the set limit, the packet is dropped or an ECN is sent in reply.

Any packet which is dropped at the end of the queue may cause global synchronisation where the connections would decrease the rate simultaneously, thus leading to possible starvation of the link. After which, every connection tries to increase their window, thus the buffer overflows due to the amount of data received at once. If the synchronisation is not solved, the network will vary between starvation and flooding.

FQ CoDel makes a distinction between 'new' and 'old' queues [5]. New queues are queues which have no standing queues, whereas the old queues do. If a queue is standing, it is classified as old, which is the state it is in until the standing queue is no more. When a standing queue is cancelled, it is moved from the old list to the new list. For each iteration a certain number of packets are moved from the queue and sent; if the queue still has any packets (which does impose more than a RTT in delay) the queue is classified as old.

Every FQ queue has CoDel implemented, which make the algorithm hierarchical. Every 100 ms (as described in section 3.10 on page 23) the minimum value is checked and the packets that have been in the queue for more than 5 ms are dropped.

3.13 Summary

Queues appear in the buffers of bottleneck devices. The buffers are implemented as a measure against bursty connections. With the increase of cheap memory, the device buffers are getting larger. Larger buffers are able to sustain a larger queue which induces an additional delay for every connection using the buffer.

If an overly large buffer exists in a heavily used bottleneck node, the buffer is able to sustain a standing queue that induces several milliseconds, or even seconds of delay for the other connections. As an increase in delay is not monitored by TCP, the queue is allowed to grow. The TCP congestion algorithm decreases the TCP congestion window when it detects congestion. Congestion is detected by a packet drop.

According to [24] there are two types of queues, 'good' and 'bad' queues. A good queue is a queue which drains after one RTT. A bad queue is a queue that lingers in the buffer after RTT and is the result of a badly configured buffer. 'Standing queue' is another name for a 'bad' queue.

There are solutions, active queue managers (AQMs), to the problem, but they are often difficult to configure and they need to be implemented and enabled at all bottleneck nodes in the internet, which is time consuming. The first AQM, RED, was difficult to configure. Recently other AQMs which have less configuration parameters, and are largely ‘plug-and-play’, have been made. Even though the configuration is minimised, the problem of distributing them to the nodes still persists.

Part II

Queue flusher

Chapter 4

Queue Flusher

A possible solution to the lack of enabled active queue management algorithms on routers is a queue flusher. The queue flusher is designed to run on any network device that is forwarding data from one network to another. If a queue is detected, the application tries to empty the queue by sending unusable data in order to fill the queue, thus making a congested buffer that the responsive connections respond to (sections 2.2 and 3.6 on page 7 and on page 19). When connections respond, they decrease their window size, which lessens the amount of 'in-flight'-packets. The effect is that the bottleneck buffer is emptied before a new load of packets is queued at the bottleneck.

Figure 4.1 shows a network with multiple points where a delay inducing queue may form. Queues often occur in the outskirts of the internet, as it is where the consumers live and the network is not closely monitored. They enable their devices, which in many cases are wrongly configured and have overly large buffers, additionally the bandwidth difference between nodes make the problem even larger as buffers are not sized correctly for the current bandwidth. Due to this mismatch, large queues are created and the network responsiveness and delay suffer [14, 16].

Listing 4.1 on the following page provides a simple algorithmic version

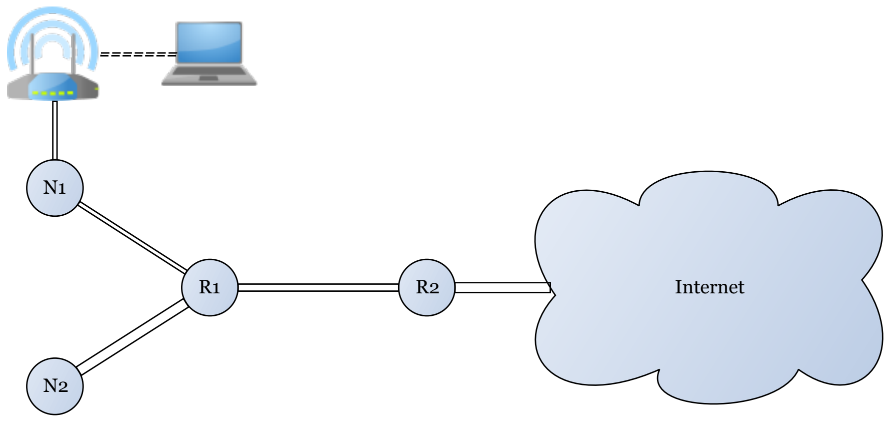


Figure 4.1: A network with multiple bottleneck points

Listing 4.1: Flusher pseudocode

```
program QueueFlusher
begin
    # Insert data packet
    function insert_data_packet(hash, packet)
    begin
        flow = get_flow(hash);
        if packet_exist(flow, packet) return;
        add_packet(flow, packet);
    end

    # Check ack
    function check_ack(hash, packet)
    begin
        if not ack return;
        flow = get_flow(hash);
        data = get_data_packet(flow, packet);
        packet.rtt = calculate_rtt(data, packet);

        # Flood if rtt is above threshold
        if packet.rtt > X*basertt flood();
    end

    # Capture packets from network interface
    while next packet
        # If packet not tcp, continue
        if packet.type not tcp continue;

        # Calculate hash
        hash = calculate_hash(packet);

        insert_data_packet(hash, packet);
        check_ack(hash.inv, packet)
    loop
end
```

of how the flusher works. For each packet it captures, the flusher, checks whether the packet is a TCP packet or not. If it is not, the packet is ignored. If the packet is a TCP packet, a hash of the flow parameters is calculated. The hash is based on the packet source and destination IP addresses and port numbers. When the hash is created, the correct stream is found by matching the hash against hashes of known streams. If the packet exists in the stream's list of packet hashes, it is ignored, else its hash is added to the list.

If the captured packet has the ACK flag set in the packet header, a lookup using the inverted hash key is done. The inverted hash is calculated from the same metrics as the regular hash, only the metrics are swapped. This lets the ACK be looked up in the packet list of the reverse stream (The ACKs are sent from receiver to sender). If the packet matching the ACK is found, both throughput and RTT are calculated.

The current RTT is checked against a set threshold of the measured base RTT. If $rtt > threshold \times basertt$ then the flooding is initiated.

4.1 Application

The queue flusher is a user space process as it (per design) is able to run as an unprivileged user and not a privileged user. The different types of users are confined to their spaces [27, 44].

An unprivileged user can execute all programs that do not need special permissions from the operating system (OS), while a privileged user can execute programs that need special permissions, such as installing new software on a computer.

As the queue flusher works in user space it can be prevented from doing malicious tasks, as opposed to if it was running in *kernel mode*. Any application running in kernel mode have the opportunity to crash the kernel if programmed incorrectly. In Linux and other *NIX variants only privileged users are able to run programs which enter kernel mode. Privileged users may be users running as super user (elevated permissions) or a user who belongs to a certain group which gives the user the right runtime permissions.

The queue flusher is of the later kind, it can be run by an unprivileged user who belongs to a certain group or is running the application as a user with elevated privileges.

4.1.1 Goals

Based on how active queue managers (AQMs) works, the queue flusher have a set of goals that comply with the ones of AQMs.

Reduce round-trip time The queue flusher should reduce the overall RTT for connections going through a bottleneck device.

Ensure fairness Fairness is not possible to achieve in a queue flusher as it does not know how the queue is managed on the bottleneck. Fairness is achieved by scheduling (section 3.6 on page 19) as the way the queue is administered is the only way fairness can be achieved. For responsive streams such as TCP, a FIFO queue using the drop-tail policy is favoring streams with low RTT [15] regardless of whether the queue flusher is running or not.

Throughput Connections delayed by a queue should see no improvement or a slight decrease in throughput while the RTT decrease. Heavy decrease in throughput indicate the presence of other streams.

4.1.2 Requirements

Every application has its own requirements in order to function properly. A queue flusher needs to be independent, both in where it is placed in the network and which protocols it supports. The queue flusher is designed for working with responsive connections, specifically the TCP. There are also responsive streams using the UDP, if the queue flusher was supporting those streams, the side effect could be unnecessary dropping of packets from unresponsive streams (e.g. real-time communication) which could result in a poorer experience for the end user. Thus, the queue flusher does not monitor UDP streams, but any responsive UDP stream should react to the packet drops imposed by the queue flusher.

In order to be able to monitor streams and discover congestion it needs the following requirements:

Measure round-trip time Measuring of the round-trip time (RTT) is required for detection of queues in a packet switched network as the RTT varies according to the queue length in the bottleneck (section 2.2 on page 7.)

Measure base RTT The base RTT is as important as the RTT as it describes the lowest measured RTT in the network. The base RTT is a measure of the latency in the network and a low base RTT indicates a low response time in the network. As the base RTT is dependent on the current stream, the base RTT for a stream may be high, even though the base RTT for that connection is low compared to the base RTT. This indicates the presence of other streams using the same path. As the flusher does not know that for certain, it should treat the base RTT for the current stream as if it is the current base RTT in the network.

Discrete congestion controls The queue flusher should work for TCP streams using different congestion avoidance algorithms.

Flush queues The ability to flush queues is an important requirement. If a queue cannot be flushed, the flusher should retry after a short period.

Run as backgroud process The flusher has to be able to run as a background process and not require additional configuration during its lifetime.

Parameterless The flusher should be able to detect its own network parameters, complying with the 'set-it-and-forget-it' method.

4.2 Data capture

4.2.1 Libraries

There are several methods of capturing packets and the best known is perhaps *libpcap*. The data capture libraries provide an easy way to capture packets as they provide an understandable application programming interface (API) for programmers to use. Without libraries the programmer has to use kernel specific APIs in order to capture packets, thus creating an application which is not easily portable to other systems.

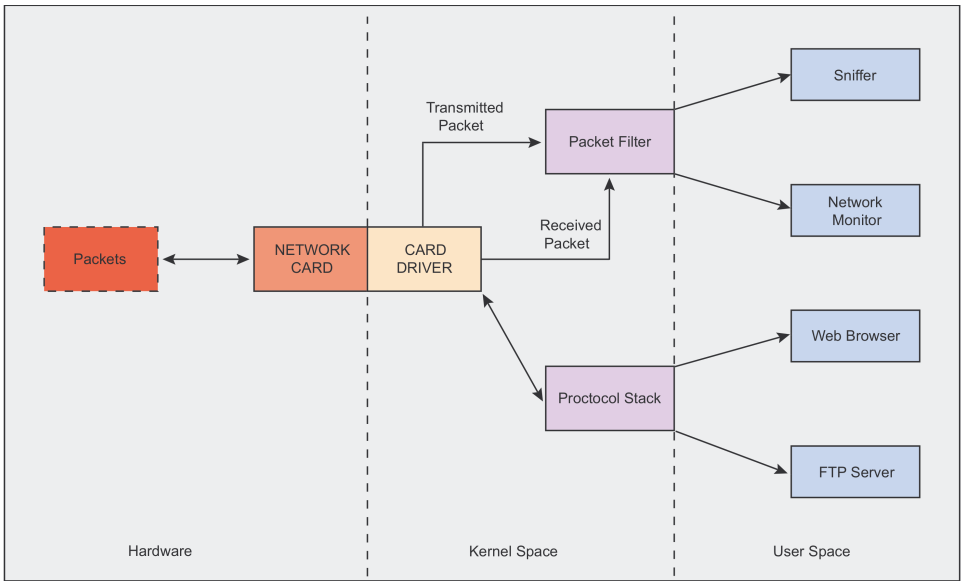


Figure 4.2: Elements involved in the libpcap capture process [13]

Libpcap

Libpcap[1] is a library specifically developed for capturing packets from the network as it is designed to provide packet capture capability to operating systems on the *nix platform (Unix, GNU/Linux, OS X, FreeBSD, BSD, Solaris ...). It is also a version for devices running Microsoft Windows, called *WinPCap*.

Applications like *Wireshark* and *TCP Dump* use *libpcap* for data capture, which makes the library is well known and thoroughly tested.

Libpcap works by hooking into the kernel, which tells the kernel to send a copy of every captured packet to libpcap. A captured packet may be a data packet which is either sent, received or passing through the device as visualised in figure 4.2.

The captured packets get a simple header containing information of the time the packet was captured and the length of the whole packet including headers.

One libpcap's big advantages is the ability to write queries, making the library responsible for returning the right packets and allowing the developer to concentrate on the programming task without paying attention to the kind of packets the library returns.

PF_RING

PF_RING[2] is a packet capture library capable of handling large amounts of data with a low impact on CPU load. It is a less mature library and not as well known as libpcap.

[1] http://www.tcpdump.org/
[2] http://www.ntop.org/products/pf_ring/

PF_RING differs from libpcap in some aspects. PF_RING does not hold additional copies of packets as libpcap do. Because libpcap makes a copy of every packet it captures, there is some overhead and this may cause some packets to be dropped and the responsiveness may suffer.

As opposed to libpcap, PF_RING only supports the Linux kernel, which makes it less portable than libpcap as applications using PF_RING cannot easily be ported to other *nix operating systems. It is possible to provide fallback to libpcap as there is a PF_RING enabled version of libpcap, making it possible to use the libpcap API and simultaneously get the speed of PF_RING on supported devices.

Netmap

A third packet capture library is called Netmap[3], it has more in common with PF_RING than libpcap as it is designed to provide a way to do high speed packet capture. It was originally developed for FreeBSD, but there is also a Linux port available.

There is a netmap-libpcap library available for easy porting of existing libpcap applications.

4.2.2 Implementation

The queue flusher uses *libpcap* for packet capture as it is the library which is supported on most platforms and makes porting the application to other systems easy.

Because TCP packets are the only packets the queue flusher is interested in, it runs a check on the packet type. If the captured packet is a TCP packet it is processed by the application, if not, the packet is ignored. When a packet is known to be a TCP packet, the flusher checks the packet details against its internal list of connections it has captured packets from. If the connection is found, the packet is appended to the list of packets captured from that connection, otherwise it adds a new connection item and adds the packet to that connection.

The list index is a hashed function using *destination IP*, *source IP*, *destination port* and *source port* as input parameters for the hash.

For each captured packet, two hash keys are made for the packet based on the stream details: the sender and receiver IP and the ports used. One 'original' and one 'inverted' key are made.

If the current packet is determined to be an ACK (by looking up the ACK number and checking if the ACK flag [see table 2.1 on page 8] in the TCP header is set) the ACK is looked up in the list of sent packets for the connection by using the inverted key, in order to do a lookup in the reverse stream. If the packet matching the ACK is found the RTT is calculated and all preceding packets (including the one matching the ACK) are deleted, as shown in figures 4.3 and 4.4 on page 38. Regardless of whether the packet is an ACK or not, the packet is added to the list of captured packets for the

[3]http://info.iet.unipi.it/~luigi/netmap/

Listing 4.2: Control loop

```
const NUM_SAMPLES = 10

function controlLoop()
begin
    while (packet = getPacket(nic)) != NULL  do
        processData(packet);

# If the flow exists, find the minimum rtt for the current
# sample and flood
        if (flow = processAck(packet)) != NULL then
            for i = 0..NUM_SAMPLES-1 do
                maxThroughput = max(maxThroughput,
                                    flow->samples[i].throughput);

                maxRtt = max(maxRtt, flow->samples[i].rtt);
                minRtt = min(minRtt, flow->samples[i].rtt);
            loop

# If minRtt exceeds a threshold and it is above 50 ms,
# increase sample counter
            if (sec(minRtt) > threshold*sec(flow->baseRtt) and
                millisec(minRtt) > 50) or
               sec(flow->ackRecvd - flow->lastFlood) > 30 then

                flow->nsamples2++;
            elseif flow->nsamples2 > 0 then
                flow->nsamples2--;
            endif

# If number of samples have exceeded 2; flush
            if flow->nsamples2 > 2 then
                root->nsamples2 = 0;
                flood = true;
                args = {
                    flowRoot = flow,
                    maxThroughput = maxThroughput,
                    maxRtt = maxRtt,
                }
                flood(args)
            endif
        endif
    loop
end
```

sender-receiver-stream using the calculated key. For each ACK captured, the packet information is logged in order to be used when creating graphs.

If the captured ACK has no corresponding packet, it is assumed that the packet has already been acknowledged and it is removed from the packet list. Thus, the ACK is erroneous and it is ignored.

Listing 4.3: Process packet

```
function processData(packet)
begin
    if flow = getFlow(packet->flow) == NULL then
        flow = addFlow(packet->flow)
    end

    flow->seq == flow->seq || packet->seq;

    if isSyn(packet) or isFin(packet) then
        packetSize=1;
    end

    if !isAck(packet) then
        flow->recvWndSize = calcWndSize(packet);
    end

    seq = packet->seq;
    flow->lastAck = nextAck;

    if (item = getPacket(flow, seq + packetSize)) == NULL then
        item = addPacket(flow, packet);

    item->timeRecvd = packet->time;
    item->seq = seq;
    item->plen = packetSize;
    item->totlen = packetSize + packet->headerSize;
end
```

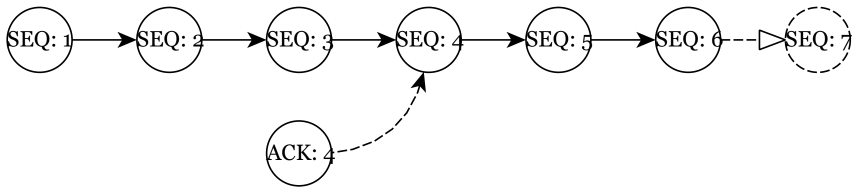


Figure 4.3: Packet list before deletion

4.3 Calculating round-trip time

The round-trip time is calculated when an ACK is received. As TCP does not necessarily send an ACK for each packet received (as discussed in section 2.2 on page 7) the flusher needs to take this into consideration when

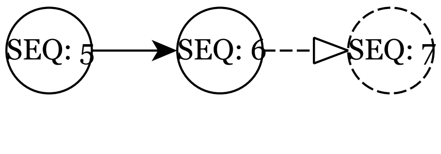


Figure 4.4: Packet list after deletion

accepting ACKs and calculating the RTT.

When an ACK is received, the corresponding packet is found by matching the ACK number to the sum of a packet sequence number and packet length, as the ACK number is the next byte segment that the receiver expects to receive, and the RTT is calculated.

For each received ACK all packets up to and including the packet corresponding to the ACK, are deleted, using only the youngest corresponding packet when calculating RTT.

For example, if packets 1, 2, ..., 6 are sent, an ACK for packet 3 will cause packets 1...3 to be deleted and the RTT is calculated by subtracting the timestamp of the ACK from the timestamp of packet 3, thus creating an RTT sample. When calculating the RTT the exponentially weighed moving average (EWMA) is used, which creates a moving average based on every sample.

If packet 3 is sent several subsequent times, which indicates a dropped packet, the packet representation at the flusher is updated with the new timestamp. This ensures that the RTT is correctly calculated when the ACK arrives, in addition to limiting the memory footprint of the flusher. This enables running on devices with limited memory capacity.

4.3.1 Exponentially weighed moving average

The exponentially weighed moving average is known from statistics as a method for calculating an average value based on several values. As it only depends on two variables and some constants, the exponentially weighed moving average (EWMA) does not require a lot of memory in order to be computed.

EWMA is defined as in equation (4.1) where α denotes the weight of the variables. The sum $\alpha + (1 - \alpha)$ is always 1 when $\alpha \in [0, 1]$, thus the variable α denotes the speed the average increases or decreases based on the data.

$$Y_n = \begin{cases} X_1 \\ \alpha \cdot Y_{n-1} + (1 - \alpha) \cdot X_n & \text{if } n > 1 \end{cases} \tag{4.1}$$

The TCP uses $\alpha = 7/8$ [34] which gives significantly more precedence to the older RTT values than the newer ones. As the α value is known, the queue flusher uses the same EWMA value to calculate the current stream RTT.

4.3.2 Base round-trip time

In order for the flusher to know when it should try to flush a queue, the base RTT needs to be calculated and monitored as the conditions of the network dictate how the RTT behave. If a connection has a base RTT of 30 ms and it suddenly changes to 50 ms, the flusher might cripple the throughput and cause more problems, not solving them. This highlights the need for updating the base RTT at set intervals.

Listing 4.4: Throughput calculation

```
function calc_throughput(flow, item, time_)
begin
    flow->numBytes += item->totlen;
    double t = sec(clock_diff(time_, root->throughputTime))
    if (t >= 1) {
        root->throughput = 2*root->numBytes*8/t;
        root->numBytes = item->totlen;
        root->throughputTime = time_;
    }
end
```

The base RTT is the minimum RTT recorded in the last segment of time. For each RTT sample that is lower than the current base RTT it is updated. As mentioned before, the base RTT in the network might change, but it is a rare event. Due to the event of path change in the network and a rise in base RTT being rare, the base RTT would only need to be updated once in a while.

RFC 6817, which describes the LEDBAT algorithm, uses 10 minutes [30, 31] as a set interval. Similarly, the base RTT of the flusher gets a forced update each 10 minutes.

4.4 Calculating throughput

The throughput is calculated by summing the amount of data sent during an interval and dividing it with the length of the interval in seconds. The calculation is shown in equation (4.2), where n is the number of packets, $payload_p$ is the payload of packet p in bytes and $interval$ is the set interval.

$$throughput = \sum_p = 1^n \frac{length(payload_p) \times 8}{interval} \qquad (4.2)$$

The throughput implementation is as shown in listing 4.4, where the packet payloads are summed and divided by the current RTT.

The reason for dividing the number of bytes with the $interval$ is that the stream and the capture interval does not match completely. With an interval of 1 second, the throughput is calculated when the difference between the current timestamp and the timestamp at the start of the interval is >= 1, thus it should update when the interval is as near 1 second as possible.

The throughput calculation is done when the flusher receives an ACK.

When the link is shared by other connections, the throughput decreases because fewer packets are sent during a specific interval.

The reason for calculating with 1500 bytes is that it is the maximum payload size of an Ethernet frame.

Listing 4.5: RTT calculation

```
function calc_rtt(rtt, rttSample)
begin
    rtt = (1-7/8)*rtt + 7/8*rttSample;
end
```

4.5 Flushing

Flushing is the operation where the application (the queue flusher) thinks it has discovered a queue and tries to empty it by 'flushing' in order to get an increase in network performance, while not having a big impact on throughput.

4.5.1 Determine when to flush the network connection

As ACKs are captured, the current RTT of each ACK is recorded. When an ACK arrives, the current RTT and throughput is added to an internal buffer of size 10. This internal buffer is a buffer of recorded samples, and its size is called the *sample size*.

For each sample recorded, a counter is incremented by 1. If the packet is captured during a flush, the RTT and throughput are not added to the sample buffer and the counter is set to 0.

When the buffer is filled (i.e. the counter have returned to 0, using a round robin buffer) the stored values are analysed.

The bandwidth delay product (BDP) (see equation (2.1) on page 8) dictates that a link is able to transport a given number of bytes at any time. Under the assumption that the BDP is constant, an increase in throughput will give a decrease in RTT. If however the RTT increases without a decrease in throughput then there is a queue in the network as packets accumulate faster than they are transported from the bottleneck, thus giving a higher RTT which increases for every ACK arriving.

If the sample buffer (of size 10) is filled and the lowest RTT in the sample buffer is higher than $threshold \times$ base RTT and the lowest RTT is higher than 50 ms, a counter is incremented. When the counter is incremented more than 2 consecutive times, the flusher is activated, as described in listing 4.6 on the next page.

The reason for measuring more than 2 consecutive filled buffers is that the flusher may be flushing. If it is flushing, it is not wanted to start another flush because that could harm the throughput in the network. Also, it makes it easier to increase or decrease the amount of RTT samples without increasing the memory footprint of the flusher.

4.5.2 Flushing

The flusher is a simple algorithm. The input parameters are the highest throughput measured and the destination of the packets.

