UiO : Department of Informatics
University of Oslo

Investigate reordering in Linux TCP

Mads Johannessen

Master's Thesis Autumn 2015

[simula . research laboratory]

Investigate reordering in Linux TCP

Mads Johannessen

August 3, 2015

Abstract

When you're using your mobile device, you can connect to several wireless networks at the same time, such as 3G (HSPA) / 4G (LTE) and WiFi. These links could be used at the same time to increase the download speed or make streaming of video more reliable. These links introduce differences in delay and TCP should fail if you simply try to aggregate them at the network layer because this will also introduce network reordering in relation to the difference in network delay.

There are several existing solutions out there that tries to fix this, but if you simply use Linux and transparently send packets over two different network links which has a large difference in network delay, you will notice that the TCP connection can maintain a high throughput which corresponds to the sum of possible throughput for both links. A patch from August 2013 has made the situation even better, but how does it work?

In this thesis, we have investigated how Linux TCP is robust against network reordering. We have also done extensive testing of Linux TCP's performance where we have transparently sent packets from a TCP connection over two different network paths which differs in both latency and delay.

Acknowledgements

I would like to thank my supervisors Prof. Carsten Griwodz and Prof. Pål Halvorsen for guiding me through my studies and taking part in many discussions.

I would also like to thank Simula research laboratory for providing a great place to conduct my studies.

And finally, I would like to thank my fellow students at Media Performance Group, which has been to good help related to my thesis and now also as some of my new friends.

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Introduction

1.1 Background

A recent PhD thesis [25] from 2011 by Kaspar disclosed an unexpected behavior in Linux TCP in the presence of heavy but regular packet reordering. The situation occurs when the TCP sender transparently sends packets over two different network links which has a large difference in network delay. This would for example be a WiFi network and a 3G (HSPA) network as shown in figure 1.1. By doing this, the TCP sender would send packets in order and the TCP receiver would naturally receive them out of order because of the difference in latency between the network links.

If the latency difference is high, the TCP receiver wouldn't receive any packets traversing the network path to the link with the highest latency before a given amount of packets traversing the network path to the link with the lowest latency resulting in duplicate acknowledgements sent to the TCP sender which would trigger the fast retransmit / fast recovery algorithm which would again send a dubious retransmission and slow down sending speed. This is the expected behaviour of TCP since it now thinks that a packet has been lost.

In Linux TCP however, it learns after an arbitrary time, which would easily be half an hour or more in Kaspar 's case, that these duplicate acknowledgements is not caused by loss, and the sending speed suddenly increases to the sum of the possible speed over both links.

The main goal for this thesis is to discover the true origin of the effect and to see how the situation is with a newer version of the Linux kernel.

1.2 Problem Statement

To discover the true origin of the effect of how Linux TCP is robust against network reordering, we needed to first recreate Kaspar's testbed used in [25] and investigate Linux TCP behavior to a higher extent.

To see how good the network layer aggregation is performing, we needed to compared it against the most promising link aggregation protocol out there, which is Multi Path TCP.