Listing 4.6: Process ACK

```
function processAck(packet)
begin
    if !isAck(packet) then return;
    if packet->srcIP == localIP then return;

    if (flow = getFlow(packet)) == NULL then return;

    flow->sendWndSize = calcWndSize(packet);

    ack = packet->ack_seq;

    if (item = getPacket(flow, packet)) == NULL then return;

    item->ackRecvd = packet->time;

    rttSample = item->ackRecvd - item->timeRecvd;

    calc_rtt(flow->rtt, rttSample);
    calc_throughput(flow, item, item->ackRecvd);

    globalTimeBase = globalTimeBase || item->ackRecvd;


    // Update interval as described in RFC6817
    updateDiff = 600;

    if sec(item->ackRecvd - flow->lastBase) > updateDiff then
        flow->baseRtt = rttSample;
        flow->lastBase = item->ackRecvd;
    else if rttSample < flow->baseRtt then
        flow->baseRtt = min(flow->baseRtt, rttSample);
        flow->lastBase = item->ackRecvd;
    endif

    flow->maxRtt = max(rttSample, flow->maxRtt);

    // If not flooding, then add throughput and rtt samples
    if !flood then
        flow->samples[flow->nsamples].rtt = flow->rtt;
        flow->samples[flow->nsamples].throughput = flow->throughput;
        flow->nsamples = (flow->nsamples +1) mod NUM_SAMPLES;
        flow->ok = true;
    else
        flow->nsamples = 0;
        flow->ok = false;
    endif


    return flow;
end
```

As the flusher starts, it tries to determine the right amount of data to be sent by using the highest measured throughput and starting with an RTT value of $2 \times$ base RTT in order to calculate the BDP which is used as a

Listing 4.7: Flooder code

```
function flood(args)
begin
        currentRoot = args->flowRoot;

        throughput = currentRoot->maxThroughput;
        rtt = currentRoot->floodRtt || 2*currentRoot->baseRtt;

        currentRoot->lastFloodRtt = currentRoot->lastFloodRtt ||
                                    currentRoot->baseRtt;

        // Always increment the multiplier
        multiplier = 1.2;

        currentRoot->lastFloodMax = max(currentRoot->lastFloodMax, item->maxRtt);

        rtt = multiplier*rtt;

        // baseRtt <= rtt <= maxRtt
        currentRoot->rttFlood = max(rtt, 2*currentRoot->baseRtt);
        currentRoot->rttFlood = min(rtt, 2*currentRoot->maxRtt);

        numberOfPackages = throughput*rtt/(8*1500);

        for p = 0..numberOfPackages-1 do
            send(args->recipient);
        loop
end
```

measure on the amount of UDP packets to send. For each subsequent flush, the RTT value is raised by 20%. I.e. the previous RTT value is multiplied by 1.2. An upper limit for the RTT value is set to 2 × max RTT. This should be enough for the flusher to successfully 'flush' a queue, even if some of the packets are dropped before the problematic buffer.

If a flush has not happened during the last 30 seconds, the flusher forces a flush. The 30 second barrier is arbitrary but necessary in case the connection has a hugely oversized buffer where data may accumulate and create a delay of several seconds. Thus, the maximum time to wait is 30 seconds, which should be more than good enough for different network base RTTs. It may be too high as links with more than 1s RTT are rare unless the link is relayed by a satellite.

4.5.3 Heuristics for enabling the flusher

The reason for the 50 ms base RTT requirement is to prevent unnecessary flushing. If there is no lower base RTT limit, the flusher may be activated even though there is no active queue. Higher RTT measurements may be the result of delayed ACKs and not only queues. Delayed ACK is a TCP functionality that avoids sending unnecessary ACKs [8] (section 2.2 on page 7.) An ACK may be delayed up to 500 milliseconds as the receiver waits for more data to arrive before the ACK is sent [8]. Due

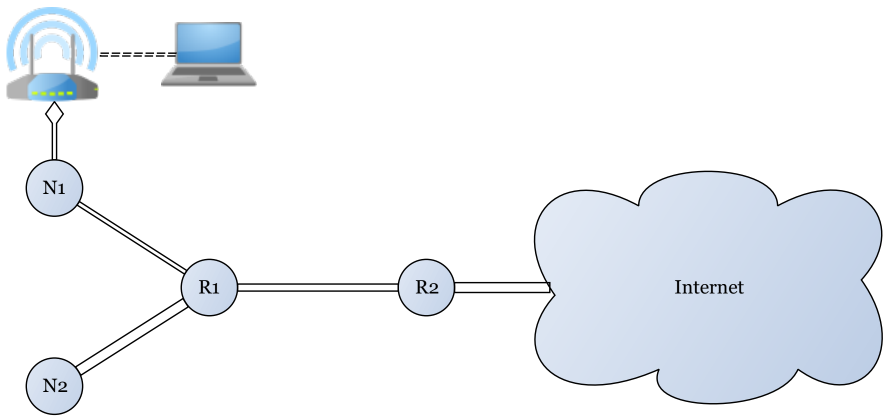


Figure 4.5: A network with multiple bottleneck points and flusher running at consumer router

to this possible delay a minimum RTT should be measured in order to get the flusher to slow down and be less aggressive. An aggressive flusher may lead to a misbehaving network as the flusher may cause queues and increased delay in the network, if the amount of packets is miscalculated. The queues form due to the amount of packets sent from an application which is non responsive to packet drops. A miscalculation of the needed number of packets may be caused by the BDP calculation which is $P_{flood} = {}^{throughput_{max} \times rtt_{max}}/_{8 \times 1500}$, this leads to numerous packets being sent if the throughput sample is extreme compared to the other throughput samples. It is therefore better to use the maximum value of the EWMA throughput and EWMA RTT as the maximum values may not be as extreme due to the inherent smoothing in the EWMA calculation.

4.6 Flusher placement

As displayed in figures 4.5 to 4.7 on the current page and on the facing page there are several places in the network where queues may occur. Queues can occur at every link in the figures. If there is an application in the N2-network which is transferring a lot of data, or otherwise creating a queue, the queue will appear in R1 as it is the slowest point. A queue at this point will impact the data flow to and from the N1-network.

As AQMs is not implemented or enabled in a lot of hardware, the queue flusher may be a temporary solution. If the flusher is running at N1 or N2 as shown in figures 4.6 and 4.7 on the next page, the queue in R1 should be dealt with as it is the bottleneck which queues packets that delay every stream going through the link.

The flusher is displayed by a diamond in the connection between a link and a node.

In figures 4.5 and 4.6 on this page and on the next page the queue in R1 is not dealt with because the N1-R1 link is slower than the N2-R1, this

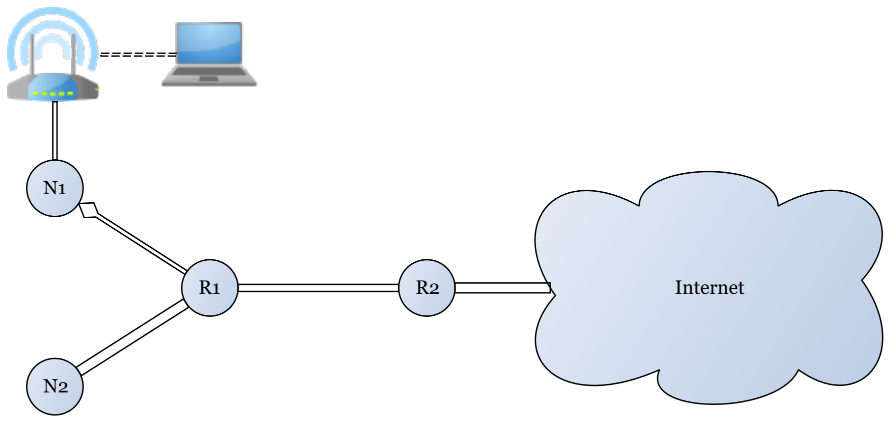


Figure 4.6: A network with multiple bottleneck points and flusher running at N1

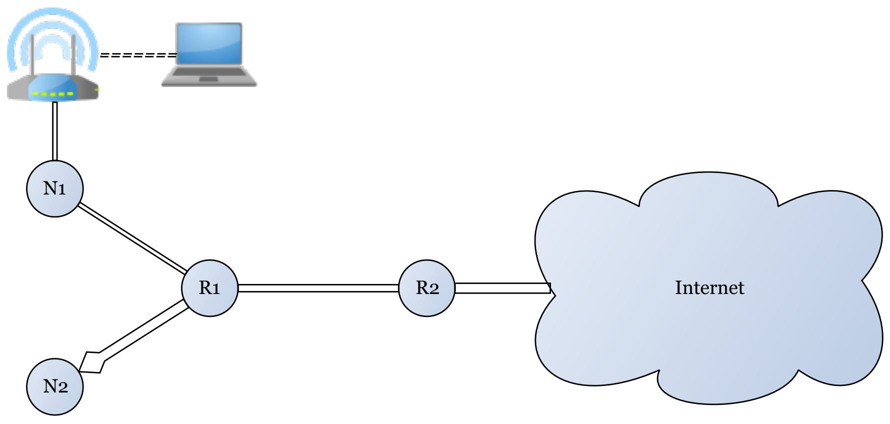


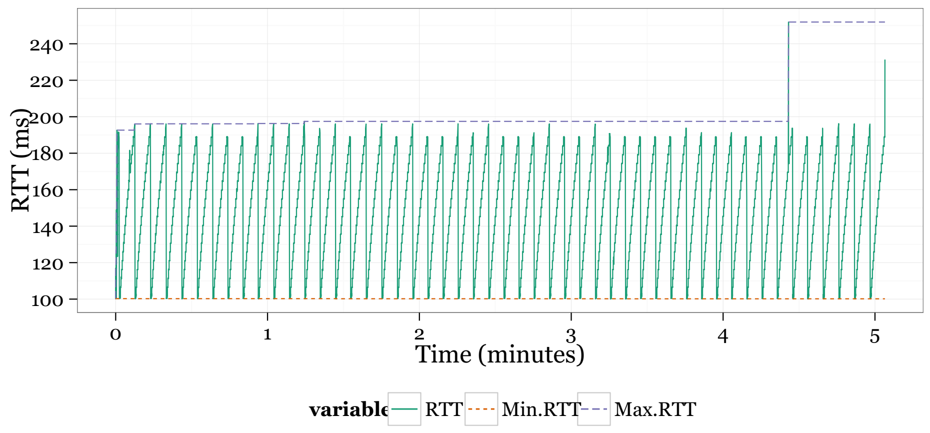
Figure 4.7: A network with multiple bottleneck points and flusher running at N2

the link bottleneck is at N1. Even though the queue in N1 is flushed, the possible queue at R1 is not dealt with due to the flusher not detecting the queue because of its position in the network. If, however, both N1-R1 and N2-R1 links have the same bandwidth and base RTT, the bottleneck will be at R1 and a flusher running at N1 or N2 is equally likely to detect and flush the queue.

4.7 Flusher operation

Figures 4.8 and 4.9 on the following page and on page 47 shows the flusher's logs of RTT and throughput during runs where the flusher is passive and active.

In the graphs that show the calculated throughput there are spikes which are a bit higher than the network bandwidth. The reason for those



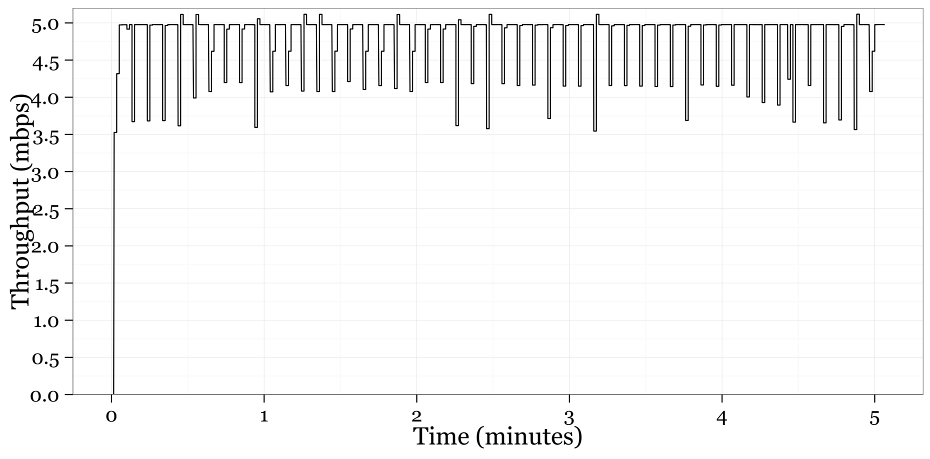
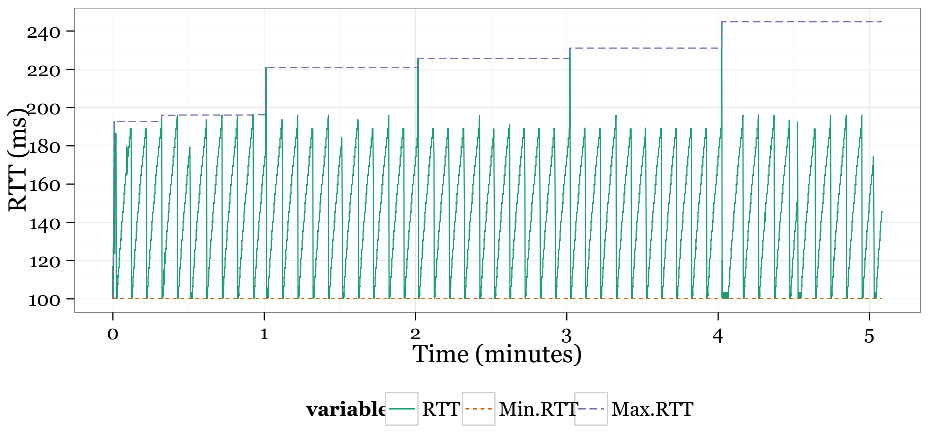


Figure 4.8: Graphs created from logs from a passive flusher. The connection has a 50 packet buffer with a base RTT of 100 ms.

spikes may be that some extra packets have been counted during that specific interval. The flusher updates its throughput each time it gets an ACK. For every ACK where a corresponding data packet is found, the throughput is calculated. When the difference between the interval start and the latest ACK timestamp is greater or equal to the interval, a new interval is started and the throughput is calculated based on a sum of the sizes of the packets ACKed during the last interval. The throughput is set by: $throughput = {}^{bits}/_{time}$, where $bits$ is the sum of ACKed packet sizes and $time$ is the time difference between the first and last packet in an interval.

Testing showed that the frequency of throughput spikes rises when the base RTT is lowered. The spikes may be the result of jitter, as some packets may come to the receiver earlier than expected. If such an event occurs, the flusher may calculate a higher throughput because of more ACKs received in a short period.

Jitter is the change in delay between each received packet. Even though the packets are sent with a constant delay, the inter-packet delay may have changed when the packets arrive at the receiver. The changes



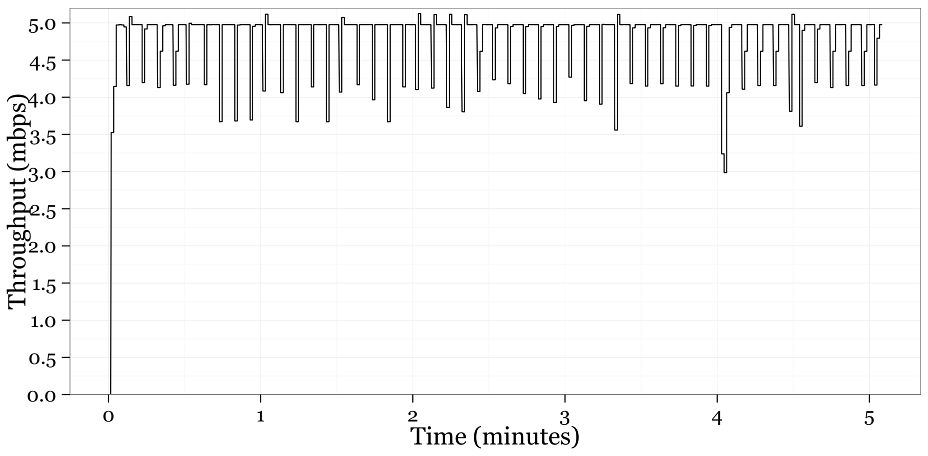


Figure 4.9: Graphs created from logs from an active flusher. The connection has a 50 packet buffer with a base RTT of 100 ms.

in inter-packet delay are known as *jitter*. Jitter has the most effect on real-time traffic such as audio streams between participants in an online chat room. Jitter may not be noticed when transferring large files, its effect is most noticeable for traffic which demands a highly responsive network.

Jitter is a naturally occurring effect in packet switched networks because of the changing RTT. When packets are queued, the RTT rises. When the queue drains there may be small variations in the draining rate. This causes variations in the inter-packet delay, and is presented as jitter at the receiver. The jitter is then transferred to ACKs and cause small variations in the flusher's throughput calculation.

The throughput jitter is shown in figures 4.8 and 4.9 on page 46 and on the current page. It is not frequent in none of the graphs, which shows that it is not necessarily an effect that causes trouble. A slightly higher throughput is not a problem for the flusher. Neither the spikes in RTT or the spikes in throughput should effect the flusher negatively, unless the RTT suddenly have spikes which is double the normal level. If such spikes occur, the flusher intensifies the flushing as more packets are sent in order to flush the queue.

Even though the throughput jitter occurs more often on connections with a lower base RTT, the variations is too small to do a big difference.

When the flusher flushes a queue, it sends a certain amount of packets towards the receiver. As it uses the maximum measured throughput and the maximum measured RTT in the BDP calculation, a spike doubling the maximum RTT would double the amount of packets sent. This may cause the throughput for a connection to be lowered for a longer time as there are more UDP packets to send.

Such spikes do not occur frequently, but they occur. The reason is the flusher RTT calculation as the spikes disappears when the same connection is monitored with other programs.

Figure 4.9 on page 47 shows the effect of a flush with too many packets. At the 4 minute point the RTT spikes and it is followed by a short period of low RTT.

4.8 Known problems

As mentioned in the previous section, a problem with the flusher is that its RTT calculation sometimes calculates a higher RTT. It does not occur frequently, and it is probably caused by the flusher failing to find the correct packet even though it exists in its internal lists.

Part III

Evaluation

Chapter 5

Testing

5.1 Testbed

In order to test the queue flusher, a local testbed was used. There are several ways to configure a testbed, ranging from testing on physical hardware to virtual machines running on a single computer.

5.1.1 Layout

There are several requirements for how the layout of the testbed should be.

Client a client machine for sending data.

Server a server for receiving data.

Router at least one router to route the data between the client and the server.

Bottleneck the testbed should feature at least one bottleneck in order to slow down the flow and create a queue.

There are several possible configurations which meet the requirements, and a minimal topology such as the one displayed in figure 5.1 on the next page is sufficient for testing and verification of the queue flusher functions. The topology displayed meets all the set requirements as it contains a bottleneck, a server, a client and a router.

Testing the queue flusher is an important task as it is the only method to assure that the application is working correctly. There are several ways to test the application. A network is required for testing, either real network using real, physical hardware, a virtual network with virtual hardware or a combination.

When using a network for testing, a homogeneous network is the best option for assuring good performance. The reason for using a homogeneous network environment is that we have full control over the network and its parameters during testing. If the tests were run using a real network, the results would have been realistic but it would have been difficult to detect bad behavior from the application as networks are heterogeneous due to the number of different configurations.

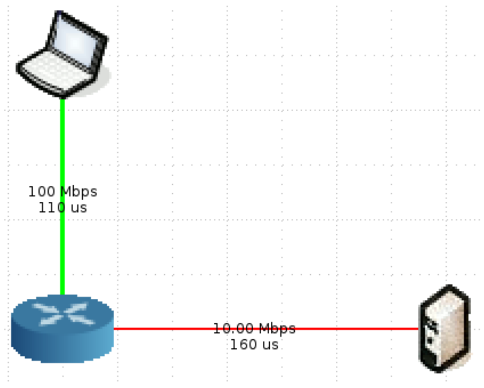


Figure 5.1: Minimal topology

5.1.2 Real hardware

Testing on real hardware would produce the most realistic results as the data had to be sent from a physical machine, over a link to a router and forwarded to the server. As each piece of hardware would induce its own latency, the inherent latency in the network would be constant if not other means of delaying packets were implemented on the devices.

The downside of using real hardware is that it is bound to a certain geographical place and all testing would have been done in the lab where the hardware are. Unless one of the machines are connected to the internet, all testing would have to be done in one place.

5.1.3 VM

As opposed to real hardware there is virtual hardware such as virtual machines which runs on a host computer. Because virtual machines run on a host computer the testbed is fully portable and the virtual machines need special hardware support in order to work correctly. Due to the nature of virtual machines and the emulation of specific hardware, the machines run a bit slower, thus it cannot be fully real time, even though it is close.

A fully virtual machine emulates a full hardware stack, from the CPU to the display. When running a machine which has the same architecture as the host machine, the virtual machine gets near native performance if the host hardware supports virtualization.

5.1.4 Containers

An alternative to fully virtualized machines are containers. Containers are lightweight machines with a minimal implementation in order to run services isolated from the host. Each container has a dedicated network stack and share the rest with the host machine. A container provides an independent namespace in order for the processes to be isolated from the host. An application running inside a container sees the container as a fully functioning machine with no other software running.

Containers are available for several operating systems. Linux have Linux container (LXC) and FreeBSD have *Jails*. Both LXC and Jails run processes in isolation from the host and they have a fully functioning network stack.

As containers are not fully virtual machines they are able to run on lesser hardware as they have a smaller footprint and does not require virtualization abilities on the host.

5.1.5 Common open research emulator

Common open research emulator (CORE) [2, 23] is a toolkit for creating testbeds distributed over one or several machines. The toolkit consists of a collection of tools, a graphical user interface (GUI) for easy management of virtual machines and paths, a daemon which communicates with the host and sets up the network topology described using the GUI.

The CORE project has the following features [2, 23]:

- Network lab in a box
  - Efficient and scalable
  - Easy-to-use GUI canvas
  - Centralized configuration and control
- Runs applications and protocols without modifying them
- Real-time connection to live networks
  - Hardware-in-the-loop
  - Distributed with multiple COREs
- Highly customizable

In addition, CORE ships with its own container implementation. The Linux version uses chroot for the containment of applications and netem for restricting bandwidth, and the FreeBSD version uses Jails and the bandwidth restriction capability found in the standard firewall packet for FreeBSD, IPFW [23].

The CORE GUI makes it easy to create a larger network containing links with different bandwidths. If CORE is installed on several connected machines, the emulated network may span every single host. This may be useful if there are many nodes needed in the emulated network. One of the advantages using CORE is the ability to easily work with different technologies, as CORE is able to emulate both wired and wireless networks with nodes disconnecting and reappearing. This is useful for studies of how the network topology works with moving nodes (e.g. a GSM network). It is also possible to connect some nodes to the NS-3[4] network simulator in order to apply more complex calculations and statistical analysis to links in the network [2].

[4]http://www.nsnam.org/

5.2 Simulation

In order to be able to have control over the available link bandwidth and latency there are some available tools. For Linux the queueing discipline is the standard way of limiting bandwidth, and in conjunction with netem make it possible to limit both bandwidth and latency for links. Some qdiscs are advanced enough to prioritize different kinds of traffic and delay others.

FreeBSD has taken another approach by implementing IPFW, a firewall with functionality for throttling incoming and outgoing connections.

Both qdisc and IPFW are capable of limiting the outgoing queue length, bandwidth and latency relatively easy. IPFW is the only one that, with relative ease, is also able to restrict incoming connections without doing heavy modifications by loading additional kernel modules.

5.2.1 Linux: Queueing Disciplines

Hierarchial token bucket

Hierarchial token bucket (HTB) is one of the qdiscs available in Linux. It is used for limiting link bandwidth by defining classes and defining how much bandwidth a class gets. HTB is a classful qdisc which uses the token generation algorithms used in token bucket filter (TBF).

HTB works, as TBF, by matching packets to available tokens. Tokens are generated at time intervals according to the preferred bandwidth. When the tokens are generated at the same speed as incoming data arrives each piece of data is matched to its token and the data is forwarded. If the incoming flow of data is slower than the token generation, the data are forwarded. When the data flow is faster than the token generation, some data segments will not get a token and will be dropped.

The HTB can be used together with other qdiscs making bandwidth throttling more defined. Using HTB and netem, an emulation of a slower link is possible to achieve by limiting bandwidth with HTB and adding delay with netem.

5.2.2 FreeBSD: IPFW/Dummynet

IPFW is the FreeBSD answer to qdisc and *netem*. It is a firewall with features for network emulation. The emulation features include queue length and bandwidth for incoming and outgoing links. Thus, it is possible to have a machine functioning as a bottleneck by limiting the outgoing connection. The main difference between IPFW and qdisc is that IPFW works on IP ranges whereas the qdisc works on network interfaces.

Being able to work on IP ranges the IPFW could be set up to delay traffic from several network interfaces by configuring which IP ranges should be used.

5.3 Data generation and logging

5.3.1 iperf

Iperf is a cross-platform software for testing the performance in a network using TCP or UDP streams. Because iperf can act as both server and client it inhabits some interesting properties. For TCP it has an option for choosing congestion control for the specific stream, which makes it a good choice when testing and studying different congestion controls and queue management.

There are several options available when configuring iperf, such as the possibility to set a timer. When the timer is set, iperf tries to get the transfer finished as near the time as possible. If there are data which have not got an ACK the transfer will be going until the ACK is received, thus some tests might go on for longer than the set time.

During, and at the end of, a transfer iperf logs the current bandwidth measurement and the amount of data transfered since the start. In addition, it is able to have several streams going from client to server, both parallel and reverse (meaning the server is the sender and the client is the receiver).

In 2009 a complete rewrite of iperf (called iperf3) began from scratch with the goal of reducing and optimising the code. The first major release of iperf3 was made in January 2014 [40].

5.3.2 Web10G

The Web10G project is the successor for the Web100 project. Both projects had the goal of implementing diagnostics for TCP streams as defined in RFC 4898 [20].

The Web100 project works for older Linux kernels and the development has stopped. Web10G is the successor of Web100 and it is in active development and new versions are made as new versions of the Linux kernel are made available.

Certain information about TCP connections is hidden from the user. Each TCP connection logs a lot of different metrics in order to provide optimal performance. Metrics such as `cwnd` size and `rtt` are usually hidden from programmers and the Web100 and Web10G projects are working on making a method for accessing the metrics.

As Web10G and Web100 are independent projects they need to be patched into the Linux kernel in order to be used. This means compiling a new kernel from scratch with the patch applied, which is a bit time consuming.

5.3.3 TCP Dump and Synthetic packet pairs

TCP Dump[5] is a utility that is able to dump the packets it is capturing, to file. The file consists of the raw packet data that TCP Dump captured. As

[5]http://www.tcpdump.org/

TCP Dump uses libpcap (section 4.2.1 on page 35) it is possible to define which packet types it should capture. When used in this thesis, TCP Dump is set to dump TCP packets only.

Synthetic packet pairs (SPP)[6] is a utility used to calculate the RTT based on captures from TCP Dump. For calculations SPP needs some information about the connection, mainly the sender's and receiver's IP addresses and the TCP dumps for both the sender and receiver. The IP addresses are used to filter out the 'correct' packets and calculate the connection RTT.

5.3.4 TCP Flooder

Starting with the release of Linux kernel version 2.4 [35] there is a way for retrieving information about a TCP connection by using its internal socket reference. The data is returned using the `struct tcp_info` (listing 5.1 on the next page) that contains information about window sizes, maximum segment sizes, sent and received ACKs, RTT etc.

Because this functionality exists in the Linux kernel there is no need for patching the kernel to get a subset of the functionality Web10G provides. As the Web10G provides an API for user space applications to use, an extra application would be needed to log the congestion window size. Because the Linux kernel provides the wanted data in an easier way, the information could be retrieved when flooding the network.

The custom software that was made in this thesis as a way to test the network is called *TCP Flooder* as it send data as fast as it is able to using TCP. Thus floods the network and fills buffers. Some if its features it shares with iperf, such as the ability to work as server and client, and a customisable congestion algorithm. It also supports logging of `struct tcp_info` to file.

For each packet sent, it pulls information from the socket and writes it to a file. The application has currently no way of sending a file or stopping after a given time has passed (which has to be managed externally). The frequent logging causes rather large log files if the *TCP Flooder* is run for a long time because it dumps all the information to a file.

Validating the RTT measurements from the Queue Flusher

The logs from the *TCP Flooder* are handy in several ways not only for logging the congestion window size, but also to check the correctness of the calculated RTT from the *queue flusher* (see section 4.3 on page 38 for more details). RTT calculated with SPP is also added for further reference.

Figure 5.2 on page 58 shows the RTT measurements taken from the flusher and the flooder logs during a run where the flusher is active. As it shows, the RTT measurements from the flusher's log roughly match the RTT measurements from the flooder's log.

In both figures 5.2 and 5.3 on page 58 the dotted vertical line marks the network base RTT, which in this case is 10 ms.

[6] http://caia.swin.edu.au/tools/spp/

Listing 5.1: struct tcp_info [35]

```
        /* Metrics. */
        __u32   tcpi_pmtu;
        __u32   tcpi_rcv_ssthresh;
        __u32   tcpi_rtt;
        __u32   tcpi_rttvar;
        __u32   tcpi_snd_ssthresh;
        __u32   tcpi_snd_cwnd;
        __u32   tcpi_advmss;
        __u32   tcpi_reordering;

        __u32   tcpi_rcv_rtt;
        __u32   tcpi_rcv_space;

        __u32   tcpi_total_retrans;
};

/* for TCP_MD5SIG socket option */
#define TCP_MD5SIG_MAXKEYLEN    80

struct tcp_md5sig {
        struct __kernel_sockaddr_storage tcpm_addr;     /* address associated */
        __u16   __tcpm_pad1;                            /* zero */
        __u16   tcpm_keylen;                            /* key length */
        __u32   __tcpm_pad2;                            /* zero */
        __u8    tcpm_key[TCP_MD5SIG_MAXKEYLEN];         /* key (binary) */
};


#endif /* _LINUX_TCP_H */
```

When studying the graph, one have to take into account that the flusher was started before the flooder, thus, the initial measurements may differ slightly.

As displayed in figures 5.2 and 5.3 on the next page there are some minor differences in the measured RTT, but they are not big enough to invalidate the flusher results. The differences may come from the way the RTT is measured and calculated in both instances. Both the Linux TCP algorithm and the flusher use an EWMA with $\alpha = 7/8$, and there may be some differences in the precision of the measurements. In addition, there may be some minor differences between the runs due to minor variance in the testbed load.

The calculated RTT from SPP differs from the other RTT calculations in that it has a larger portion with a high RTT. This increase in high RTT is probably due to the fact that the SPP manages to calculate a more correct RTT than the flooder and flusher. Graphs showing the RTT calculations from SPP shows the effect of the flusher as large spikes, as shown in figure 5.4 on page 59. The graph shows a run where the base RTT is 100 ms, a run with a lower base RTT would make a graph where the spikes are difficult to see.

Overall, the RTT difference between the flusher and the flooder is

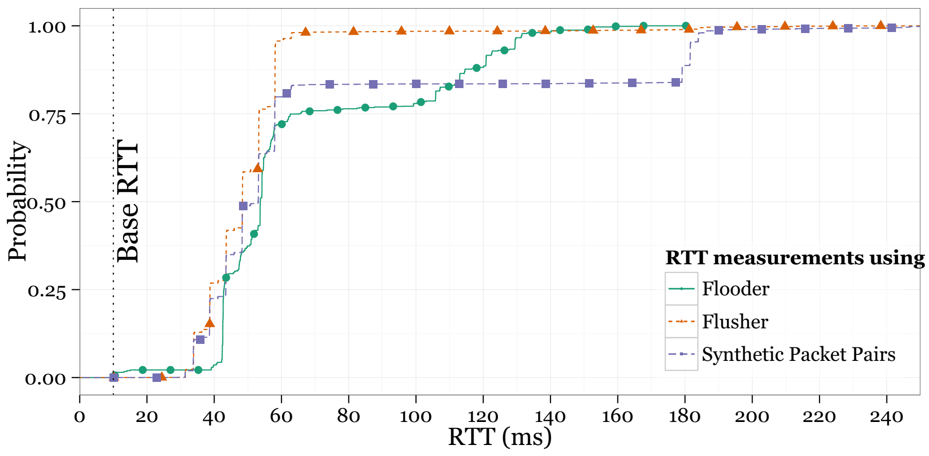


Figure 5.2: Difference between measured RTT and actual RTT with an active flusher.

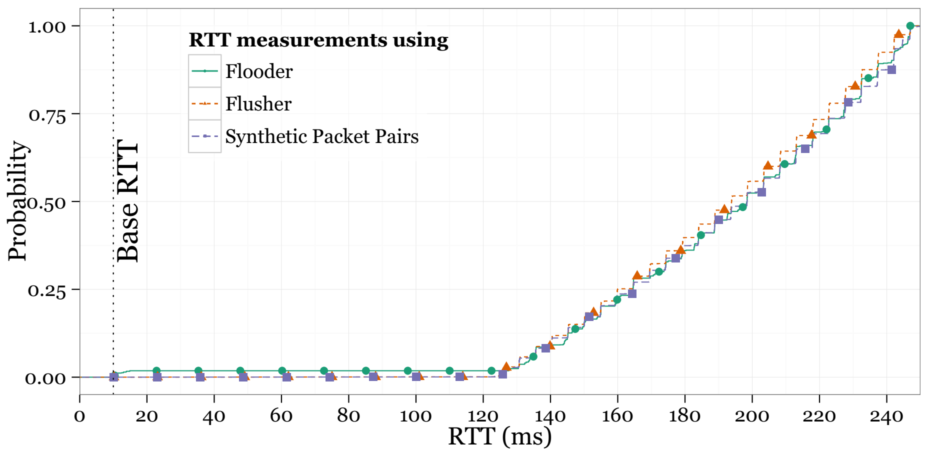


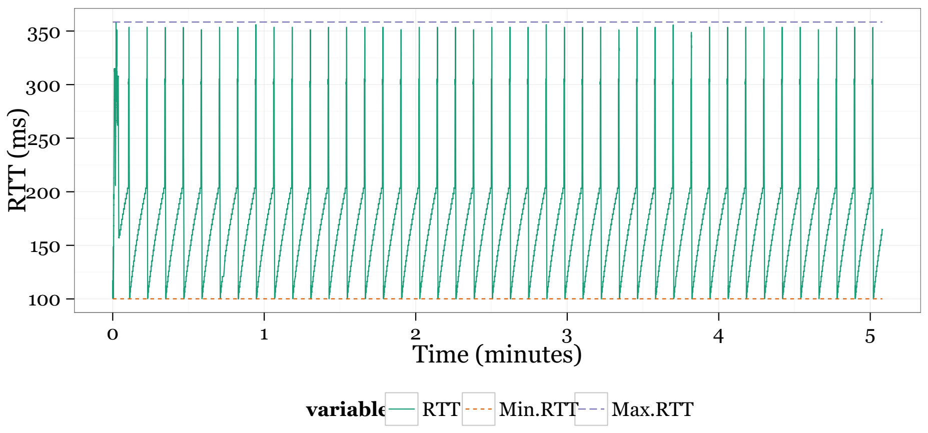
Figure 5.3: Difference between measured RTT and actual RTT with a passive flusher.

too small to ignore the RTT measurements from the flusher. There may always be differences in measured values due to internal differences in the calculation of RTT, even though the measurements ideally should be equal to each other.

5.4 Testing

The technologies used for testing are as described in earlier sections. Using a Linux host which hosts all other software, containers are used as clients connected to a discrete network with its own IP range in order to ensure that the network traffic is not disturbed by passing traffic from other networks.

The container implementation is the one which is bundled with CORE



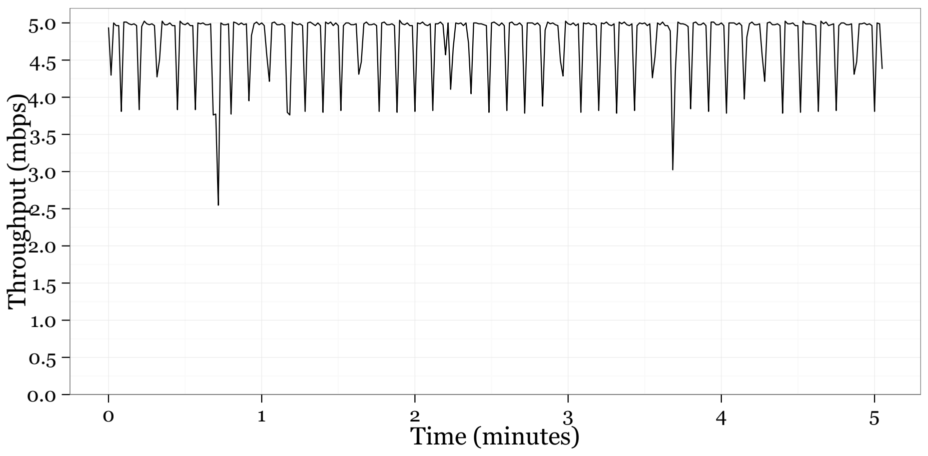


Figure 5.4: Graphs showing the calculated RTT using SPP in addition to throughput calculated from the same TCP dumps with an active flusher and a base RTT of 100 ms.

(section 5.1.5 on page 53) as it is easy to set up and configure. Even though they are separate implementations, they use a standard Linux container implementation as a base, thus the CORE container implementations should work as expected.

For the network emulation, both ipfw and netem are used in separate configurations. The configuration using ipfw is running in a virtual machine due to the way ipfw works. Ipfw is only able to relay data between separate IP ranges, which is not an easy task when it is running on the host machine, thus it is running inside a virtual machine. The virtual machine is running Fedora 18 [12, 38] with the Linux implementation of ipfw. Another possibility is to use a virtual machine running FreeBSD [36, 39] as it has ipfw installed as default.

The other possibility for restricting bandwidth and adding delay is a configuration using both HTB and netem, which is a Linux native and works on links created between virtual machines and containers without having to deal with IP addresses. As netem and qdiscs are native to Linux it is possible to set up a container based network on the host machine which provides less overhead than fully viritualized machines. Thus, the reaction and performance is near real time.

The application used for sending data is the custom made 'TCP Flooder' as it supports logging of the congestion window and `struct tcp_info` in order to get accurate data from the host.

Chapter 6

Results and discussion

6.1 Introduction

This chapter introduces the results of the tests run according to the scenarios introduced in section 6.2.

The graphs displayed in section 6.3 on page 63 are created from data collected during runs where the flusher was running. Some graphs show runs where the flusher is passive, only collecting data, while other graphs display data from runs where the flusher is active. The software used for logging the active connections was TCP Dump and the RTT is calculated by SPP from the dumps generated by TCP Dump. In addition, the throughput graphs are generated by the same TCP dumps as the RTT.

All runs are done according to the scenarios in section 6.2 with different network base RTT. Unless otherwise described, the graphs show runs where the bottleneck buffer is capable of holding a queue with a maximum length of 100 packets (150 KB). The bottleneck buffer sizes range from 10 packets to 100 packets with an increment of 10 packets. The collected data is from runs with the following bottleneck buffer sizes: 10, 20, 30, 40, 50, 60, 70, 80, 90 and 100 packets. Every run, whether it was done with an active or a passive flusher, ran for 5 minutes. 5 minutes should be long enough to simulate a big file transfer, even though a lower time should suffice to check the flusher performance.

6.2 Scenarios

The flusher is run on networks that have the configurations that are presented in table 6.1 on the next page. The scenarios listed are created to see how the flusher is reacting under certain conditions. Every scenario is run using both Cubic and Reno as congestion control, using both netem and ipfw in order to be able to determine if the 'queue flusher' has a positive effect on the RTT. The results are presented and discussed in section 6.3 on page 63.

The network configurations presented in table 6.1 on the next page are selected in order to force a bottleneck queue which the flusher should try to empty. As the network is a 'black box' the queue flusher does only know

Table 6.1: Tests where the fast link is 10Mbps and the slow link is 5Mbps

Fast link (10 Mbps) One-way delay	Slow link (5 Mbps) One-way delay	Total RTT	Theoretical link BDP
20 ms	30 ms	100 ms	41
10 ms	15 ms	50 ms	20
2 ms	3 ms	10 ms	4

how many packets are sent and how many ACKs are received. Values such as RTT and throughput are measured and calculated for each received ACK.

The scenarios are run with the mentioned network parameters and with different flusher thresholds in order to determine which threshold has the best results with regards to RTT and throughput, and how the flusher performs in comparison to existing active queue managers. The function of the thresholds is to determine when the flusher should start flushing. A threshold of $2 \times$ base rtt would cause the flusher to flush when the current RTT is at least $2 \times$ base rtt. The selected thresholds are:

- $1.5 \times$ base rtt
- $1.7 \times$ base rtt
- $2 \times$ base rtt

The BDP values in table 6.1 are calculated as described in section 2.2.4 on page 10, the theoretical BDP is the maximum calculated amount of packets the sender may send before the packets are queued in the buffer.

The theoretical BDP in table 6.1 is the limit for when packets are queued. If the congestion window for a connection increases above this threshold, packets are probably queued. The reason for this is that a TCP connection observes the link from the outside and therefore the congestion window size is $BDP + queue$.

$$\text{pipe size} = \left\lfloor \frac{BDP_{link}}{8 \times 1500} \right\rfloor = \lfloor bandwidth \times rtt \rfloor \tag{6.1}$$

The BDP is a measure of the amount of packets the link is able to transfer in a RTT, this gives a minimum congestion window value in order to judge the results.

The calculated BDP is important and needs to be taken into account. A link with a low BDP results in a more reactive flusher than a link with higher BDP. The BDP is a measure of the amount of packets the link is able to transport. A connection with a low BDP would react quicker to packet loss than a connection with a higher BDP. If the link has a BDP of 4, it takes 3 transferred packets before a connection reacts to a packet loss. A link with a BDP of 41 would transfer an additional 40 packets before a packet loss is detected.

The latencies should represent values which are found on connections in the internet. An RTT of 10 ms is typical when connecting to servers that are geographically close. A higher RTT indicates connections to servers that are further away. When measuring an RTT of 100 ms or more, the servers are either on the other side of the earth, the connection is relayed via satellites or there is some congestion on the link. Thus, the RTTs of 10 ms, 50 ms and 100 ms should represent realistic RTT values.

6.3 Results

Figures 6.2 to 6.7 on pages 65–70 show the differences in measured mean RTT when the flusher is passive and active, i.e. the connection is making a queue and the flusher tries to lower the RTT by creating a short congestion in order to allow TCP to react, and then the queue to drain. Each graph shows box plots of the measured RTTs in that run.

As stated in [37]: 'The upper and lower "hinges" correspond to the first and third quartiles (the 25th and 75th percentiles). The upper whisker extends from the hinge to the highest value that is within $1.5 \times IQR$ of the hinge, where IQR is the inter-quartile range, or distance between the first and third quartiles. The lower whisker extends from the hinge to the lowest value within $1.5 \times IQR$ of the hinge. Data beyond the end of the whiskers are outliers and plotted as points (as specified by Tukey)'. Both the IQR and the percentiles are calculated as according to [21]. An example plot, describing the limits are displayed in figure 6.1 on the next page.

The runs have different queue lengths at the bottleneck in order to get a picture of how the flusher works with different queue lengths.

As the graphs show, the flusher has a significant impact on the mean RTT when it is running. The mean RTT is significantly lowered when the flusher is active as opposed to when it is passive, while the maximum RTTs are equal for a run with a certain queue length. This is because the buffer is filled during the flushes and some packets experience a maximum delay. When the buffer is filled a spike occur in the graph, in the box plot the spikes are represented by the outliers. As the spikes only occur when a certain condition occurs and the flusher tries to flush the queue, there are longer periods with lower RTT. This is the reason for the lower mean RTT when the flusher is active, as opposed to when it is passive.

Figures 6.8 and 6.9 on page 71 and on page 72 show that with the reaction time of TCP Cubic, the connection seems to be responding faster to congestion than a connection using TCP Reno. This is also displayed in figures 6.2 to 6.7 on pages 65–70.

As both figures 6.8 and 6.9 on page 71 and on page 72 display, both TCP Cubic and TCP Reno are able to fill the bottleneck buffer as the maximum congestion window size on both runs are about 110 packets, which is more than four times the link BDP. Table 6.1 on page 62 shows that the appropriate BDP for a link with an RTT of 50 ms is 20 packets. As the congestion window roughly doubles each RTT, the congestion window would be 40 packets within one RTT. 40 packets 'in-flight' translates to a bottleneck queue of 20 packets. If the connection closes, the queue would drain within one RTT. Any higher queue would take longer to drain and create extra delay for concurrent connections. A congestion window of 110

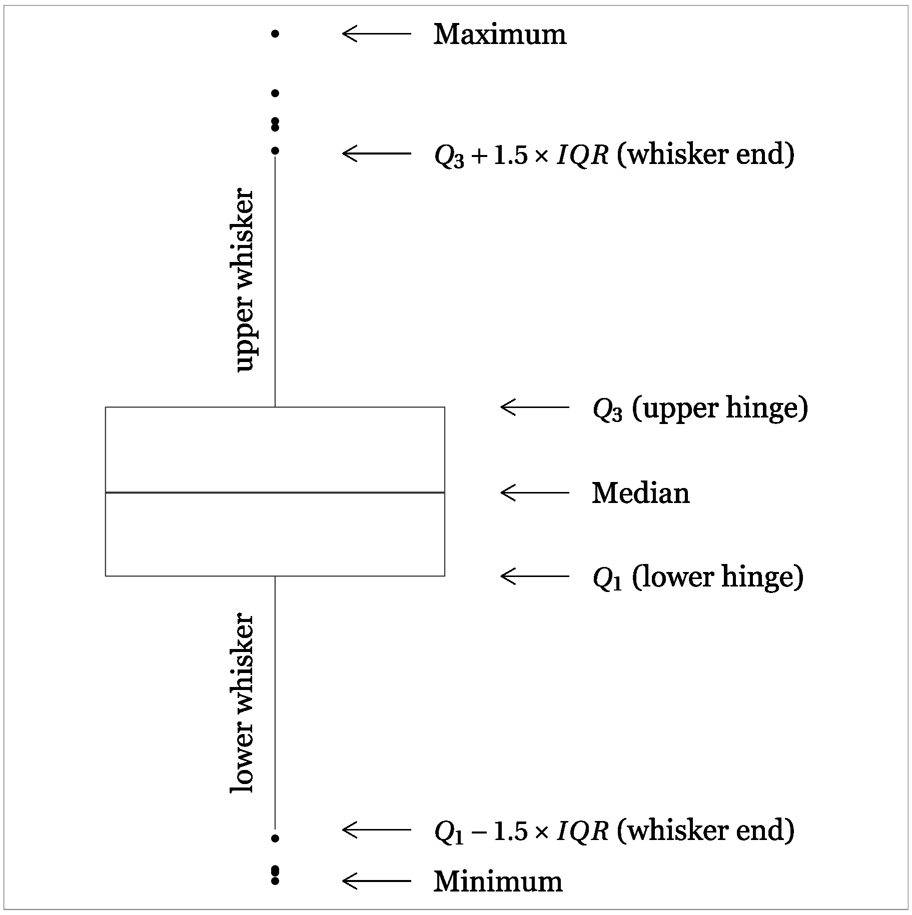


Figure 6.1: Descriptive example of the boxplots used.

packets would roughly translate to a queue of $110 - 20 = 90$ packets. This queue would drain after roughly 216 ms, as shown in equation (6.4). As shown, the time it takes for a queue to drain does not depend on the link delay.

$$delay = \frac{90\ packets}{5\ Mbps \times 15\ ms} \times 15\ ms \tag{6.2}$$

$$= \frac{90 \times 8 \times 1500b}{5 \times 10^6 bps} \tag{6.3}$$

$$= 216\ ms \tag{6.4}$$

The 5 Mbps link with 15 ms delay is the slowest link, which is the link transferring packets from the bottleneck. This link is listed in table 6.1 on page 62. The TCP Cubic congestion window size is a bit higher than the size of the TCP Reno congestion window due to the exponential increase that TCP Cubic uses, as the next congestion window increase is larger relative to the last. Regardless, the highest measured congestion window size when using TCP Cubic seems to be one or two packets more than

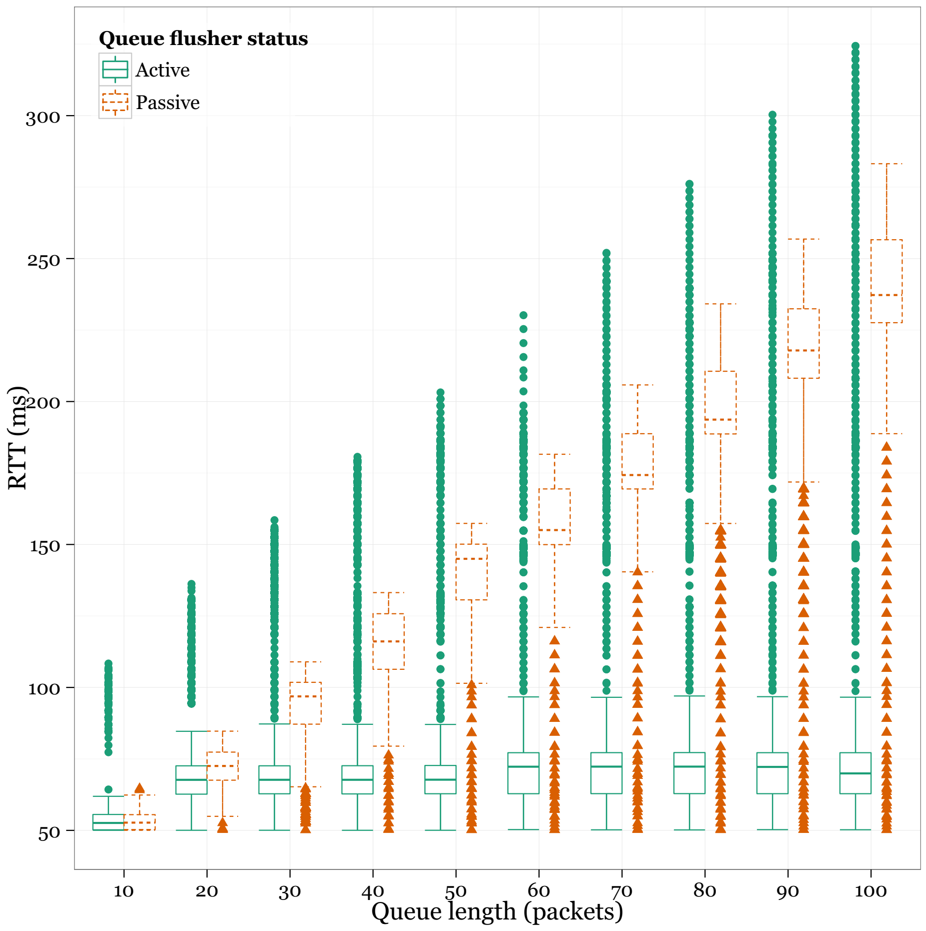


Figure 6.2: Differences in RTT over different queue lengths using a base RTT of 50 ms, a flooder threshold of 1.5 and TCP Reno

with TCP Reno. The congestion window size was measured using the logs from the TCP Flooder, which was introduced in section 5.3.4 on page 56. The flooder queries the active socket for information, which includes the current EWMA RTT measurement and the size of the congestion window. Testing showed that the congestion window size pulled from the socket does not include the initial 'slow start' period. This was discovered when the congestion window logs from the flooder were compared to logs made by `tcpprobe`.