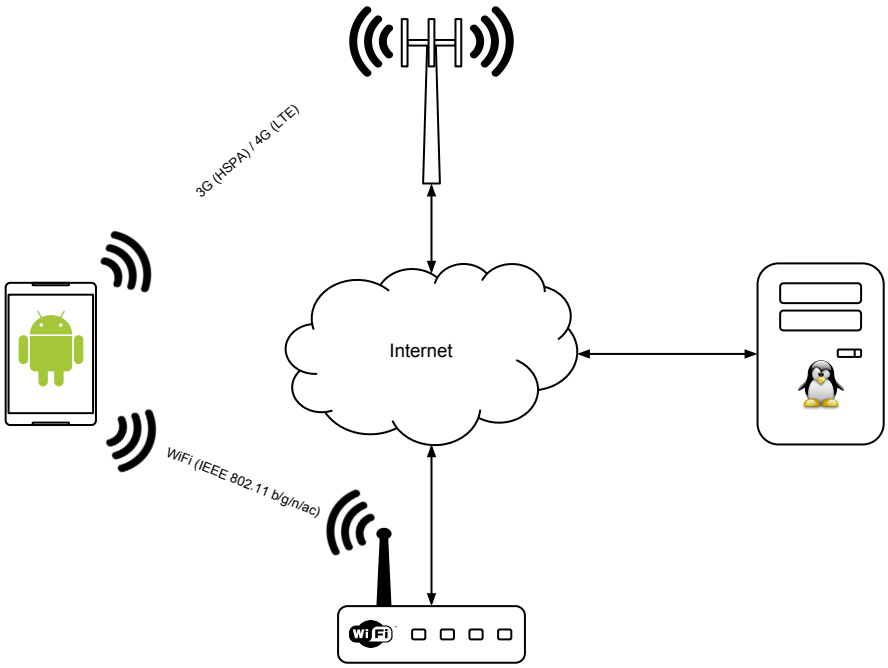


Figure 1.1: An example of the available bandwidth we could potentially utilize in today's smartphones

We can than summarize this into three problems we wanted to solve in this thesis as listed below:

1. Recreate the testbed Kaspar used in [25] and test this out on a newer version of the Linux kernel.
2. Investigate how Linux TCP is robust against network reordering.
3. Compare the performance of our network layer aggregation against MPTCP with different test cases where we experiment with the ratio and order of bandwidth and delay to the aggregated paths.

1.3 Research Method

In this thesis we follow the design paradigm as described in [12] by Comer et al. This entails the following:

1. Stating the requirements
2. Stating the specifications
3. Design and implement the system
4. Evaluate the system

The system we are implementing and evaluating in this thesis is a sender and a receiver side Linux Kernel modules that enables us to transparently send data from a TCP connection over two network paths.

1.4 Main Contributions

- A sender side Linux Kernel module to filter out and diverge a subset of the IP packets to a selected TCP connection.
- A receiver side Linux Kernel module to filter out and merge all IP packets received at a selected address and port number to another selected address and port number.
- A tool to analyze TCP sequence numbers to find out how much a packet is displaced at the receiver.
- Various of BASH scripts to enable the receiver to communicate over multiple network interfaces.
- Evaluation of network layer aggregation in regards to:
  - How much network reordering is introduced by using network layer aggregation
  - How Linux TCP enables to make it self robust against network reordering.
  - How good it performs against MPTCP

1.5 Outline

The remaining part of this thesis is organized as follows:

- Chapter 2: **Related Works**
  - In this chapter we have provided an overview of existing solutions in regards to multi-path aggregation, and we have also looked into some research on have to make TCP reordering robust.
- Chapter 3: **TCP**
  - In this chapter we have looked into the Transmission Control Protocol standard: what it provides, different extensions and bug-fixes and also how its implemented in Linux.
- Chapter 4: **Robustness against network reordering in Linux TCP**
  - In this chapter we have looked into how Linux TCP is robust against network reordering.
- Chapter 5: **Network Emulation Testbed**

  - In this chapter we have described how we designed and implemented a similar testbed Kaspar used in [25] with some minor modifications in regards to increasing the performance and making our results more trustworthy. All the tools and metrics we have used in this thesis is also mentioned here.

- Chapter 6: **Results**
  - In this chapter we have summarized all our results in regards to network reordering and performance measurements.

- Chapter 7: **Conclusion**
  - In this chapter we have summarized our work.

- Chapter 8: **Future Work**
  - In this chapter we have listed some future work.

Chapter 2

Related Works

2.1 Multi-Path Aggregation

In this chapter we will go through some existing solutions related to enabling multi-homed hosts to utilize all their available links structured by the Internet Protocol Layer they operate on in section 2.1.1, 2.1.2 2.1.3 and 2.1.4.

2.1.1 Data Link Layer

Multi link Point-to-Point Protocol (MPPP)

MPPP is specified in RFC1990 [37] as a data-striping protocol where logical links are created by bonding multiple physical links into a bundle. The logical links may include various types of link technology. The binding of links is completely transparent to higher layer protocols, they only use the logical link, not knowing that their data is striped across multiple links.