As described earlier, an appropriate congestion window size for the link would be about 40 packets, which is the double of the theoretical BDP displayed in table 6.1 on page 62. The reason for the double BDP size is that one BDP takes one RTT to transfer. Thus, the queue in the bottleneck buffer is a 'good' queue as it is able to disappear within one RTT after the transmission ends. When the congestion window size is more than the double of the theoretical BDP for a link, the queue at the bottleneck would not disappear as quickly and would cause a significant delay for other connections.

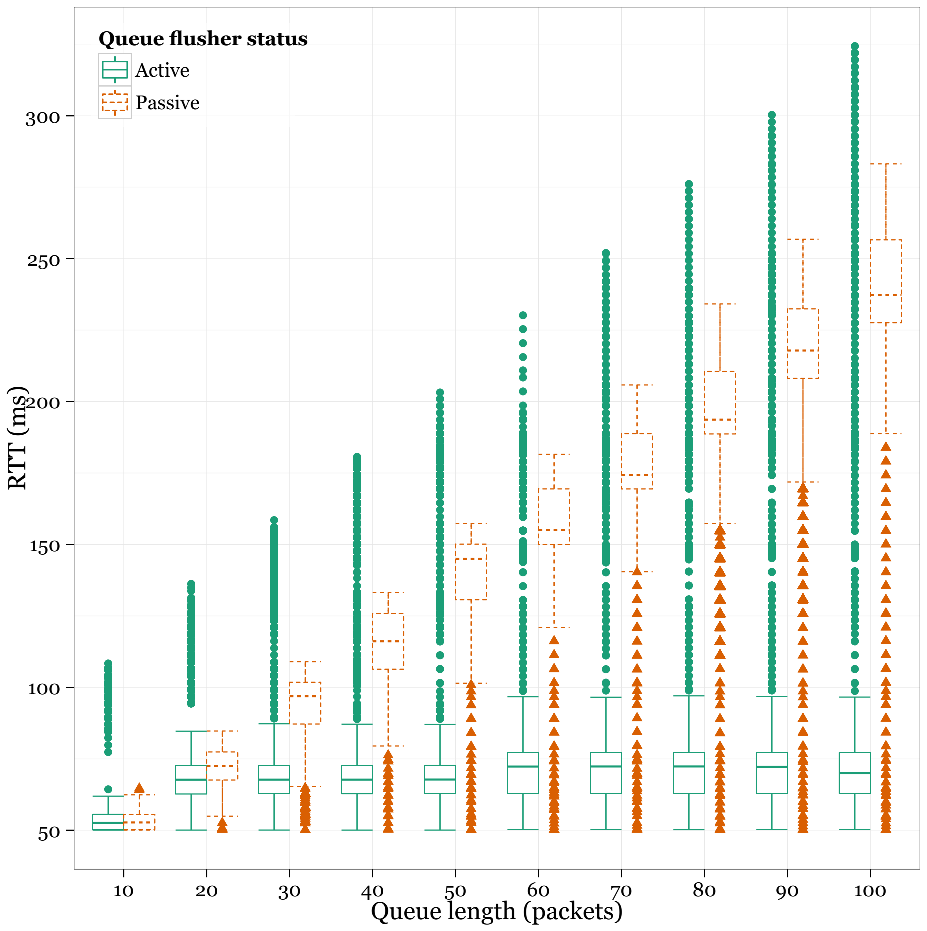


Figure 6.3: Differences in RTT over different queue lengths using a base RTT of 50 ms, a flooder threshold of 1.5 and TCP Cubic

Figures 6.2 to 6.7 on pages 65–70 show that there is no visible difference between the runs with TCP Cubic and TCP Reno. Because of the different nature of the two algorithms, some variations were expected. The graphs do not show a big difference between the two algorithms in any of the runs. Runs done with different base RTT might display different results because a different number of packets are sent before the algorithm responds. Because the maximum buffer size is set for a certain run, the maximum recorded RTT should be the same regardless of the algorithm used. The minimum RTT should also be the same. When there are concurrent connections, the results might be different as the algorithms have to respond to the additional connection. When those conditions arise, TCP Cubic might perform better than TCP Reno when the flusher is passive.

Figures 6.10 to 6.12 on pages 72–74 shows the gain in RTT between runs where the flusher is passive and active. Every figure shows a run with a maximum queue length of 100 packets at different base RTTs. Because the graphs show the gain in RTT between runs, negative values indicates that the RTT was higher when the queue flusher is active than it was when the

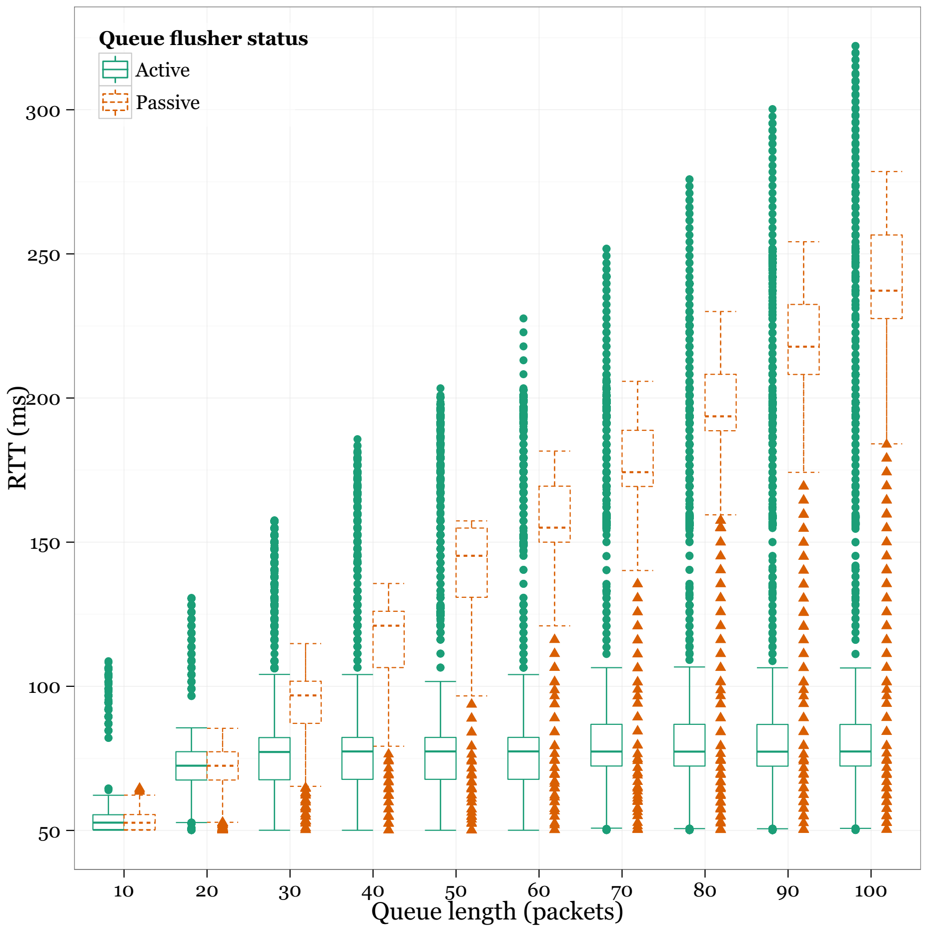


Figure 6.4: Differences in RTT over different queue lengths using a base RTT of 50 ms, a flooder threshold of 1.7 and TCP Reno

flusher was inactive. A higher positive difference indicates that the flusher is doing its job and reducing the mean RTT in the network. A low gain indicates similar RTT measurements in both runs, even though the RTT difference is positive. Different runs may have the same distribution in RTT even though the mean RTT differ.

As the graphs show, the runs using TCP Cubic as congestion algorithm usually is the ones where the improvement is best. When using a flood threshold of 1.5 or 1.7 the improvement is better than if the flood threshold is set to 2. At runs with low base RTT the improvement is about the same whether the threshold is set to 1.5, 1.7 or 2 times the base RTT. At higher RTTs a threshold of 1.5 seems to give the best overall improvement. The improvements shown in figures 6.10 to 6.12 on pages 72–74 does display different RTT gains for TCP Reno and TCP Cubic, where TCP Cubic improves faster. When the base RTT increases, the difference between the TCP algorithms increases. Figure 6.12 on page 74 shows that for a base RTT of 100 ms, TCP Reno has the best overall improvement. The difference between the best and worse improvement is about 25 ms, which probably is

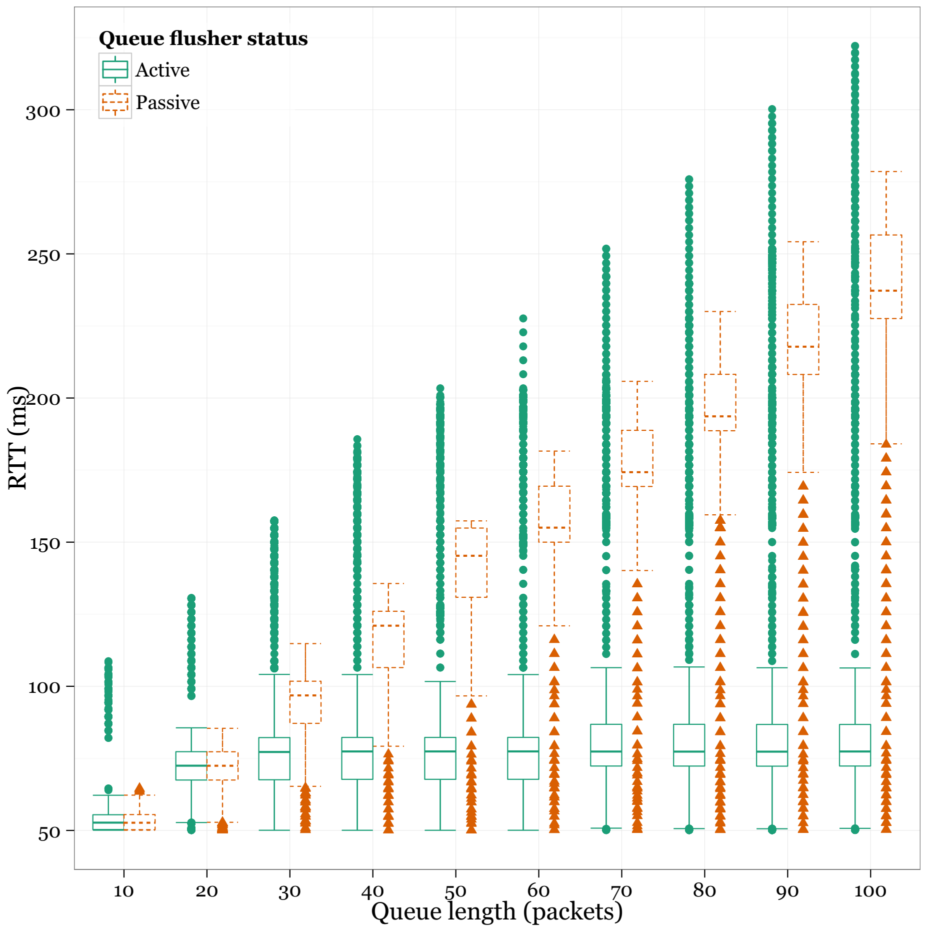


Figure 6.5: Differences in RTT over different queue lengths using a base RTT of 50 ms, a flooder threshold of 1.7 and TCP Cubic

noticeable. Figure 6.12 on page 74 shows that the TCP Reno, when running a flusher with a threshold of 1.5 has the best improvement, with the runs where the threshold is 1.7 and 2, not far behind. It is only in figure 6.12 on page 74 that TCP Reno has the best improvement, with lower base RTT, TCP Cubic does it better. Figure 6.11 on page 73 shows that the algorithms have roughly the same improvement when the flusher is active. The main difference is that TCP Cubic improves faster than TCP Reno, but TCP Reno has a more linear improvement. There could be multiple reasons for the lower improvement when the connection uses TCP Cubic. One reason might be that TCP Cubic is reacting faster than TCP Reno, and the flusher may flush more often. More often flushes with TCP Cubic may be an effect of the TCP Cubic exponentially increasing congestion window, as it does not take as long time to reach the threshold and activate the flusher. When the flusher is activated, it sends a load of UDP packets to flush the queue (as described in chapter 4 on page 31). A higher amount of packets on the link would cause the RTT to spike more often, thus raising the mean RTT and lowering the overall gain.

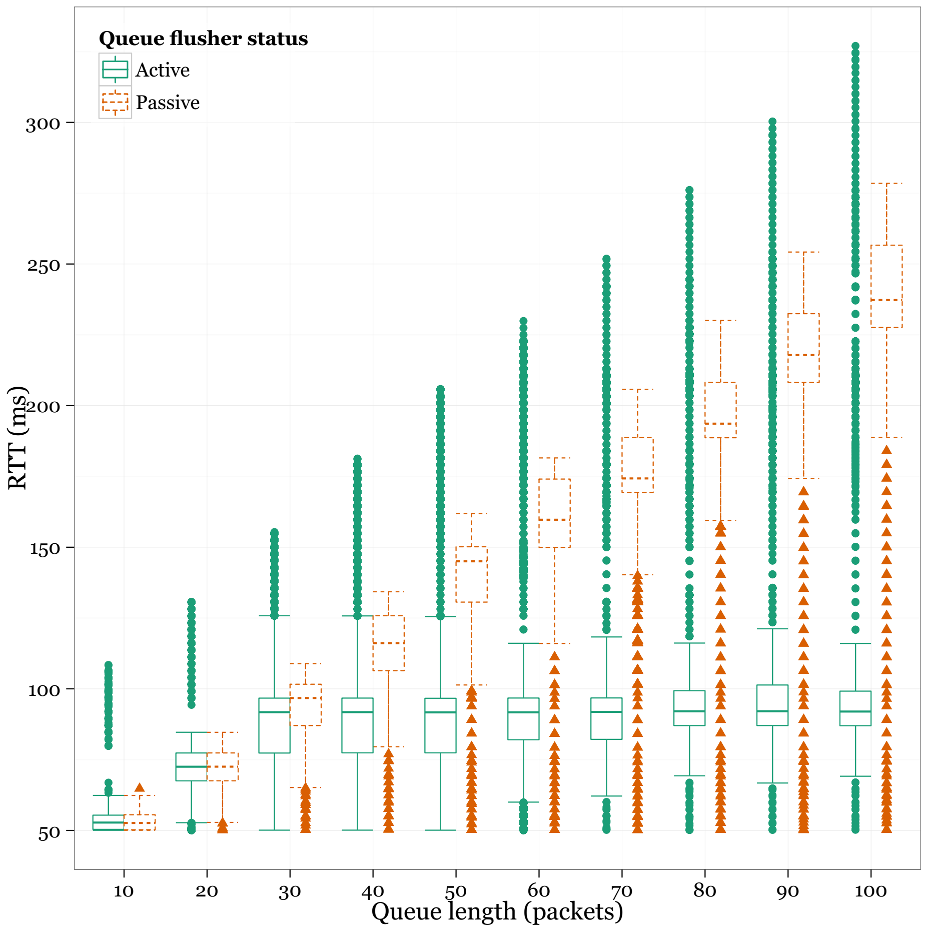


Figure 6.6: Differences in RTT over different queue lengths using a base RTT of 50 ms, a flooder threshold of 2 and TCP Reno

Figures 6.13 to 6.15 on page 76 and on page 77 shows graphs of the mean RTT during runs with a base RTT of 10 ms, 50 ms and 100 ms. The graphs show that the runs with a flood threshold of 1.5 have the lowest mean RTT in addition to the highest gain when the flusher is passive and active. The gain is shown in figures 6.10 to 6.12 on pages 72–74.

For lower base RTTs the difference between the different thresholds is difficult to see, as displayed in figure 6.13 on page 76. What is visible, is that TCP Cubic has a lower median RTT than TCP Reno. This effect might be connected to the slow increase of congestion window in TCP Reno. A slower, steadier increase does not drop packets before the queue is full. With an increase as the one in TCP Cubic, packets are dropped more often due to the exponential increase of the congestion window. This might lead to a slightly lower mean RTT.

When the base RTT is increasing, the improvements becomes more significant. In the run with 100 ms base RTT (figure 6.15 on page 77) it is clear that when the flusher is configured with a threshold of 1.5 or 1.7, the improvements are best. Which is backed by both figures 6.11 and 6.12 on

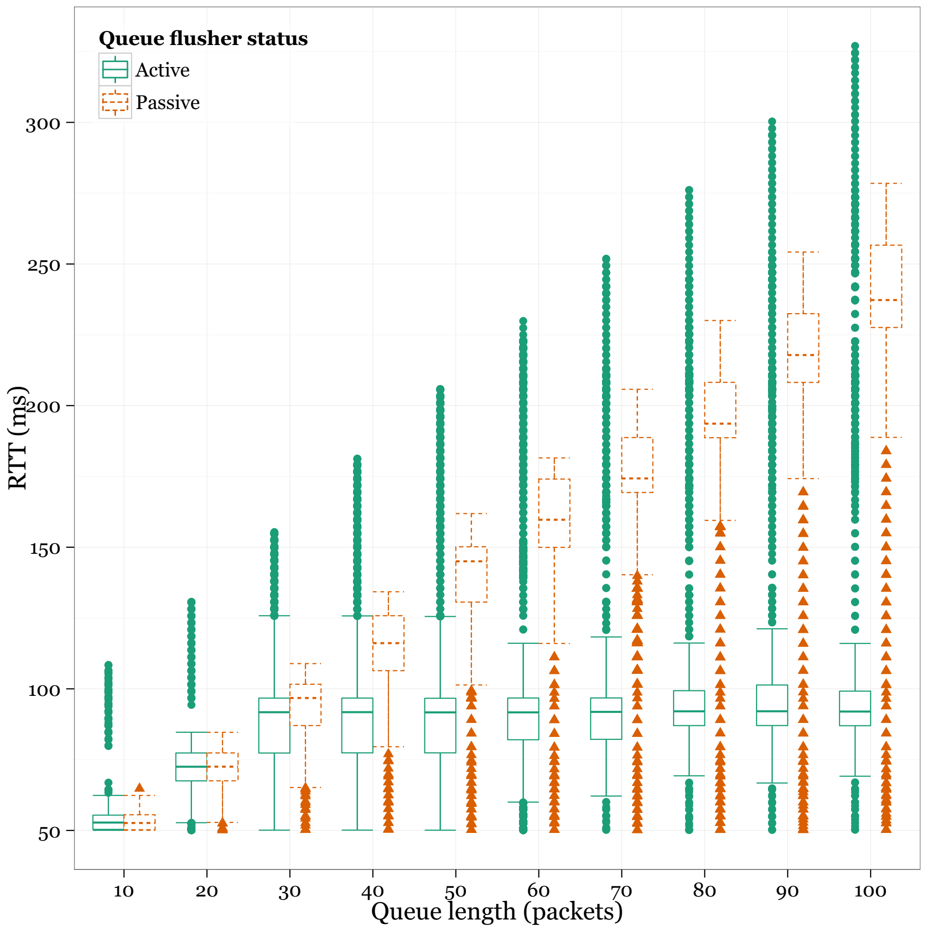
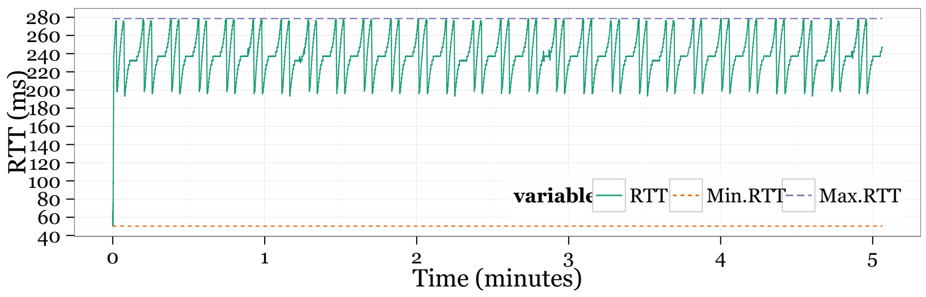
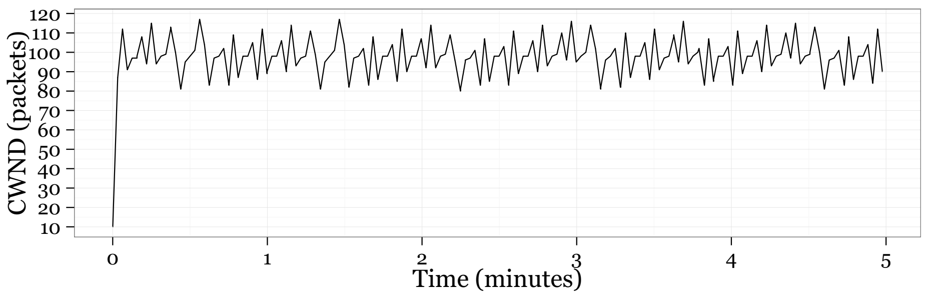


Figure 6.7: Differences in RTT over different queue lengths using a base RTT of 50 ms, a flooder threshold of 2 and TCP Cubic

page 73 and on page 74 and figures 6.14 and 6.15 on page 76 and on page 77. Studying the graphs for the improvement in RTT and the median RTT for the runs using TCP Cubic with a flood threshold of 2, displays a trend where the flood effect is better than it was for TCP Reno. Even though the results are relatively equal, and TCP Reno has a better improvement in the lower end of the scale, TCP Cubic does it marginally better when the whole run is taken into account.

A reason for the slightly lower effectiveness for TCP Cubic connections probably lies in the way the flusher monitors packets and deciding when to flush. Because the flusher measure the minimum RTT for ten successive packets, it requires the measured minimum RTT to be higher than the base RTT at least three times in a row (as described in section 4.5 on page 41). Because this is the same as finding the minimum RTT of the last 30 succeeding packets, the time interval might be larger than it optimally should when operating with TCP Cubic connections. When the connection bandwidth and RTT differ between connections, the time it takes for the thirty packets to be captured differ from connection to connection. If a





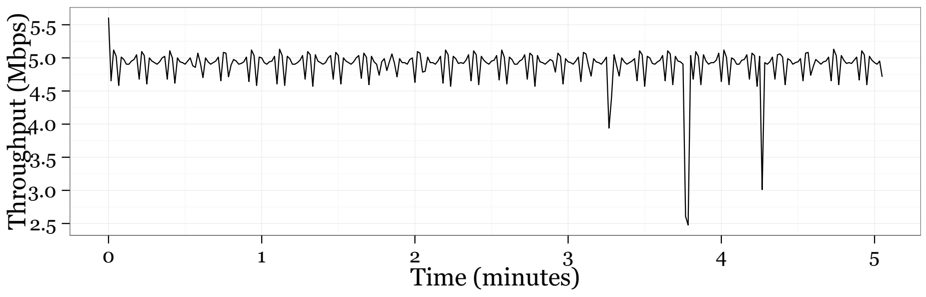


Figure 6.8: Graphs diplaying a passive flusher on a link with a base RTT of 50 ms and a link using TCP Cubic and a bottleneck buffer of 100 packets.

packet drop occurs while capturing packets, and the minimum RTT of the current sample drops below the threshold it takes at least 10 packets until the flusher tries to flush the connection again, because the flusher fills three samples of 10 packets and measures the lowest RTT in each sample. When the minimum RTT in each of 3 successive samples is higher than or equal to the threshold, the flusher tries to flush the queue. If the congestion avoidance algorithm is exponentially scaling the congestion window, such as TCP Cubic does, the congestion avoidance algorithm might discover a packet drop before the flusher react, due to the amount of packets needed. When such an occasion happen, the total RTT improvement may be minimised as the congestion avoidance algorithm is reacting quicker than the flusher, actively disabling the flusher. This should happen on connections with a low BDP and a reasonably sized buffer.

The effect of the measured minimum RTT can also be found when studying the graphs for TCP Reno, but as Reno is increasing the congestion window slower, there is a higher chance for a successful flush as the flusher tries to flush the connection as long as the minimum RTT of the last

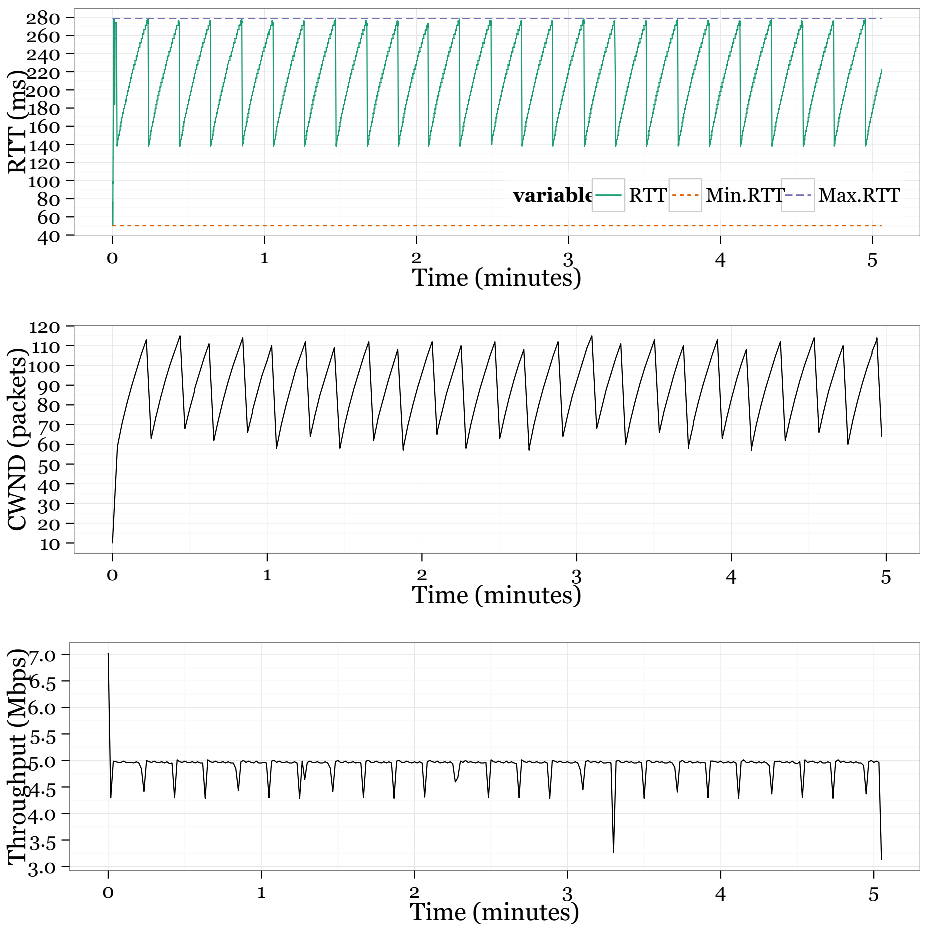


Figure 6.9: Graphs diplaying a passive flusher on a link with a base RTT of 50 ms and a link using TCP Reno and a bottlenck buffer of 100 packets.

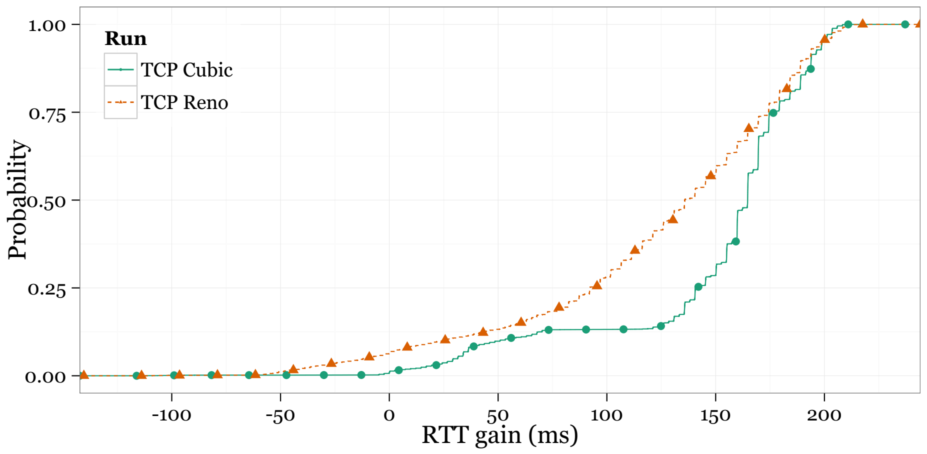


Figure 6.10: Gain in RTT with a base RTT of 10 ms and a bottleneck buffer of 100 packets. Only one graph is shown per congestion algoritm as there is no visible difference betwen the thresholds.

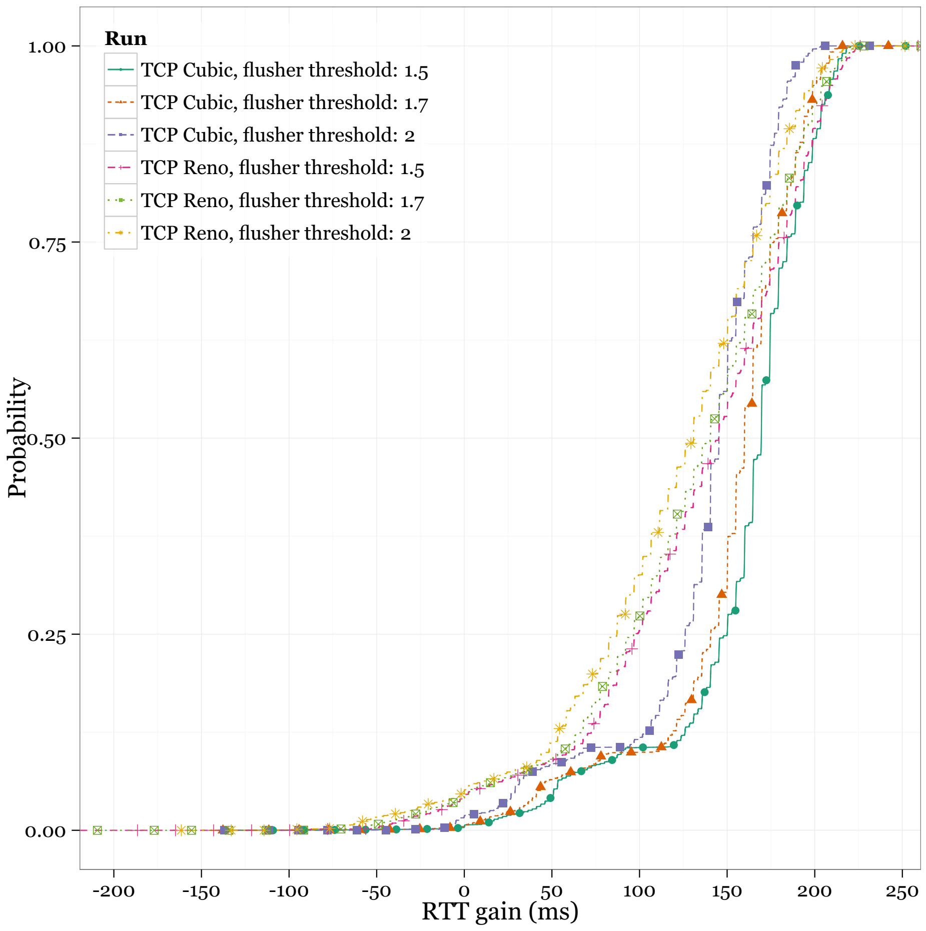


Figure 6.11: Gain in RTT with a base RTT of 50 ms and a bottleneck buffer of 100 packets.

30 packets is above the base RTT. Thus, raising the probability for the flusher to be more active, when dealing with TCP Reno connections where the buffer is slightly larger than $2 \times bdp$. A consequence of measuring the minimum RTT of 30 successive packets is that it allows the queue (and the congestion window) to increase by 30 packets because of the reactive nature of the flusher and the TCP congestion control algorithms. For buffers with a max queue size set to 30 packets or fewer, the flusher should remain inactive. If a connection has a high base RTT, the maximum queue length before the flusher activates is $Q_{length} + 30p$, which for a connection with a base RTT of 100 ms and a bandwidth of 5 Mbps would be a queue length of 12 packets, thus making the minimum queue length 42 packets before the flusher kicks in. In reality the queue might be as large as 71 packets before the flusher kicks in, thus creating a queue which is longer than necessary.

Based on the previous reasoning, a smaller packet sample size should let the flusher respond quicker and be able to be more active when using TCP Cubic connections, thus reducing the worst case scenario. Even though a smaller sample size probably is going to help the responsiveness, the

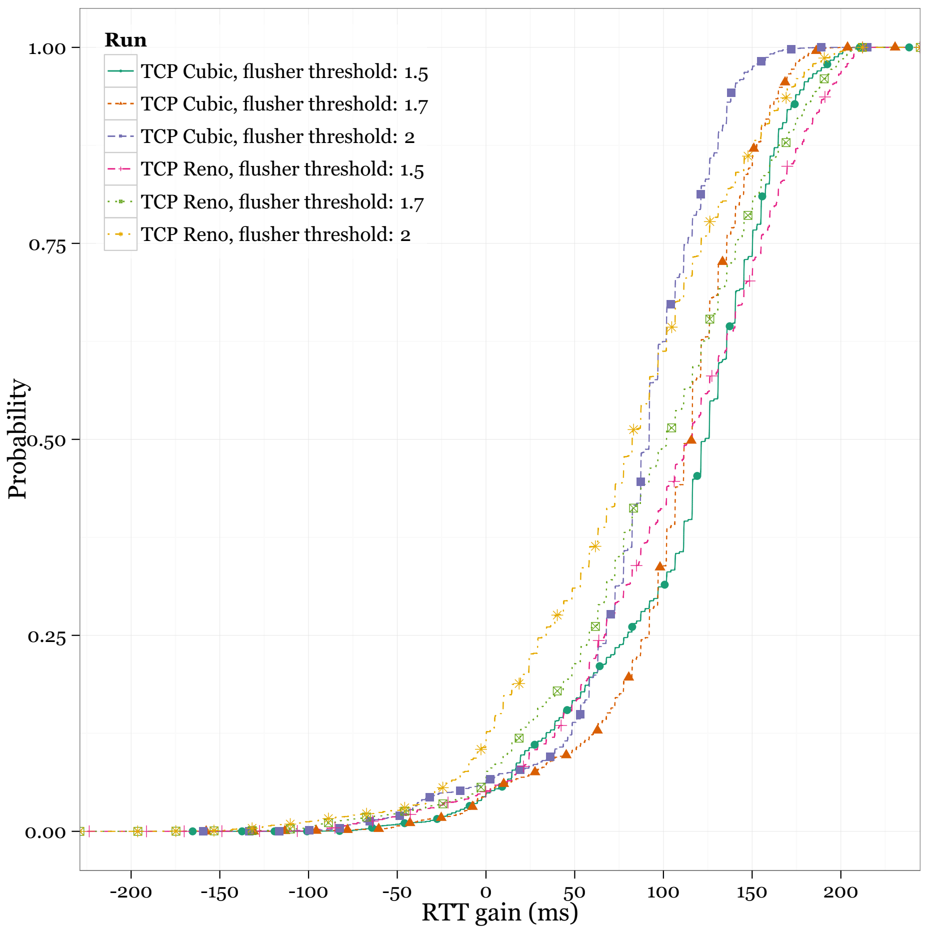


Figure 6.12: Gain in RTT with a base RTT of 100 ms and a bottleneck buffer of 100 packets.

sample size should not be smaller than necessary. A sample size of 1 would flush the moment the measured RTT is higher than the base RTT. In the same way such responsiveness is good, it is also a bad thing when taking the flushing into account. As the flusher sends UDP packets in an attempt to flush the queue, the large amount of sent packets could have a negative effect on throughput and possibly lead to starvation, which is unwanted. Starvation is achieved when an unresponsive connection uses the whole capacity of a link. Because it is unresponsive, other connections do not get their packets to their receiver and the connection break.

Reducing the sample size to 10 packets should make the flusher more responsive and avoid starvation of the connection. Following the calculation described earlier, the flusher would activate when $delay(Q_{length} + 10p) > basertt$. The minimum queue length where the flusher would activate is 32 packets, which is more likely to be nearer the congestion window size on that specific connection, while the higher limit where the flusher could activate is at 52 packets as the limits would be $[32, 52]$ as opposed to $[12, 72]$.

The ranges show the bottleneck queue length before the flusher activates. In the best case scenario the 30 packet sample size is better, but as the maximum value for the 10 packet sample size is lower, the worst case scenario is definitively better. TCP Reno may fall under the best case scenario, but the TCP Cubic algorithm would fall under the worst case scenario due to its exponential nature. Thus, a lower sample size would lead to a more active flusher when using TCP Cubic. Which in turn allows it to be better when dealing with TCP Reno as the flusher is more reactive.

Another optimisation is to allow a queue of a certain length regardless of the bandwidth. When a sample size of a certain number of packets is used, the time is changing with the bandwidth. Thus, a slow link takes longer to get to the wanted sample size than a link with higher bandwidth. Using a time sample could also be a viable method to determine when to activate the flusher. As humans measure time in time and not packets, a time limited function might be appropriate. The possible downside is that it will check a variable amount of packets in response to the link bandwidth. The sample size could range from a few packets to several hundred when a sample size is for instance 100 ms.

There should be little to no serious impact when using a time based sample size as opposed to a packet based sample size as there are positive and negative sides with both methods. For a time based sample size, the bottleneck queue length might be high for a certain connection. Which could cause, when another connection share the bottleneck, the slower connection's RTT to rise because the length of the bottleneck queue length. This is the case if the queue is a FIFO queue and does not have implemented an AQM like PIE or CoDel (for a brief on CoDel and PIE see sections 3.10 and 3.11 on page 23 and on page 25). As for the packet limited sample size, if configured incorrectly it will cripple the effectiveness of the flusher as the flusher will likely not activate as often and may not discover the buffer size. While the time limited sample size will check for need of flushing once in a specified time interval, the packet limited sample size will check for the need of flushing each time a set amount of packets have been captured. The packet limited sample size may be better at keeping a low queue length at the bottleneck, but at the risk of reducing throughput if the sample size is too small. Reduced throughput is unwanted due to the effectiveness of other transfers. As the flusher sends unnecessary packets, which only purpose is to be dropped at the receiver, the packets adds unnecessary traffic if too many are sent in rapid succession. Due to this, the flusher should not be overly aggressive, but neither too slow as that may also affect the overall responsiveness in the network.

A time limited sample size may decrease the throughput of connections with a high base RTT. If the sample size is not long enough, the flusher could be trying to flush the queue several successive times. This increase in packets could negatively impact the network throughput and connections with high RTTs would react slowly to the flushes.

Another improvement may be to only let the flusher become active when the minimum measured RTT in a sample size is a multiple of the network base RTT and higher than a set point, i.e. 50 ms. This would allow the flusher to remain inactive if there is an AQM present at the bottleneck or

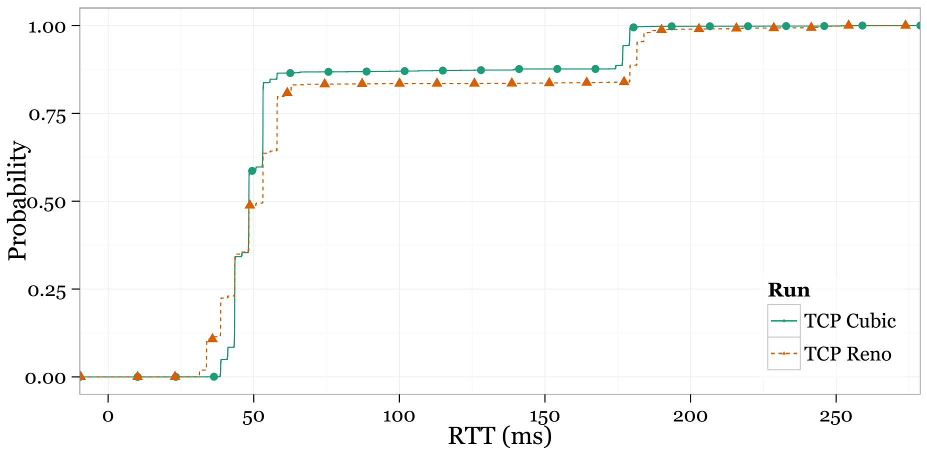


Figure 6.13: RTT with a base RTT of 10 ms and a bottleneck buffer of 100 packets. Only one graph is shown per congestion algoritm as there is no visible difference betwen the thresholds.

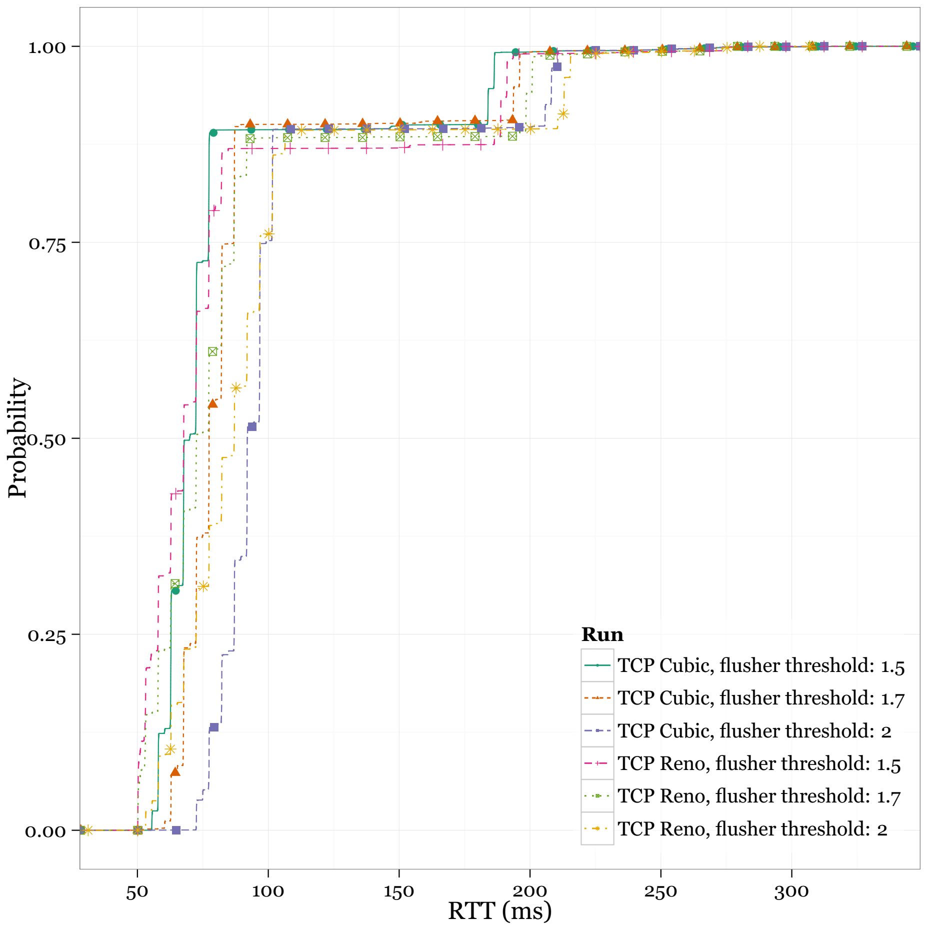


Figure 6.14: RTT with a base RTT of 50 ms and a bottleneck buffer of 100 packets.

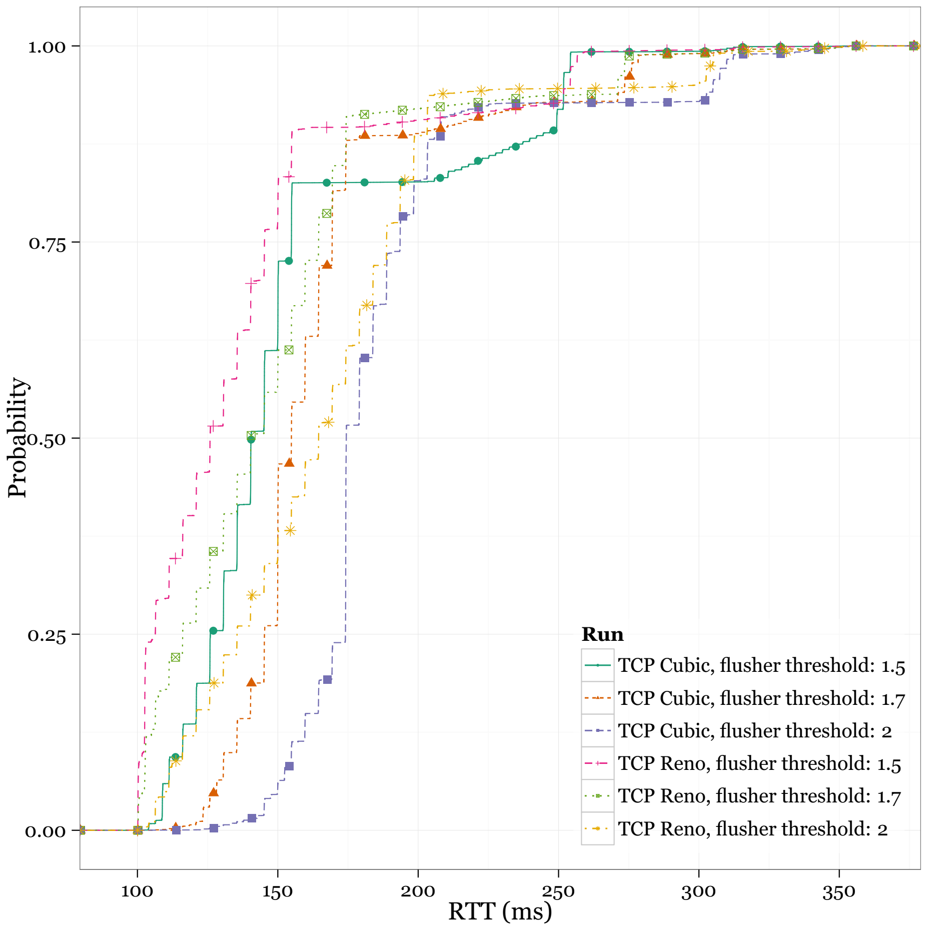


Figure 6.15: RTT with a base RTT of 100 ms and a bottleneck buffer of 100 packets.

if the network base RTT is very low and/or the bottleneck buffer is of a reasonable size, which cannot support queues that have a large impact on RTT.

In addition to the flusher activation at the flood threshold and the minimum measured RTT in the sample size is at least 50 milliseconds, the flusher also activates when there is at least 30 seconds since the last flush. This way it should be able to improve the RTT when there is a buffer in the bottleneck which is capable of holding a very large queue which induces a large delay. As the forced flush after 30 seconds is implemented in such a way that it disregards the measured minimum RTT for the last sample size, the flusher tries to flush at 30 second intervals if none of the other conditions occurs. Even though the flusher only forces a flush when it is 30 seconds since the last flush, the effect of this should only be visible on runs where the bottleneck buffer is reasonably sized and the RTT does not reach the set threshold. On runs where the buffer is able to sustain a queue that induces a large delay, the forced flush will not happen unless the buffer never fills and the base RTT is high. This might happen when the flusher is started during an ongoing connection and the bottleneck buffer is huge. After the first forced flush, the flusher should resume normal operation where it measures the RTT against a set threshold.

6.3.1 Throughput

The differences in RTT are mainly because the flushing as the flusher sends UDP packets which momentarily generate more traffic. The increased traffic fills the buffer and the TCP algorithm shrinks its congestion window. This is equal to decreasing the load on the link by sending fewer packets. The expected result is that the throughput decreases during flushes, but it should improve one RTT after the queue was flushed.

Figure 6.16 on the next page is a box plot showing the differences in mean throughput for queue lengths from 10 to 100 packets. The run is made using TCP Reno with a flusher threshold of 1.7 and a base RTT of 50 ms. The flusher threshold marks the threshold for when the flusher starts to flush. A threshold of 1.7 is equal to measuring RTTs which is greater than or equal to $1.7 \times$ base RTT. When the threshold is exceeded, the flusher tries to flush the queue.