MPPP uses the extensible negotiation option in Link Control Protocol (LCP) as defined in RFC1661 [36] to indicate its peer that it is capable of combining multiple links into a bundle.

Since MPPP is a point-to-point protocol, it has to originate and terminate on the same pair of endpoints. Both endpoints must also support MPPP, since each endpoint is responsible of splitting and recombining of data.

2.1.2 Network Layer

Equal-Cost Multi-Path routing (ECMP)

Typical routing in the network layer is based on Open Shortest Path First (OSPF) algorithms. They all lack the ability to balance the traffic if there exist multiple paths to the same destination. That's why ECMP was developed.

In ECMP as described in RFC2991 [40] and analyzed in RFC2992 [21] each router calculates multiple shortest paths to a destination. When packets arrives, the router makes a hash value based on the packet header, and uses this value to choose which path to forward the packet on.

The reason for hashing the header is to mitigate the reordering problem when packets belonging to the same packet stream traverses different routes. Hashing will mitigate this problem,so that all packets belonging to the same packet stream will always take the same path. A drawback here is that we can have a path that is congested by other sender/senders and at the same time there are other paths to the same destination which is not utilized. This makes the traffic unbalanced.

To solve this problem it was suggested by Xi, Liu, and Chao in [41] that we can remotely control the path taken and redirect our traffic over a different path. Just by probing the network by manipulating the port numbers, we can learn more about the network and which path it is traversing. We can then choose the best path from source to destination from the knowledge we obtained.

This works since the hash values differs with different port numbers, and the router may forward the stream on a different path. The probing and manipulation of port numbers is proposed done in either the transport layer or the application layer.

2.1.3 Transport Layer

Stream Control Transmission Protocol (SCTP)

Stream Control Transmission Protocol (SCTP) is specified in RFC2960 [39], and is a protocol where we communicate over data-streams between two endpoints as shown in figure 2.1. Each data-stream is identified with its own identification number and each stream is sending and receiving SCTP packets.

A SCTP packet as shown in figure 2.2 contains a common header and multiple chunks. The common header as shown in figure 2.3 contains a source port, destination port, verification tag and a checksum field. The source and destination port number is used in conjunction with the IP addresses to identify which endpoint/application this SCTP packet belongs to. The verification tag is used for validation of sender. The checksum is used for strengthening integrity of the transmission. Figure 2.4 shows us the field format of each chunk and table 2.1 shows us the different chunk types.

If we want to utilize all the available links on a multi-homed host, we simply send data over multiple streams, where each stream utilize different pair of interfaces between the two endpoints.

To set-up a SCTP connection between two endpoints, we must first go though an initialization procedure. In this procedure there is a four way cookie handshake as shown in figure . On idle established connections, a hart beat chunk is sent for maintaining the path manager.

Packet validation and path management handle every incoming SCTP packet before they are handed over for further processing.

The path management is responsible to validate reachability between the two endpoints. If a path goes down the path management must notify the user.