As figure 6.16 on the facing page shows, the throughput does not suffer much when the flusher is active. The mean throughput is lower when it is active, but the maximum throughput is about the same.

The graphs using runs with a base RTT of 50 ms, shows the throughput declining when the flusher kicks in. As shown in the graph, the throughput when the flusher is active and passive is equal when the buffer size is 10 packets. The reason for this is that the flusher does not kick in because of the queue not managing to create a large enough delay. When the buffer is able to sustain a queue of 60 packets, the throughput has reached its lowest point. The graphs showed that every increment in queue size above 60 packets did not decrease the throughput further.

When the flusher is not running there is no other traffic on the line and maximum throughput is achieved. A standing queue in the buffer leads to the suffering of network responsiveness when throughput is at its highest.

Figure 6.17 on the next page shows the same box plot as figure 6.16 on the facing page. The only difference is that TCP Cubic is used for the runs, not TCP Reno. As shown, the throughput is almost identical to the throughput in figure 6.16 on the next page which shows that both TCP Cubic and TCP Reno are able to sustain the same throughput on long transfers.

As figures 6.18 to 6.20 on pages 80–82 show, there are some differences in throughput between the runs with different flooder thresholds. In every figure, an active flusher with a threshold of 1.5 had the biggest impact on throughput. Even though the threshold had a positive impact on RTT, the active flusher resulted in an increase in traffic. Thus, lowering the throughput because of the increased link load.

Figure 6.18 on page 80 shows two graphs, one for TCP Cubic and one for TCP Reno. The reason is that the difference in throughput was difficult to see. This may be because of the flusher was configured to flush connections where the measured RTT was over 50 ms. Giving a standing queue at the

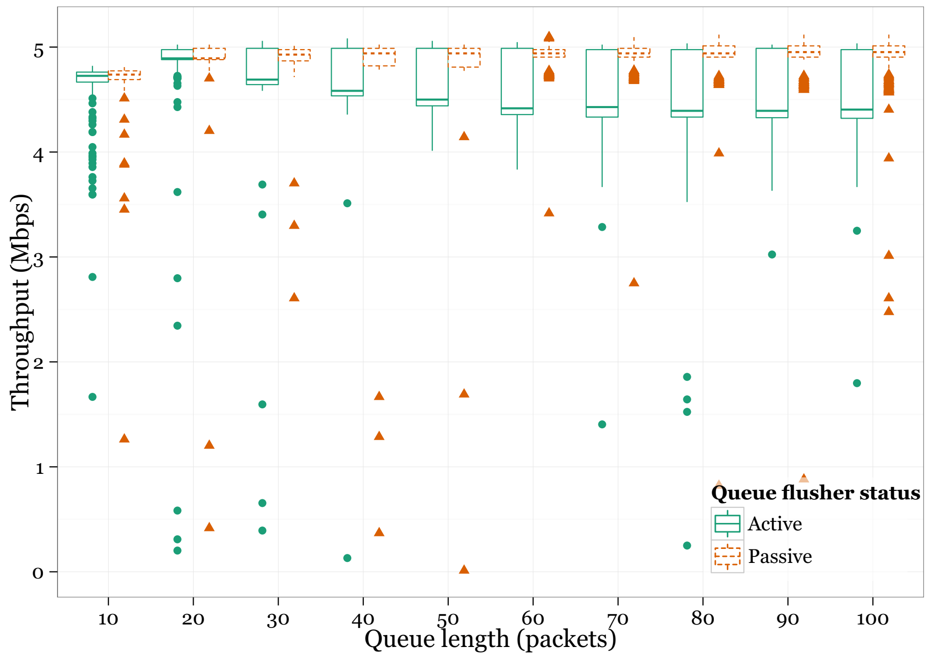


Figure 6.16: Difference in throughput when the flusher is running on a link with base RTT of 50 ms, TCP Reno and has a threshold of 1.7.

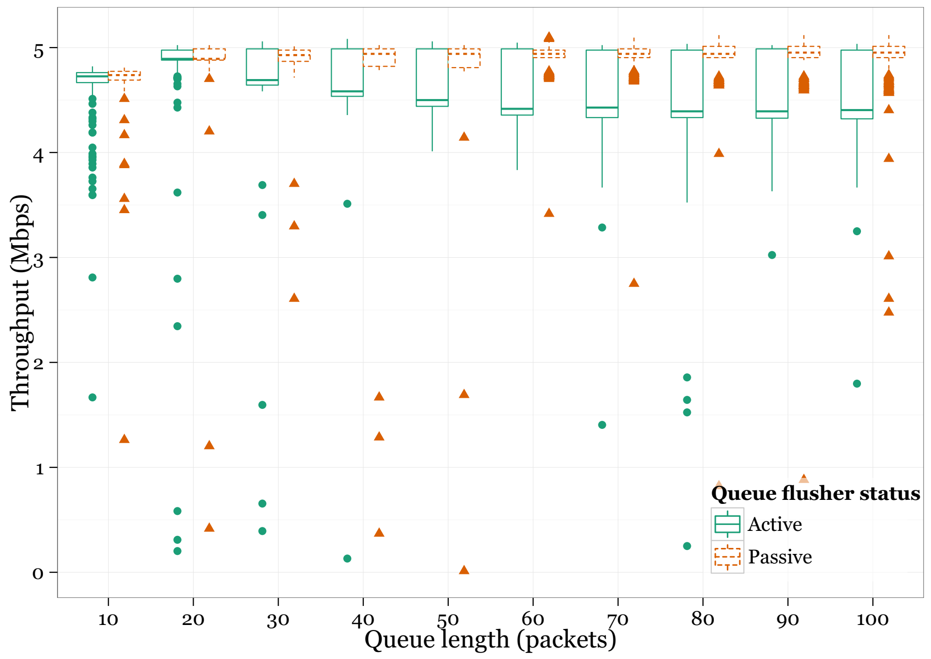


Figure 6.17: Difference in throughput when the flusher is running on a link with base RTT of 50 ms, TCP Cubic and has a threshold of 1.7.

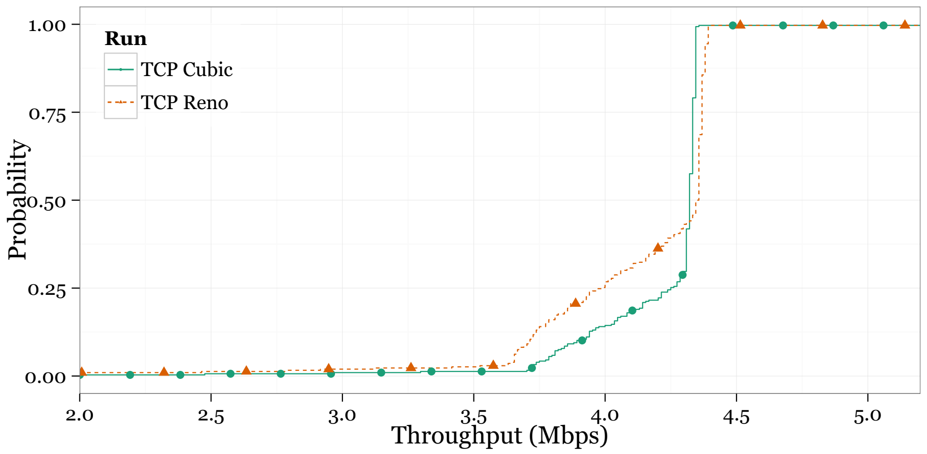


Figure 6.18: Throughput when using an active flusher and a base RTT of 10 ms with a buffer size of 100 packets. As the differences between the thresholds were miniscule, only one graph per TCP algorithm is showed.

bottleneck. As the flusher responds a short time after an RTT of 50 ms is reached, the differences are minuscule.

When the base RTT is increased, and the BDP follows, the flusher is responding slower. When the flusher responds slower, the thresholds get more precedence as they dictate when the flusher should respond to a queue. As shown in figures 6.19 and 6.20 on the next page and on page 82, a flusher running with a threshold of 1.5 has a bigger impact on the throughput than the higher thresholds. The throughput is best when the flusher runs with a threshold of 2. As shown in figures 6.14 and 6.19 on page 76 and on the facing page, TCP Cubic has slightly lower RTT than TCP Reno and in addition to the throughput being slightly lower. This shows that the exponentially expanding congestion window of TCP Cubic has its downsides. As TCP Reno has slightly higher overall RTT it does transfer packets better because of less variation in RTT, giving a higher throughput.

Figures 6.19 and 6.20 on the next page and on page 82 show that there are some differences between the thresholds. As observed, and discussed earlier, a lower threshold results in a lower RTT and a lower throughput.

Figure 6.15 on page 77 shows a small increase in RTT when the flusher threshold is raised from 1.5 to 1.7. The difference in RTT between the thresholds 1.5 and 1.7 is lower than the RTT increase between the runs with thresholds 1.7 and 2. This is true for both TCP Cubic and TCP Reno. A similar correlation is shown in figure 6.20 on page 82, where the throughput difference between the runs with thresholds of 1.7 and 2 is lower than it is for the runs with thresholds of 1.5 and 1.7.

The current suggestion for an appropriate flusher threshold is $1.7 \times$ base RTT.

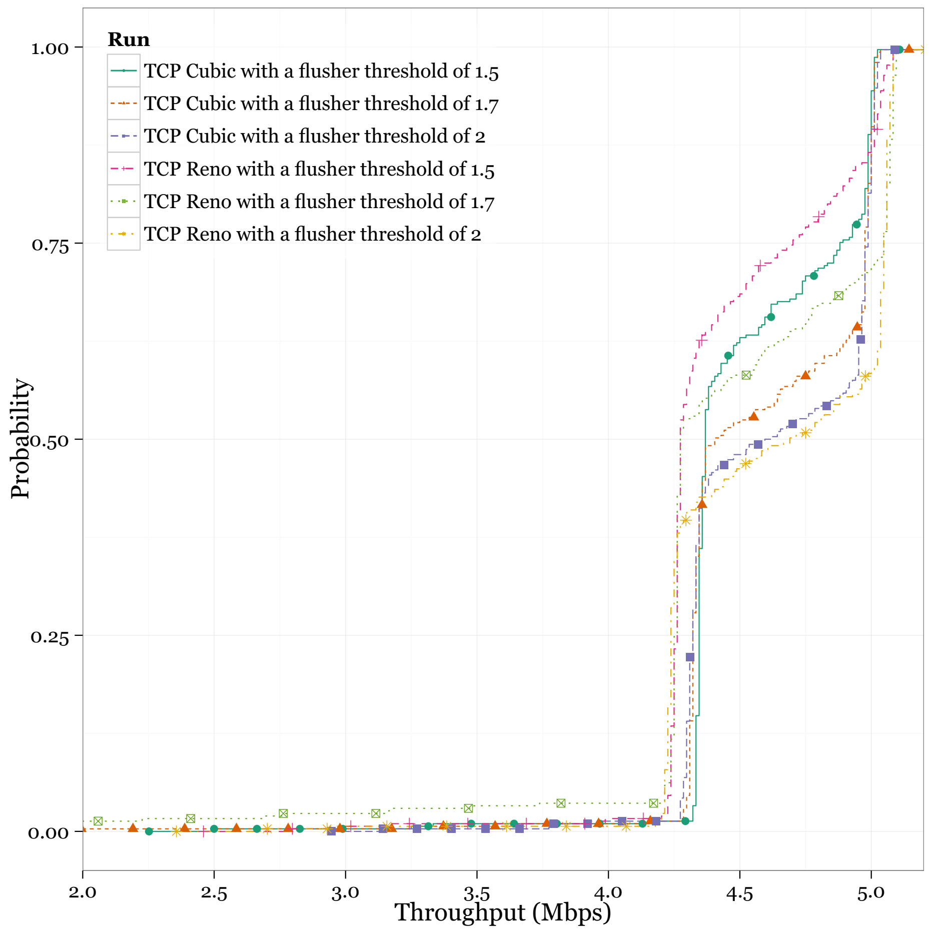


Figure 6.19: Throughput when using an active flusher and a base RTT of 50 ms with a buffer size of 100 packets.

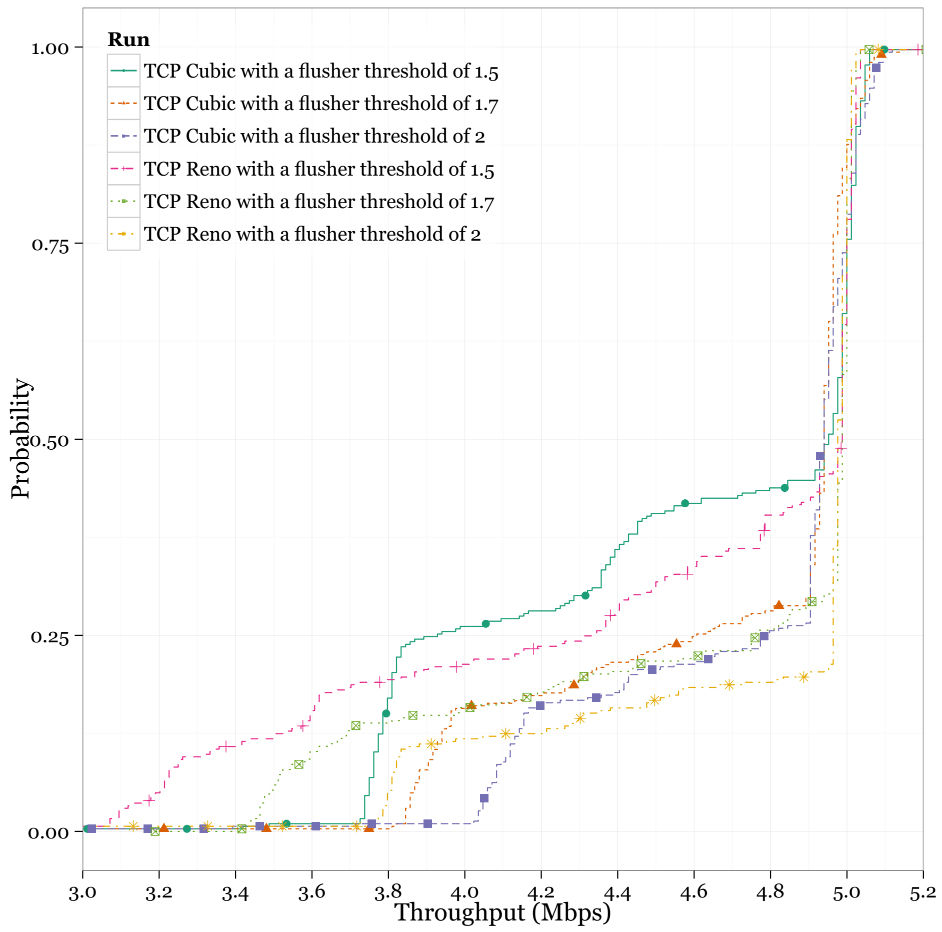


Figure 6.20: Throughput when using an active flusher and a base RTT of 100 ms with a buffer size of 100 packets.

6.3.2 Active queue managers

In order to see how the queue flusher's efficiency compares to existing technologies, some AQMs needed to be tested. The variants selected were CoDel and PIE. These are self adjusting AQMs, meaning they generally work just fine when configured with their default values. In special cases they may need to be configured differently, but the measurements here are done with the default parameter values because the 'queue flusher' also uses an adaptive algorithm.

The default parameters of CoDel and PIE are:

- CoDel [11]

  Queue limit 1000 packets

  Target queue delay 5 ms

  Update interval 100 ms

- PIE [25]

  Queue limit 1000 packets

  Expected queue delay 20 ms

  Drop probability update frequency 30 ms

The runs to measure the performance of both CoDel and PIE were made with a passive flusher to measure the RTT in addition to TCP Dump running at the sender and receiver. An active flusher should not have an impact due to the RTT being kept under control by the AQMs. If the flusher had tried to flush the queue, most of the packets would be dropped as the active AQM would step into action. As the queue is handled by the AQM, it quickly drains even though the flusher were flooding the network, which ensure a responsive network connection.

The reason for not using FQ Codel is that it is essentially the same as CoDel, it is only the scheduling process that differs. CoDel implements only one queue, while FQ Codel uses a Fair Queueing scheduler that stores packets in different queues. The decision of which queue a given packet is sent to, is done by a hashing algorithm that takes the IP addresses and ports of the connection into account. As the scheduler is the only difference between FQ Codel and CoDel and the tests are run with one lengthy transfer, the FQ Codel results would be indistinguishable from the results when using CoDel.

As shown in figures 6.21 to 6.26 on pages 84–89 the flusher does a reasonable job when compared to the existing AQMs. It is expected that the flusher is not performing as well as CoDel or PIE as it temporarily increases the link load, causing spikes in the RTT. The existing AQMs know exactly what is going on inside the device they run on while the flusher has to guess in order to get a result. The RTT does not spike when a packet is dropped, this makes the RTT lower when using AQMs.

In the best case scenarios the flusher is almost on par with the PIE AQM while it differ more in other scenarios. Figures 6.21, 6.23 and 6.25 on the

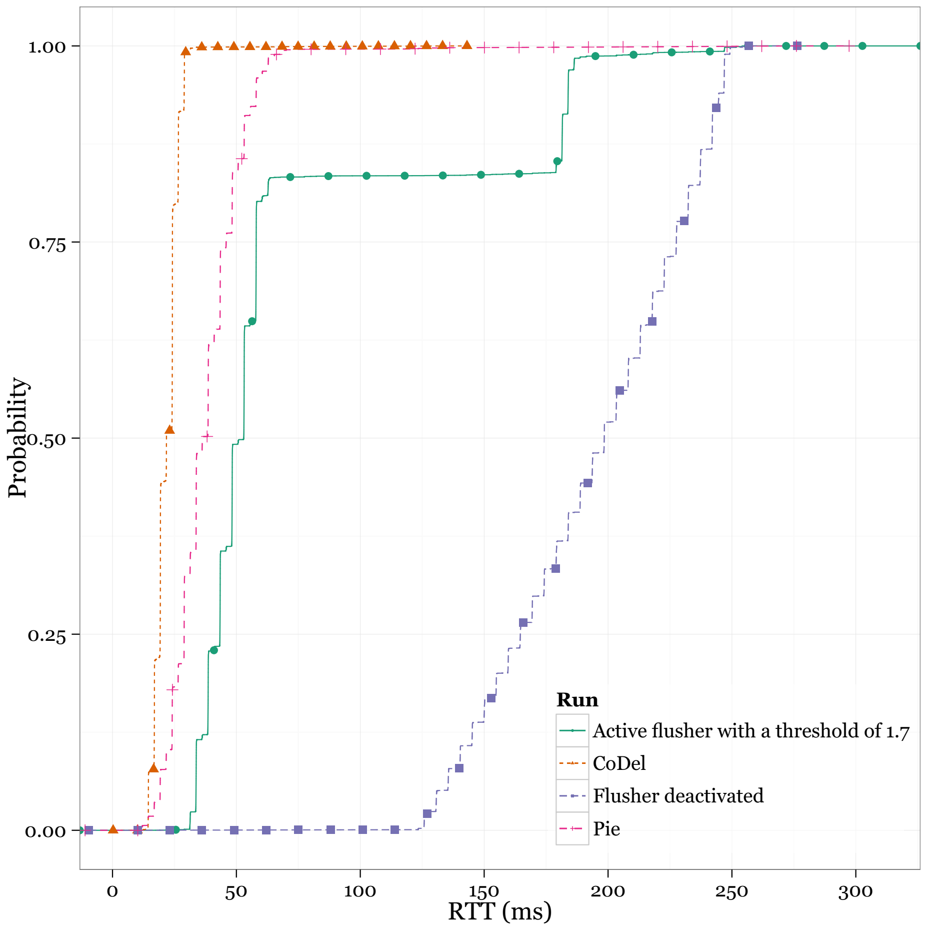


Figure 6.21: RTT with a base RTT of 10 ms using TCP Reno

current page, on page 86 and on page 88 show runs using TCP Reno. When the runs with TCP Reno are compared with the runs that used TCP Cubic, they show a better improvement. The improvement may be caused by the slower increase of the congestion window. The queue is 100 packets, and a flush may be able to stop the TCP Reno transmission earlier. A packet drop may be forced because the UDP packets fill the buffer before the TCP packets reach the bottleneck. With TCP Cubic, the congestion window increases exponentially, which may cause the buffer to be fuller when a flush happens. This suggests that the flush may not have the same effect with TCP Cubic as with TCP Reno.

As shown in figures 6.21 and 6.22 on this page and on the next page, TCP Cubic has a slightly better performance than TCP Reno on links with a low base RTT. The same effect should also be seen on smaller buffer sizes. When using either TCP Reno or TCP Cubic the RTT is almost on par with PIE when the link has a low base RTT. The flusher is almost as effective as PIE because of the probability algorithm PIE uses to determine a maximum queue length at the router. As the algorithm uses probability and have target queue delay it does it best to stay below (as described in section 3.11

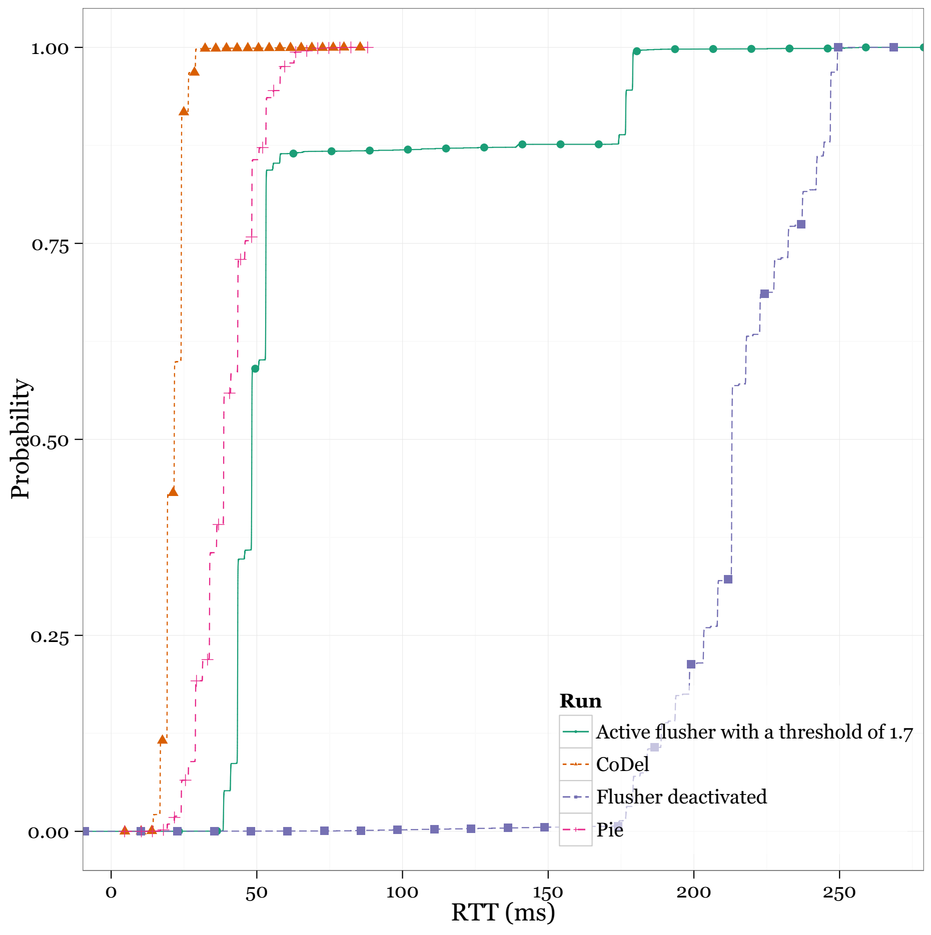


Figure 6.22: RTT with a base RTT of 10 ms using TCP Cubic

on page 25), PIE may not have dropped as many packets when the RTT is as low as 10 ms as the queue will be gone quickly when the maximum bottleneck queue length is 100 packets.

As figures 6.21 to 6.26 on pages 84–89 show, the flusher does a good job compared to when it is not active. The RTT differences displayed does indicate that the flusher does its job properly. At its best, the flusher is on par with the PIE AQM.

When the flusher is inactive and the queue at the bottleneck is a FIFO queue, the queue size is allowed to steadily increase before it overflows and packet drops occur.

With TCP Cubic, which have a more rapid expansion of its congestion window, the flusher has an effect. The effect is visible, but the flusher may not be as effective as it is with TCP Reno streams because the TCP Cubic algorithm detects congestion and reacts before the flusher, which causes the flusher to be less active when TCP Cubic is used. Because the flusher is configured with a sample size of 30 packets, it needs the lowest measured RTT in a sample to be greater or equal to the $threshold \times$ base RTT. A smaller sample size may cause the flusher to be more active when a

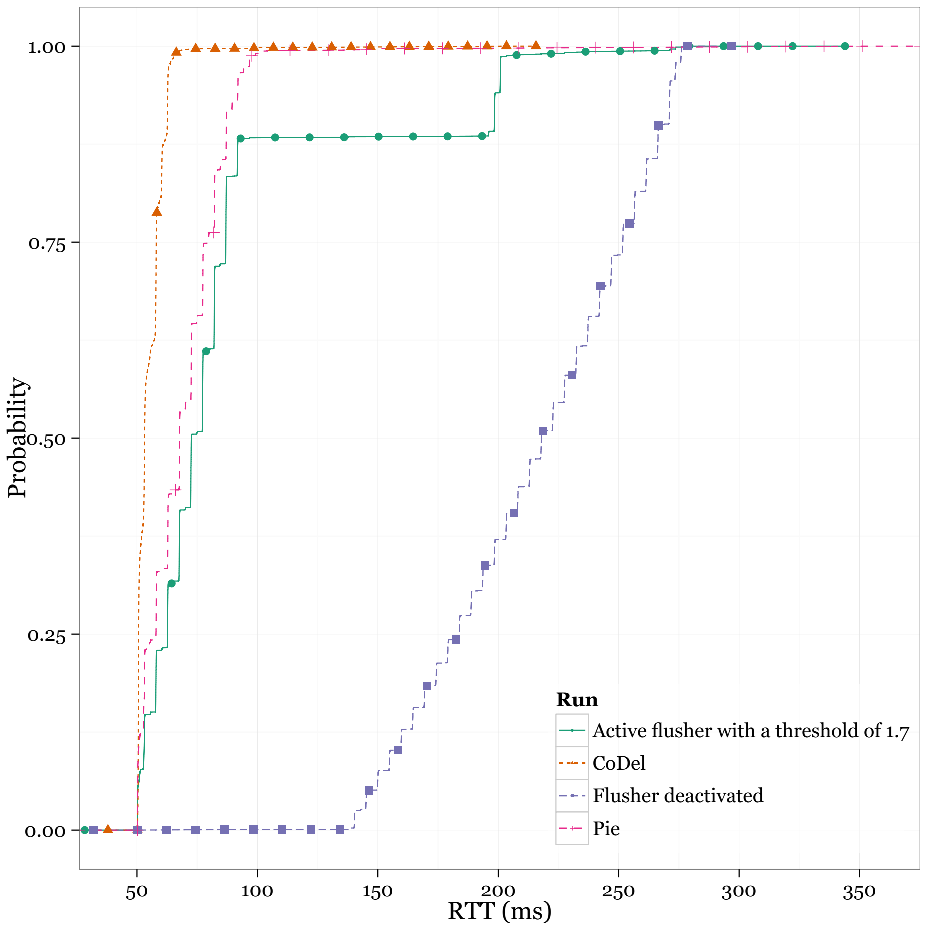


Figure 6.23: RTT with a base RTT of 50 ms using TCP Reno

connection is using TCP Cubic as the amount of sampled packets is smaller.

As shown in figures 6.21 to 6.26 on pages 84–89 when the probability reaches about 88% the RTT suddenly makes a jump. This jump is caused by the RTT peaks generated during flushes and is expected.

The flushing increases the amount of large RTT peaks, compared to the run where the flusher is not running. Frequent peaks have an impact on jitter. Frequent peaks show that the flusher is active and tries to flush the connection, and the queues are reduced. Reduced queues are the wanted effect for a queue flusher.

Compared to the AQMs CoDel and PIE the flusher is indeed doing a decent job when configured right. As described earlier, the flusher may not be as active with TCP Cubic connections due to the rapid expansion of the congestion window. Even though the flusher is less active on a TCP Cubic connection than a connection using TCP Reno, it does not mean that the flusher is not working with TCP Cubic.

When the base RTT increases, the difference between the flusher and the AQMs increases. There may be several reasons for the increase, but the main reason would be that the TCP algorithms are reacting slower due to

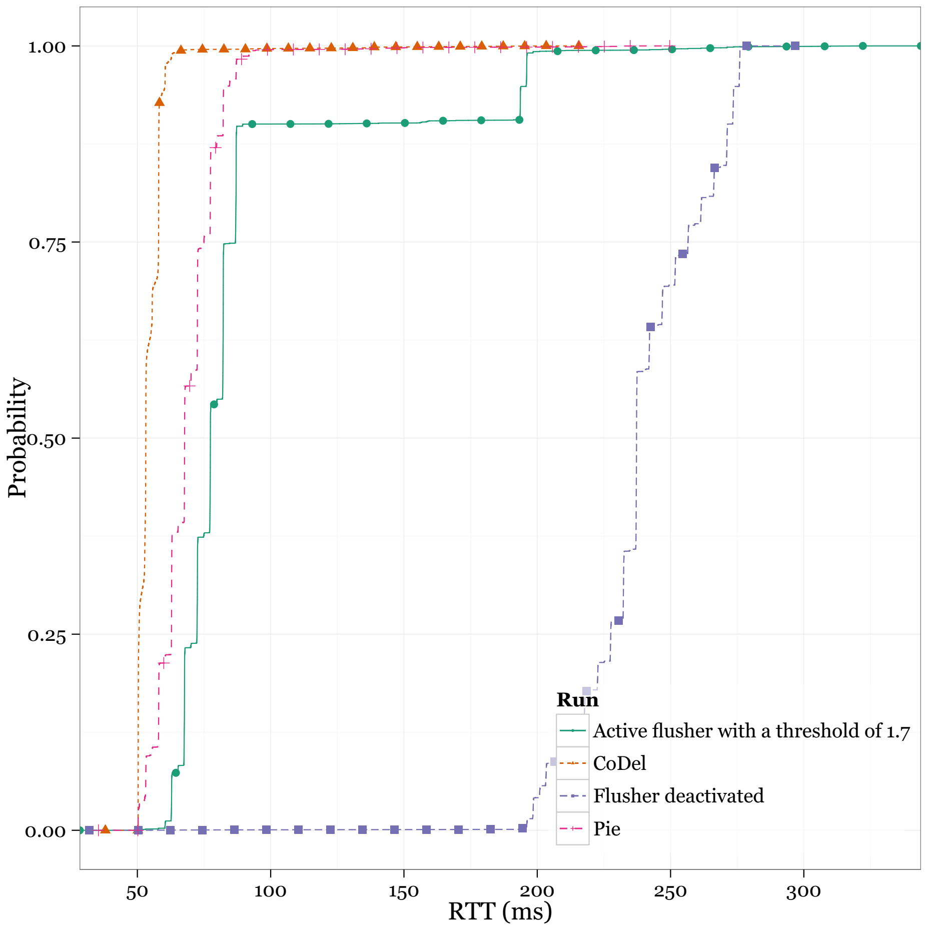


Figure 6.24: RTT with a base RTT of 50 ms using TCP Cubic

the higher base RTT. A higher base RTT impacts both the TCP reaction time and the reaction time of the flusher. Because the flusher measures the RTT by monitoring ACKs and it is placed at another spot than the bottleneck, the flusher reaction increases with the base RTT. The AQM algorithms are run at the bottleneck, and know every node parameter. Such as the buffer size and arrival/departure rates. They may react quicker because they know when the queue is filling the buffer, or when the queue is bigger than it ideally should be because they are detached from the base RTT of the connections passing through the bottleneck.

6.3.3 Flusher throughput compared to the threshold of active queue managers

As presented in section 6.3.2 on page 83 the mean RTT measured when the flusher is active can be almost as good as when AQMs are running at the bottleneck.

It is obvious that the flusher decreases throughput. It is interesting to see how the throughput compares to the throughput measured when

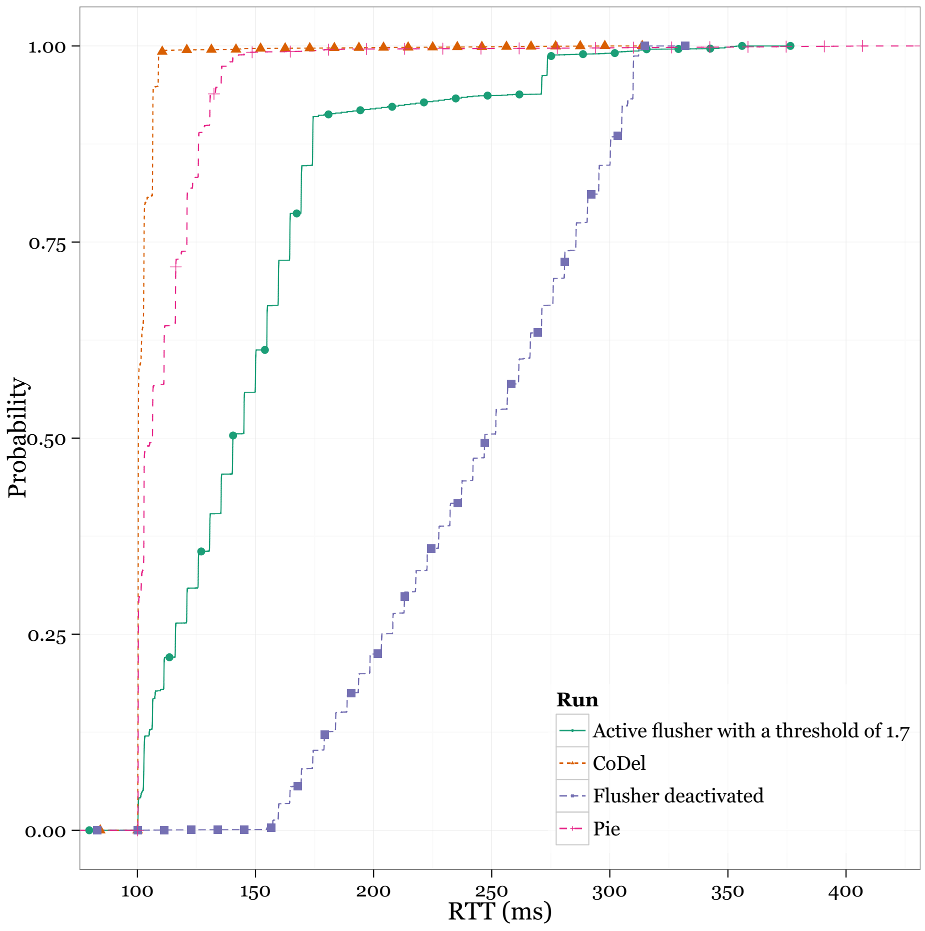


Figure 6.25: RTT with a base RTT of 100 ms using TCP Reno

using AQMs. Because the throughput did not differ between the runs with TCP Cubic and TCP Reno, figures 6.27 to 6.29 on pages 90–92 shows the throughput from the runs using TCP Reno.

When studying the graphs it becomes apparent that the throughput when CoDel is active, is actually decreasing when the base RTT increases. A reason for this may be that the CoDel AQM increases its drop probability when the queue causes a delay of 5 ms [3, 11] (see section 3.10 on page 23). As the setpoint for the target delay is fixed it ensures a higher throughput on links with low base RTT. A setpoint of 5 ms may be good for connections with low base RTT, but connections with higher base RTT should have a higher setpoint before packets are dropped.

The PIE AQM (see section 3.11 on page 25) is scaling better because its drop probability is calculated by measuring the transmission rate [4, 25]. Because the drop probability is calculated based on the transmission rate, PIE allows the queue to be as long as needed for optimal throughput, as opposed to CoDel.

Figure 6.28 on page 91 shows some difference between the throughput when the flusher is active and the AQMs are enabled. In the best case the

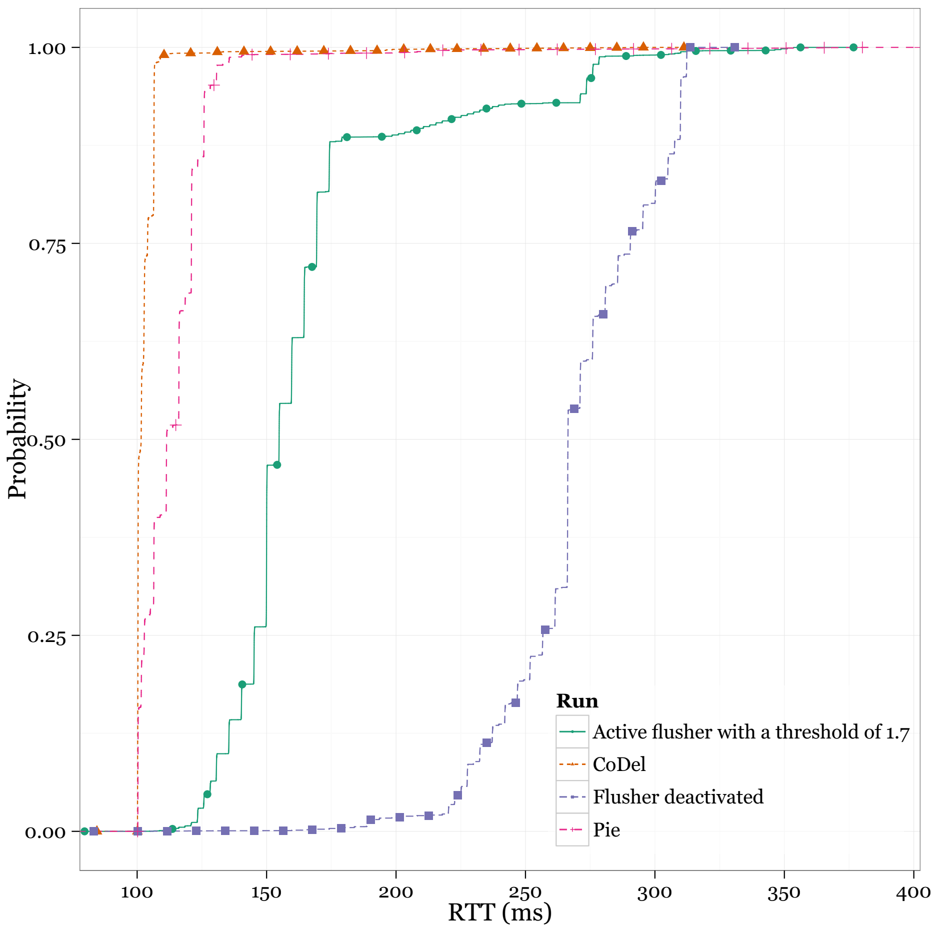


Figure 6.26: RTT with a base RTT of 100 ms using TCP Cubic

throughput when the flusher is active is almost as good as PIE. As PIE is optimised for high throughput, its throughput is the benchmark. When having a base RTT of 50 ms, the 50 ms threshold is basically deactivated and the given flusher threshold of 1.7 times the base RTT, takes control. The effect is visible in the graph as the calculated throughput is sub par during half the run. This suggests that an other fixed threshold might be appropriate.

When the base RTT increases, the flusher seems to be more effective at keeping a high throughput. With a base RTT of 100 ms, the impact of the flusher on the throughput is minimised. This is shown in figure 6.29 on page 92. And it is actually better than PIE when running on a connection using TCP Reno.

A probable reason for this is that the flusher is reacting slower than PIE. This would create fewer packet drops that TCP Reno would react to. In addition, the TCP Reno algorithm is halving its congestion window for each packet drop. The halving of the congestion window would happen more often when the bottleneck queue is short. If TCP Reno did not react to a packet drop by halving its congestion window, or it could get information

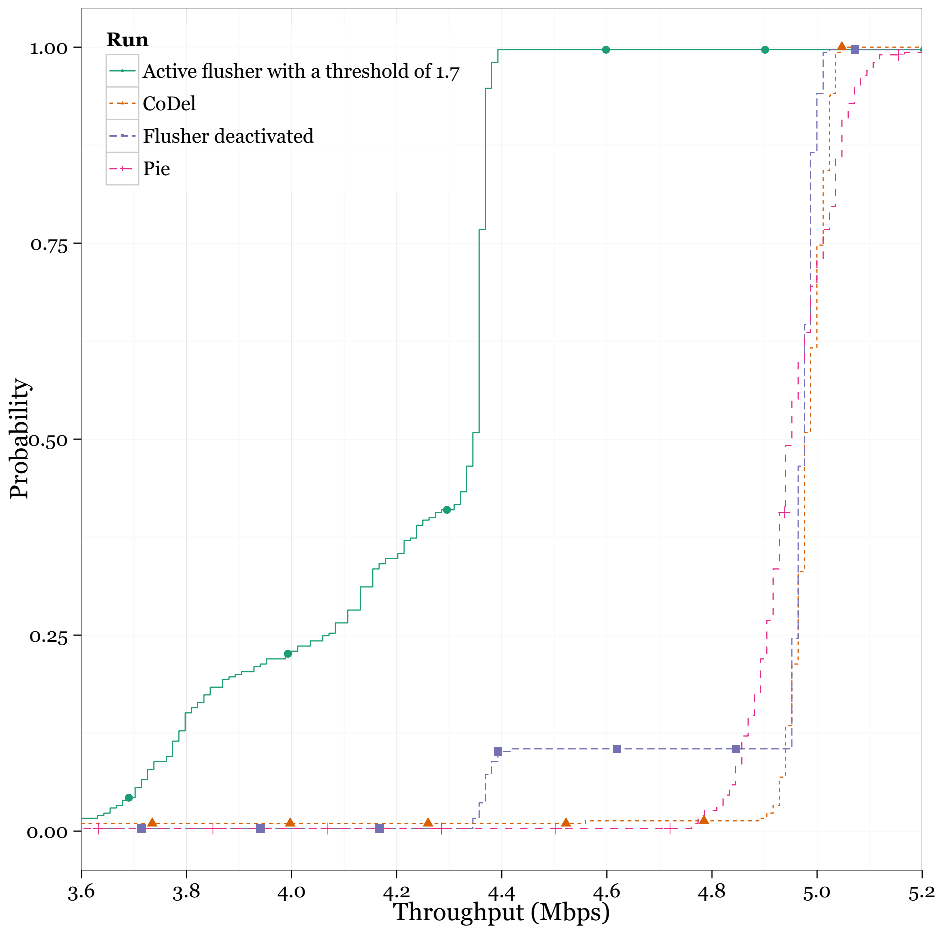


Figure 6.27: Throughputs when using TCP Reno and a network RTT of 10 ms and a queue length of 100 packets.

about how many packets it should drop, the throughput would have been higher.

When running TCP Cubic, the flusher is almost on par with PIE. This is shown in figure 6.30 on page 93. A probable reason for this is that the buffer needs fewer packets to fill, and TCP would therefore react to a filled buffer. If the base RTT is low, a filled buffer takes proportionally longer to drain and the TCP packets become a smaller percentage of the total amount of transfered packets.

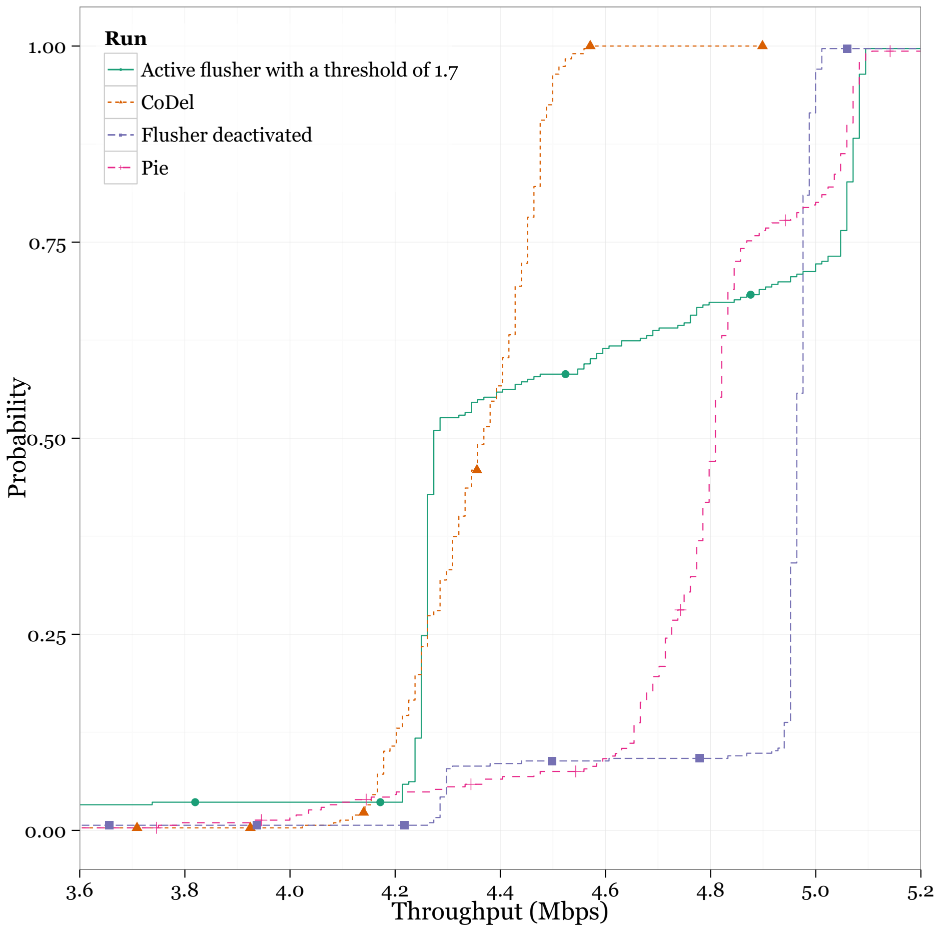


Figure 6.28: Throughputs when using TCP Reno and a network RTT of 50 ms and a queue length of 100 packets.

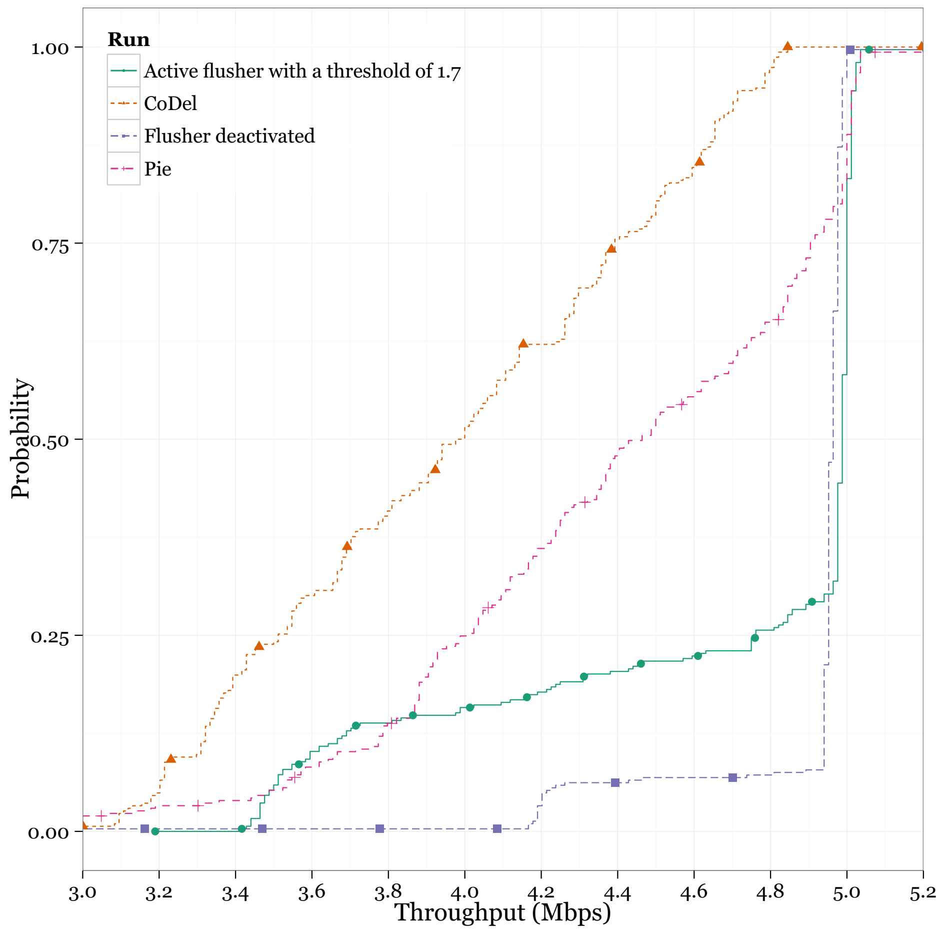


Figure 6.29: Throughputs when using TCP Reno and a network RTT of 100 ms and a queue length of 100 packets.