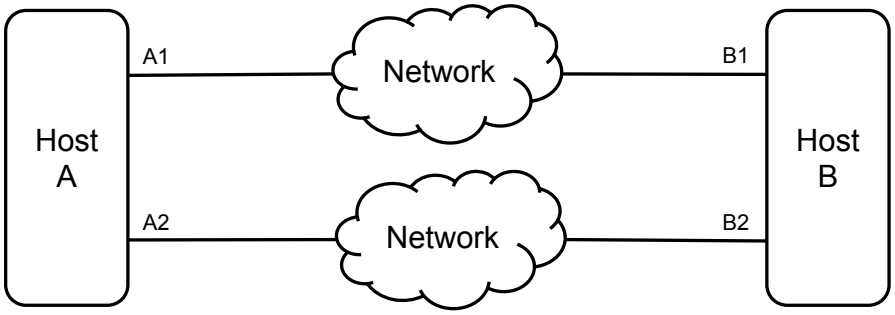


Figure 2.1: SCTP Overview

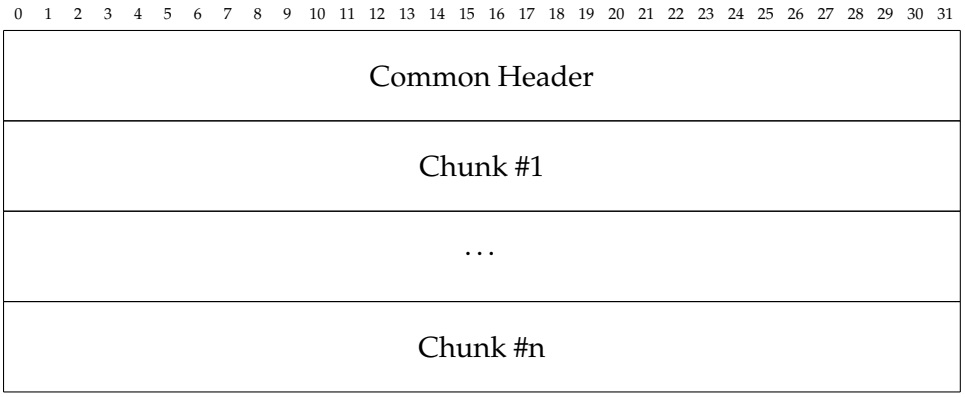


Figure 2.2: SCTP packet Format

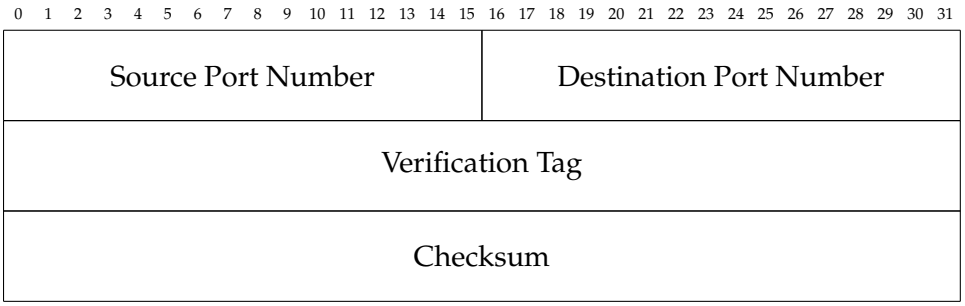


Figure 2.3: SCTP Common Header Format

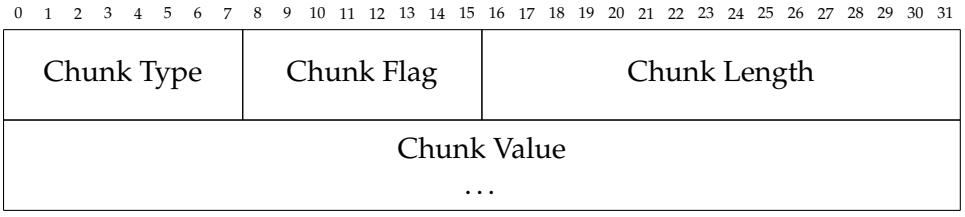


Figure 2.4: SCTP Chunk field Format

ID Value	Chunk Type
0	Payload Data (DATA)
1	Initiation (INIT)
2	Initiation Acknowledgement (INIT ACK)
3	Selective Acknowledgement (SACK)
4	Heartbeat Request (HEARTBEAT)
5	Heartbeat Acknowledgement (HEARTBEAT ACK)
6	Abort (ABORT)
7	Shutdown (SHUTDOWN)
8	Shutdown Acknowledgement (SHUTDOWN ACK)
9	Operation Error (ERROR)
10	State Cookie (COOKIE ECHO)
11	Cookie Acknowledgement (COOKIE ACK)
12	Reserved for Explicit Congestion Notification Echo (ECNE)
13	Reserved for Congestion Window Reduced (CWR)

Table 2.1: SCTP Chunk Types

When we want to send data from the application layer using SCTP, we must use the returned identification number for the path we want to send our data over. If we want to send our data over multiple paths to increase the throughput, we simply stripe our data over multiple SCTP streams. If we want a more resilient connection, we can use one stream as primary and another as backup. For low latency we can send the same data across all available SCTP streams and just use the one that is fastest.