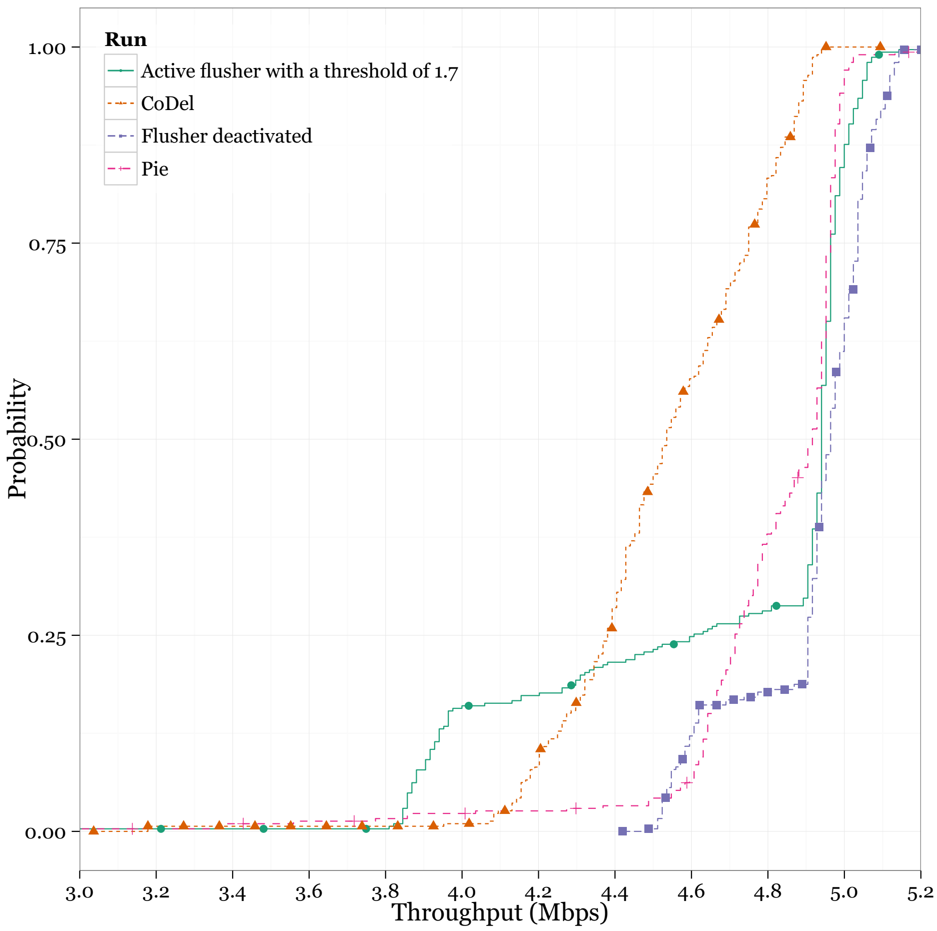


Figure 6.30: Throughputs when using TCP Cubic and a network RTT of 100 ms and a queue length of 100 packets.

6.4 Conclusion

The goal of this thesis was to create a 'queue flusher' that detects queues which induce big delays in a subnetwork. As the queues are forming at the bottleneck it is detected as an induced delay in the network. The induced delays may range from non-negligible milliseconds to several seconds. A delay of several seconds is a bad thing from the user's point of view as the user which is performing a task may give it up due to the delay as the user have limited patience.

Given the fact that a router may be a bottleneck in several networks, a queue which is affecting one stream is likely affecting every other streams using that specific bottleneck.

Overly large buffers in routers are an effect of the fact that memory has become cheap, thus manufactures include vast amounts of memory in their routers, allowing queues to grow and cause delay in the network.

Active queue managers, as discussed in section 3.6 on page 19, are one solution to the problem of overly large queues, 'standing queues' and 'buffer bloat'. Because they are running on the routers, the AQM algorithms have an inside view of what is happening on the router making them capable to limit the queues in real time as the queues are expanding. The AQMs need some configuration, but the two AQMs described in this thesis are mostly working fine at their default configurations.

If there already are solutions to the 'standing queues' problem, why do we need another? The main problem with AQMs is that they are running on the routers. Given the number of routers out there, implementing AQM algorithms on them is a huge task which would take some time. As the internet grows, it is still a valid problem.

The solution presented in this thesis is a queue flusher which is capable of running on any device in a network. It monitors the traffic passing through a device and analyses it. When a possible queue is detected, the 'queue flusher' tries to get rid of the queue by sending a lot of unnecessary UDP packets to the receiver in an attempt to overflow the buffer. A flooded buffer will incur packet drops which will reduce the congestion window on the TCP streams passing through the bottleneck.

The 'queue flusher' solution differs from the AQMs due to the way it works. AQMs are running on the routers, giving them the advantage of being able to control the queue directly. The 'queue flusher' on the other hand, sees the network as a 'black box' as it knows what is going in and out of it, but not how it works. How the 'black box' network works is not known to the flusher, hence it has to estimate the needed amount of packets to send for the TCP streams to respond. As the flusher is able to measure the effect, it does not operate in complete darkness. This means the flusher is able to reduce the amount of packets when needed.

During testing it became clear that the 'queue flusher' improves the responsiveness of the network while reducing the mean RTT, which was the wanted result. The best results were achieved with TCP Reno streams as their congestion window does not change its size as rapidly as the congestion window in a TCP Cubic stream. When using a TCP Cubic stream the results were not as good, which indicates that it would be preferable to do further work on the flusher to better handle TCP Cubic streams.

The flusher was tested with different RTT and buffer sizes in order to get a picture of the effectiveness. The tests were also done using a different flood threshold, which is a multiple of the measured base RTT in the network. A threshold should not be too high as it would lead to the flusher being less effective, nor should the threshold be too low as it would force the flusher to be more active, in extreme cases crippling to the network throughput. A decent threshold value seemed to be a threshold of $1.7 \times$ base RTT as it would not be too aggressive or too loose, enabling the flusher to flush the queue before it becomes a problem.

When compared to the AQM algorithms, the flusher was on par with the PIE AQM when it was at its best. On average the flusher improved the situation when it was active compared to when it was passive. But the existing AQMs are better due to their inside knowledge.

Does the 'queue flusher' work? Yes, it does. It does a decent job as a temporary fix to the standing queue problem. In the long run the AQM algorithms are a better solution due to the shortcomings of the flusher and the fact that the deployment of the queue flusher would also take some time. Even though the flusher is capable of running at network endpoints, it would still need to be deployed on a large scale. Thus, it is subject to the same challenges as existing AQMs. If AQMs like CoDel or PIE were enabled by default in customer hardware it would improve the overall responsiveness in the edges of the internet, which is where the queues need to be handled. The 'queue flusher' may serve as a temporary solution until active queue managers are deployed.

6.5 Future directions

This section addresses directions for future work to be done in order to improve the efficiency of the flusher and to check the flusher correctness.

6.5.1 Use a different number of samples before a flush is initiated

The flusher currently works by flushing after at least 3 samples of 10 packets have been filled and the minimum RTT in all samples is higher than the flusher threshold. It could be interesting to see if another number of successive samples has an impact on the flusher effectiveness.

Another sample size may also be preferable.

6.5.2 Use real hardware

The queue flusher should be tested using real hardware in order to test its effectiveness. The results presented in this thesis is created from logs from runs where the flusher is tested on virtual hardware.

Testing the flusher on real hardware would help test its correctness as the results would be more realistic.

6.5.3 Test with several concurrent streams

The flusher's performance in a system with several concurrent streams should be tested. This would be a natural step before going to the next step of checking its performance with streams originating from other places in the network.

To do this test, a slight modification of the original test should be done. An additional stream should start from the node and send to the receiver. The flusher should not start two concurrent flushes, but rather flush based on the RTT of one of the streams.

6.5.4 Test the flusher's effectiveness when the queue is created by another stream originating from another place in the network

An interesting case to test is the flusher effectiveness when another stream is creating a queue. This scenario is displayed in figure 6.31 on the facing page. It displays a topology for a network where two streams converge in a bottleneck. The streams originate from the nodes N_1 and N_2 while the bottleneck buffer is positioned at R_1.

If the flusher runs on N_1, it also needs a stream to originate from N_1 in order to measure the connection base RTT. There is already a stream originating from N_2 which is causing a standing queue at R_1. A standing queue is a 'bad' queue which induces delay and impairs network responsiveness (this is described in chapter 3 on page 13).

The stream originating from N_1, where the flusher is, should experience high delays. If the RTT never exceeds the set flusher threshold, the flusher tries to flush after 30 seconds. This flushing should have an effect on RTT if the flusher has not flushed earlier.

6.5.5 Test the flusher performance when it runs at a random node in the network

In addition to test the flusher effectiveness with concurrent streams originating from different places in the network, it would be interesting to see how th flusher responds when it is running at a router-node. The router-node could be a random router or the bottleneck node.

Using figure 6.31 on the facing page, the flusher could be set to run on R_1 or R_2. The flusher may not be able to flush when it is running on the bottleneck node, but it should be able to flush when it is running on another node.

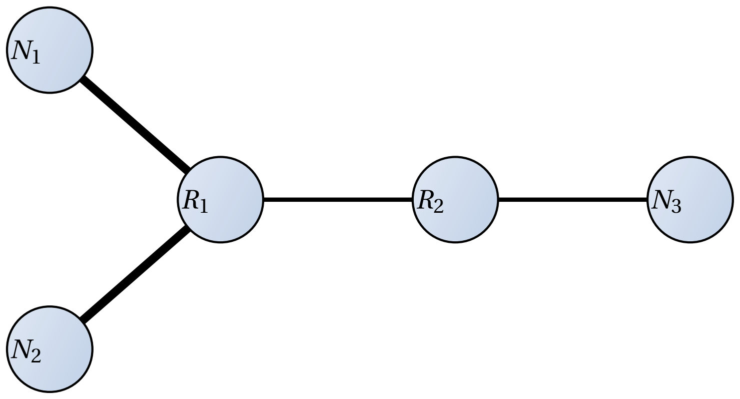


Figure 6.31: Minimal topology for testing the flusher in a network where the queue is created by another stream originating from another place in the network.

Bibliography

[1] R. Adams. 'Active Queue Management: A Survey'. In: *Communications Surveys Tutorials, IEEE* 15.3 (Mar. 2013), pp. 1425–1476. ISSN: 1553-877X. DOI: 10.1109/SURV.2012.082212.00018.

[2] J. Ahrenholz. 'Comparison of CORE network emulation platforms'. In: *MILITARY COMMUNICATIONS CONFERENCE, 2010 - MILCOM 2010*. Oct. 2010, pp. 166–171. DOI: 10.1109/MILCOM.2010.5680218. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5680218.

[3] K. Nichols et al. *Controlled Delay Active Queue Management*. Internet-Draft draft-ietf-aqm-codel-00.txt. IETF Secretariat, 24th Oct. 2014.

[4] R. Pan et al. *PIE: A Lightweight Control Scheme To Address the Bufferbloat Problem*. Internet-Draft draft-ietf-aqm-pie-00.txt. IETF Secretariat, 27th Oct. 2014.

[5] T Hoeiland-Joergensen et al. *FlowQueue-Codel*. Internet-Draft draft-hoeiland-joergensen-aqm-fq-codel-01.txt. IETF Secretariat, 10th Nov. 2014.

[6] M. Allman, S. Floyd and C. Partridge. *Increasing TCP's Initial Window*. RFC 3390 (Proposed Standard). Internet Engineering Task Force, Oct. 2002. URL: http://www.ietf.org/rfc/rfc3390.txt.

[7] M. Bashyam, M. Jethanandani and A. Ramaiah. *TCP Sender Clarification for Persist Condition*. RFC 6429 (Informational). Internet Engineering Task Force, Dec. 2011. URL: http://www.ietf.org/rfc/rfc6429.txt.

[8] R. Braden. *Requirements for Internet Hosts - Communication Layers*. RFC 1122 (INTERNET STANDARD). Updated by RFCs 1349, 4379, 5884, 6093, 6298, 6633, 6864. Internet Engineering Task Force, Oct. 1989. URL: http://www.ietf.org/rfc/rfc1122.txt.

[9] J. Chu et al. *Increasing TCP's Initial Window*. RFC 6928 (Experimental). Internet Engineering Task Force, Apr. 2013. URL: http://www.ietf.org/rfc/rfc6928.txt.

[10] D.D. Clark. *Window and Acknowledgement Strategy in TCP*. RFC 813. Internet Engineering Task Force, July 1982. URL: http://www.ietf.org/rfc/rfc813.txt.

[11] *CoDel(8) CoDel - Controlled-Delay Active Queue Management algorithm*. 12th May 2012.

[12] *Fedora*. URL: https://getfedora.com.

[13] Louis Martin Garcia. 'Programming with Libpcap – Sniffing the network from our application'. In: *Hakin9* Vol 3. No. 2 (2/2008). URL: http://recursos.aldabaknocking.com/libpcapHakin9LuisMartinGarcia.pdf.

[14] Jim Gettys and Kathleen Nichols. 'Bufferbloat: Dark Buffers in the Internet'. In: *Commun. ACM* 55.1 (Jan. 2012), pp. 57–65. ISSN: 0001-0782. DOI: 10.1145/2063176.2063196. URL: http://doi.acm.org/10.1145/2063176.2063196.

[15] Stephen Hemminger. *[Linux.conf.au 2013] - Bufferbloat from a plumber's point of view*. 2013. URL: http://mirror.linux.org.au/pub/linux.conf.au/2013/webm/Bufferbloat_from_a_Plumbers_point_of_view.webm (visited on 22/01/2015).

[16] Toke Høiland-Jørgensen. *Battling Bufferbloat: An Experimental Comparison of Four Approaches to Queue Management in Linux*. Tech. rep. Roskilde, Denmark: Roskilde University, 17th Dec. 2012. URL: http://rudar.ruc.dk/handle/1800/9322.

[17] A. Kuzmanovic et al. *Adding Explicit Congestion Notification (ECN) Capability to TCP's SYN/ACK Packets*. RFC 5562 (Experimental). Internet Engineering Task Force, June 2009. URL: http://www.ietf.org/rfc/rfc5562.txt.

[18] LARTC. *Linux Advanced Routing & Traffic Control*. 19th May 2012. URL: http://www.lartc.org/ (visited on 28/02/2014).

[19] Marek Maowidzki. *SIMULATION-BASED STUDY OF ECN PERFORMANCE IN RED NETWORKS*. URL: http://maom_onet.republika.pl/papers/ecn/ecn-spects03.pdf (visited on 25/04/2015).

[20] M. Mathis, J. Heffner and R. Raghunarayan. *TCP Extended Statistics MIB*. RFC 4898 (Proposed Standard). Internet Engineering Task Force, May 2007. URL: http://www.ietf.org/rfc/rfc4898.txt.

[21] David S. Moore and George P. McCabe. *Introduction to the Practice of Statistics*. W. H. Freeman, 2006. ISBN: 0-7167-6400-8.

[22] J. Nagle. *Congestion Control in IP/TCP Internetworks*. RFC 896. Internet Engineering Task Force, Jan. 1984. URL: http://www.ietf.org/rfc/rfc896.txt.

[23] Office of Naval Research. *Common Open Research Emulator (CORE)*. 2014. URL: http://www.nrl.navy.mil/itd/ncs/products/core (visited on 06/01/2015).

[24] Kathleen Nichols and Van Jacobson. 'Controlling Queue Delay'. In: *Queue* 10.5 (May 2012), 20:20–20:34. ISSN: 1542-7730. DOI: 10.1145/2208917.2209336. URL: http://doi.acm.org/10.1145/2208917.2209336.

[25] *PIE(8) PIE - Proportional Integral controller-Enhanced AQM algorithm.* 16th Jan. 2014.

[26] J. Postel. *Internet Protocol.* RFC 791 (INTERNET STANDARD). Updated by RFCs 1349, 2474, 6864. Internet Engineering Task Force, Sept. 1981. URL: http://www.ietf.org/rfc/rfc791.txt.

[27] The Linux Information Project. *Root definition.* 27th Oct. 2007. URL: http://www.linfo.org/root.html (visited on 25/04/2015).

[28] K. Ramakrishnan, S. Floyd and D. Black. *The Addition of Explicit Congestion Notification (ECN) to IP.* RFC 3168 (Proposed Standard). Updated by RFCs 4301, 6040. Internet Engineering Task Force, Sept. 2001. URL: http://www.ietf.org/rfc/rfc3168.txt.

[29] P. Read, M.P. Meyer and Gamma Group. *Restoration of Motion Picture Film.* Butterworth-Heinemann series in conservation and museology. Butterworth-Heinemann, 2000. ISBN: 9780750627931. URL: http://books.google.no/books?id=OKZzxUV33zUC.

[30] David Ros and Michael Welzl. 'Assessing LEDBAT's Delay Impact.' In: *IEEE Communications Letters* 17.5 (2013), pp. 1044–1047. URL: https://heim.ifi.uio.no/michawe/research/publications/ledbat-impact-letters.pdf.

[31] S. Shalunov et al. *Low Extra Delay Background Transport (LEDBAT).* RFC 6817 (Experimental). Internet Engineering Task Force, Dec. 2012. URL: http://www.ietf.org/rfc/rfc6817.txt.

[32] Dan Siemon. 'Queueing in the Linux Network Stack'. In: *Linux J.* 2013.231 (July 2013). ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=2509948.2509950.

[33] W. Stevens. *TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms.* RFC 2001 (Proposed Standard). Obsoleted by RFC 2581. Internet Engineering Task Force, Jan. 1997. URL: http://www.ietf.org/rfc/rfc2001.txt.

[34] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks.* Fifth edition. Pearson Education, Limited, 2011. ISBN: 978-0-13-255317-9. URL: http://books.google.no/books?id=dHQMkgAACAAJ.

[35] *TCP(7) Linux Programmer's Manual.* 31st Mar. 2014.

[36] *The FreeBSD Project.* URL: https://www.freebsd.org/.

[37] Hadley Wickham. *ggplot2: elegant graphics for data analysis.* Springer New York, 2009. ISBN: 978-0-387-98140-6. URL: http://had.co.nz/ggplot2/book.

[38] Wikipedia. *Fedora (operating system) — Wikipedia, The Free Encyclopedia.* [Online; accessed 14-January-2015]. 2014. URL: http://en.wikipedia.org/w/index.php?title=Fedora_(operating_system)&oldid=638555212 (visited on 14/01/2015).

[39] Wikipedia. *FreeBSD — Wikipedia, The Free Encyclopedia*. [Online; accessed 14-January-2015]. 2015. URL: http://en.wikipedia.org/w/index.php?title=FreeBSD&oldid=640826741 (visited on 14/01/2015).

[40] Wikipedia. *Iperf — Wikipedia, The Free Encyclopedia*. [Online; accessed 8-January-2015]. 2014. URL: http://en.wikipedia.org/w/index.php?title=Iperf&oldid=638979459 (visited on 08/01/2015).

[41] Wikipedia. *IPv4 — Wikipedia, The Free Encyclopedia*. [Online; accessed 16-January-2014]. 2014. URL: http://en.wikipedia.org/w/index.php?title=IPv4&oldid=590162375 (visited on 16/01/2014).

[42] Wikipedia. *Latency (audio) — Wikipedia, The Free Encyclopedia*. [Online; accessed 24-November-2014]. 2014. URL: http://en.wikipedia.org/w/index.php?title=Latency_(audio)&oldid=635170457.

[43] Wikipedia. *Silly window syndrome — Wikipedia, The Free Encyclopedia*. [Online; accessed 26-November-2014]. 2014. URL: http://en.wikipedia.org/w/index.php?title=Silly_window_syndrome&oldid=601129926.

[44] Wikipedia. *Superuser — Wikipedia, The Free Encyclopedia*. 2015. URL: http://en.wikipedia.org/w/index.php?title=Superuser&oldid=656626639 (visited on 25/04/2015).

[45] Wikipedia. *Transmission Control Protocol — Wikipedia, The Free Encyclopedia*. [Online; accessed 28-January-2014]. 2014. URL: http://en.wikipedia.org/w/index.php?title=Transmission_Control_Protocol&oldid=592471764 (visited on 28/01/2014).

Glossary

bandwidth the measure of the capacity of a circuit or channel. More specifically, bandwidth refers (1) to the total frequency range on the available carrier in Hertz (Hz) for the transmission of data, or (2) the capacity of a circuit in bits per second (bps). There is a direct relationship between the bandwidth of an analog circuit or channel and both its frequency and the difference between the minimum and maximum frequencies supported. Although the information signal (bandwidth usable for data transmission) does not occupy the total capacity of a circuit, it generally and ideally occupies most of it.The balance of the capacity of the circuit may be used for various signaling and control (overhead) purposes. In other words, the total signaling rate of the circuit typically is greater than the effective transmission rate. In an analog transmission system, bandwidth is measured in Hertz (Hz). In a digital system, bandwidth is measured in bit per second (bps). 14, 20, 45

congestion window an opening or opportunity for passage of data frames or packets without the requirement for an acknowledgement from the receiving device. 11, 14–16, 22, 25, 27, 56, 60, 62–65, 68, 69, 71, 73, 74, 78, 80, 84–86, 89, 94

hertz the measurement of frequency, which previous to 1930 was expressed as cycles per second (cps). 17

jitter variability in latency of a block, cell, frame, packet, or other message unit. Data message units can suffer jitter not only due to issues of signal jitter, but also because they may encounter different levels of congestion, which may cause them to spend different amounts of time in queues.These factors, and others, contribute to jitter. Some applications, such as e-mail, are tolerant of jitter, while other applications, such as real-time, uncompressed voice, are highly intolerant of jitter. 46–48, 86

network topology the physical and logical structure of a network. Physical topology refers to the physical layout of a network, specifically the physical positioning of the nodes and the circuits that interconnect them. Logical topology refers to the manner in which devices logically interconnect in a network, and may differ considerably from the physical topology. For example, an Ethernet LAN segment may comprise a number of workstations and peripheral devices that interconnect through a hub, with each device connecting directly to a hub port.The physical topology is that of a star, but the logical topology is that of a bus. That is to say that, although the devices connect to the hub over circuits that emanate from the hub like the rays of a star, they interconnect through a collapsed bus, or common electrical path, housed within the hub. LAN and WAN topologies variously include bus, mesh, partial mesh, ring, star, and tree. 53

packet a collection of data, with a header and checksum. 55

round robin a way of traversing a list, where each item is changed in succession. 16, 19

throughput the amount of useful data, user data, or payload that can be processed by, passed through, or otherwise put through a system or system element when operating at maximum capacity. In this sense, overhead, i.e., signaling and control data, is of no relevance except for the fact that it reduces the payload and, therefore, the throughput. Throughput is always less than bandwidth. In other words, the transmission rate, or data rate, is always less than the signaling rate. 9, 32, 40, 41, 45–48, 61, 62, 75, 78–82, 88, 89

Acronyms

ACK	acknowledgement.	8, 9, 11, 14–17, 20, 22, 32, 36–43, 46, 47, 55, 56, 62, 87
API	application programming interface.	34, 56
AQM	active queue manager.	xi, 20–25, 27, 28, 33, 45, 62, 75, 77, 83–89, 94, 95
BDP	bandwidth delay product.	8, 41, 42, 44, 48, 62, 63, 65, 71, 80
CORE	common open research emulator.	53, 58, 59
ECN	explicit congestion notification.	20, 22, 23, 27
EWMA	exponentially weighed moving average.	39, 44, 57, 65
FIFO	first in, first out.	17–19, 33, 75, 85
GUI	graphical user interface.	53
HTB	hierarchial token bucket.	54, 59
HTTP	hypertext transfer protocol.	6
IP	internet protocol.	5, 7, 8, 19, 32, 36, 54, 56, 58, 59, 83
KB	kilobyte.	7
LXC	Linux container.	53
Mbps	megabit per second.	10, 11, 14, 17, 23, 73
MIMO	multiple-input-multiple-output.	25
MTU	maximum transmission unit.	7
NIC	network interface controller.	13
NS-3	network simulator.	53
OS	operating system.	33, 35, 36, 53
path MTU	path MTU.	7
PID	proportional-integral-derivative.	24
qdisc	queueing discipline.	18, 19, 54, 59
QoS	quality of service.	16
RED	random early detection.	20
RTT	round-trip time.	8, 10, 11, 14, 15, 17, 18, 20, 24, 25, 27, 32–34, 36, 38–48, 56–59, 61–96
SISO	single-input-single-output.	24
SPP	synthetic packet pairs.	55–57, 59, 61
ssthresh	slow start threshold.	11
SWS	silly window syndrome.	9, 10
TBF	token bucket filter.	54

TCP Transmission Control Protocol. 5, 7–11, 14, 15, 17–19, 21, 22, 25, 27, 32–34, 36, 38, 39, 43, 55–57, 61–73, 75, 78–80, 84–95

UDP User Datagram Protocol. 5, 7, 17, 25, 33, 34, 42, 48, 55, 68, 74, 78, 84, 94