Multi-Path Transmission Control Protocol (MPTCP)

In RFC6182 [16] and RFC6824 [17] MPTCP is specified as an extension to regular/single-path TCP that supports multiple paths between two endpoints see figure 2.5.

MPTCP partition it's data-stream over multiple regular/single-path TCP sub-flows. Each sub-flow has its own sequence number and congestion control. The reason for partition it's data over regular TCP is that it is then supported at any middle-boxes such as NAT's, proxies, and firewalls. This was a problem with SCTP.

To acknowledge a successfully transmitted segment on a sub-flow, a connection-level ACK is transmitted over the path with the lowest Round Trip Time (RTT).

If a path then fails under transmission of a segment, the segment can then be sent over a different sub-flow. When sender has no more data to send, it signals the receiver with a DATA FIN package. When the MPTCP receiver has successfully received all the data a DATA ACK package is sent to the MPTCP sender. The DATA FIN and DATA ACK is very similar to TCP FIN and TCP FIN/ACK, the difference is that it happens on the connection level.

When initializing a MPTCP connection, it first starts off with a

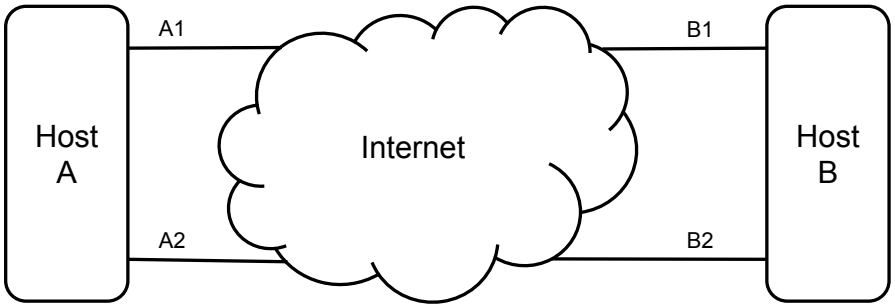


Figure 2.5: MPTCP architectural overview. A1, A2, B1, and B2 are addresses on both end-hosts

single sub-flow where you signal the other endpoint that you want to communicate over MPTCP. Then it is doing a three-way handshake like its done with regular TCP. The difference here is that the SYN, SYN/ACK and ACK segment in the handshake also carries a MP CAPABLE option. At first it just verifies whether or not the other endpoint supports MPTCP. In the MP CAPABLE option there is a key generated by the sender. This key is used when adding or removing sub-flows after connection is established. The MPTCP receiver then generate its own key and sends that back with the senders key in the SYN ACK segment with the MP CAPABLE option if it is capable of supporting MPTCP. If not it falls back to regular TCP. When the sender receive the SYN ACK package, it will send back an ACK segment to verify that the establishment is complete. When adding sub-flows after the establishment, a regular TCP is sent with the option MP JOIN. This is also a three way handshake like in regular TCP. If the MPTCP sender is adding a new sub-flow it must first hash the MPTCP receivers key we got earlier and put that in as a token in the MP JOIN option. With the token the MPTCP receiver knows which MPTCP connection the new sub-flow can be added to. The adding and removing of sub-flows is also called the path management of MPTCP.

2.1.4 Application Layer

Overlay Network

As explained by Han and Jahanian in [19] an overlay network is a virtual network built on top of another network. When construction an overlay network we deploy a set of overlay nodes above the existing IP routing infrastructure. The overlay nodes then builds a routing table between each other. This forms the overlay network. The virtual path between two overlay nodes can consist of multiple physical links if the nodes are multi-homed. By probing the path taken between to overlay nodes, we can find out if we have overlapping paths between host A and B. If we then want to have a primary - backup model between these two host, we can make sure that the primary path and the backup path are disjoint. This is also useful

if we want to stripe data over multiple paths and not congest a node that is common in all the paths.

P2P with network coding

A Peer to Peer network is a decentralized network in which all participating nodes act as both producer and consumer. The P2P network then forms an Overlay Network as discussed in 2.1.4.

When we are transmitting a file in a P2P network, we usually divide the file up in smaller blocks and gossip these blocks to a subset of the overlay network. When a node receives a block it is doing the same thing as the sender, namely forwarding the block to a random subset of the overlay network. The drawback here is that a node can receive the same block multiple times and this can be very inefficient. To optimize this we can use network coding as done by Chiu et al. in [11].

With network coding, we code the incoming block with an already received block and send that new coded block to a random subset of the overlay network. If A possess block b1 and the coded block b1+b2 we can solve for b2 using the received block b1 and the coded block b1+b2 with Gaussian elimination.

If the file is large it can be useful to divide the file up in generations and divide each generation up in blocks.

We can then only solve a block from a coded block within the same generation. If we are not doing this for a large file there will be to many coded blocks.

2.2 Improvements to TCP in relation to network reordering

Since TCP performs poorly on paths that reorder packets, there is done much research on ways to fix it. We have looked into two proposals to address this issue.

2.2.1 Reorder Robust TCP (RR-TCP)

The RR-TCP algorithm as its proposed in [42] by Zhang et al. It is an algorithm that aims to improve TCP's robustness against network reordering by extending the sender to detect and recover from false fast retransmissions with the use of D-SACK information, and to protectively prevent false fast retransmissions by adaptively varying dupthresh. Their algorithm also limits the growth of the dupthresh to avoid unnecessary timeouts.

2.2.2 TCP-NCR

The TCP-NCR algorithm as specified in RFC4653 [6] aims to improve the robustness of TCP against Non-Congestion Events, hence the name.

The algorithm makes changes to the dupthresh variable to fast retransmit / fast recovery algorithm, so that it noe depends on the current flight size. The algorithm also decouples the initial congestion control decisions from retransmission decisions which in some cases delays congestion control changes relative to TCP's current behavior. The algorithm also provides two alternatives for extended limited transmit as listed below:

- **Careful limited transmit** reduces the send rate at approximately the current TCP reduces its send rate. But at the same time, it withholds a retransmission and a final congestion determination for approximately one RTT.
- **Aggressive limited transmit** maintains the sending rate in the face of duplicate ACKs until TCP concludes that the segment is lost, and needs to be retransmitted. TCP-NCR will here delay the retransmission by one RTT compared to TCP's current behavior.

Upon termination of the limited transmit when an ACK advances the cumulative ACK point and before loss recovery is triggered. This signals the sender that the series of duplicate ACKs was in fact due to network reordering and TCP-NCR now resets the congestion windows and slow start threshold.

Chapter 3

TCP

3.1 Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP) specification is described in RFC793 [32]. There is also many more RFCs which adds extensions and bug-fixes, so we are using RFC4614 [13] which is a TCP road map to find the correct RFC to study.

Because TCP is located in the transport layer in the Internet Protocol Stack, it provides data transmission from a process on a source machine to a process on a destination machine. There is mainly two protocols in the transport layer, a connectionless protocol and a connection-oriented one. TCP is a connection-oriented protocol, where as the User Datagram Protocol (UDP) is a connectionless protocol. They both complement each other.

Since TCP is connection-oriented, each connection has three phases: establishment, data transfer and release. TCP uses a TCP header format shown in figure 3.1 to exchange protocol data. We will now study TCP in detail by studying each of these phases.

Fist we will look at the establishment phase in section 3.1.1 and then the release phase in section 3.1.2. In the transfer phase, TCP provides three additional services: Reliable transmission, flow control and congestion control. So we will first look at how TCP can provide a reliable end-to-end byte stream over an unreliable internetwork in section 3.1.3, the flow control protocol for avoiding the source to send more data then the destination can handle in section 3.1.4 and the congestion control protocols for achieving high performance and avoid congestion collapse in section 3.1.5.

3.1.1 Connection Establishment

To allow many processes within a host to use TCP simultaneously, TCP uses a set of addresses or port numbers within each host. Concentrated with the network and host addresses from the IP layer, this forms a socket. A pair of sockets uniquely identifies a connection. When we are establishing a connection we are pairing the socket at each host. To

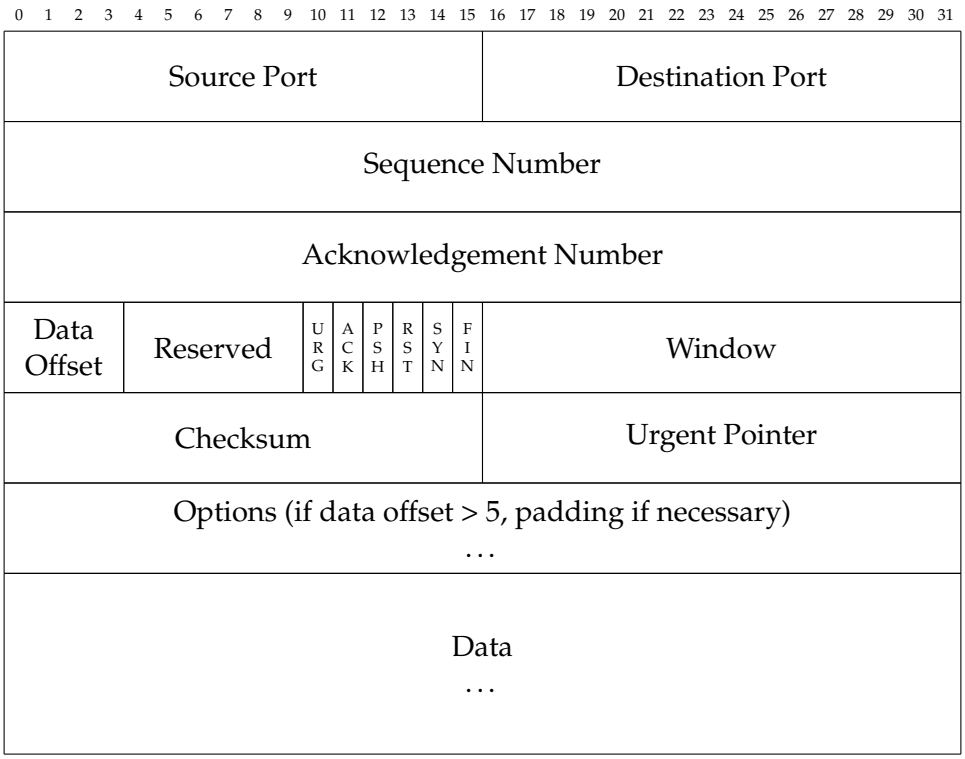


Figure 3.1: TCP Header Format

reconstruct the byte-segments in the same order as they were sent out, TCP uses a sequence number to identify each byte of data. Within the connection establishment, the hosts needs to exchange the sequence number that the data stream will start from. This sequence number is randomly chosen by the host (client process) actively connecting to the passively waiting host (server process).

The server process and client process can also exchange some optional information during the connection establishment. This options can be: maximum segment size (MSS), window scale, if the host is allowing selective acknowledgement and a time-stamp. This is of course an optional specification. If for example the MSS is not specified it defaults to 536-byte payload.

To exchange this information between the server process and client process, TCP uses a three-way handshake shown in figure 3.2. First the client process sends a synchronize segment (SYN). Within this SYN segment, the client process must set the sequence number field to a randomly chosen value. It must mark the SYN bit and add any options field and add the option field size to the data offset field. The client process must also set the receiving port number in the destination field and a source port number in the source field.

When the server receives the SYN segment, it first checks if there is any process listening on that particular destination port number. If not, the server will reply with a segment where the RST bit set to reject the connection and set the destination port to the received source port and

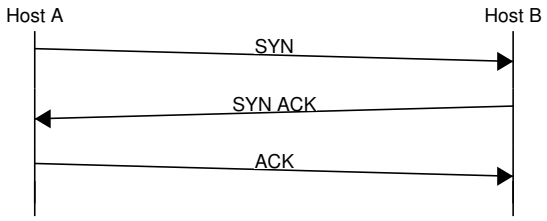


Figure 3.2: TCP three way handshake for initial synchronization

the source port to the received destination port. If there is some process listening on that port, the process is given the incoming segment. The server process can then choose to either reject or accept the connection. If the server process accept the connection, it must send back a synchronize acknowledgement segment (SYN-ACK) to the client process. In this SYN-ACK segment, the server process must set the sequence number field to another random value that it can use if it needs to send any data to the client process. This makes the connection full-duplex since data can be sent in both directions. Both the SYN and ACK bit must be marked, the acknowledgement field must be set to one more then the sequence number received from the client process SYN segment, it can add any optional fields and set the optional filed size in the data offset field.

When the client receives the SYN-ACK segment from the server process, it must send back an acknowledgement segment (ACK) to the server process. In this ACK segment the client must set the sequence number field to the received acknowledgement value, the acknowledgement field must be set to one more then the received sequence number value and only the ACK bit must be set. When the server process receive the ACK segment the connection is considered established at both ends.

3.1.2 Connection Termination

When we are releasing a TCP connection, which is a full duplex connection as described in section 3.1.1, we need to release each simplex connection independently. This makes it also possible to have a half open TCP connection, where just one of the simplex connections is closed.

To release a connection, lets say host A wants to release a connection to host B. A must send a segment with the FIN bit set, this means that A has no more data to transmit to B. When B has acknowledged that segment the connection is terminated in that direction. The data can still continue to flow in the other direction. When both directions is terminated, the connection is released.

As you can see in figure 3.3, we normally need to send four segments to release a connection, one FIN and one ACK for each direction. There is a way to do this in just three segments where we piggyback the second FIN segment to the first ACK segment as shown in figure 3.4. The latter is considered the most common method.

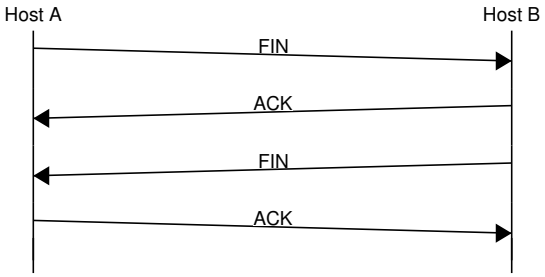


Figure 3.3: TCP four way handshake for connection release

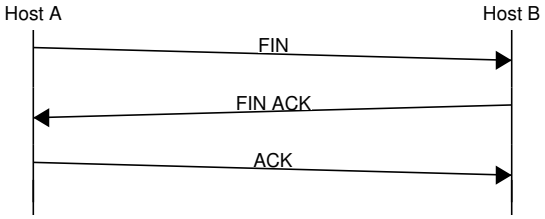


Figure 3.4: TCP three way handshake for connection release

3.1.3 Reliable Transmission

For TCP to provide a reliable end-to-end byte stream over an unreliable internetwork, it must recover from damaged, lost, duplicated or out of ordered segments.

To recover from damaged segments, the TCP sender adds a checksum to each segment. The TCP receiver also makes a checksum of the segment received and compare it with the received checksum. If they don't match, the segment is discarded.

To recover from lost segments, the TCP sender must retransmit any lost segment. This is achieved by adding a time-out interval at the TCP sender. If a positive acknowledgement segment is not received within this interval, the segment is retransmitted.

The TCP receiver uses the sequence numbers to discard any duplicate segments and to correctly order the segments that may be received out of order.

3.1.4 Flow Control

Because TCP is using a sliding window protocol, it needs to enable the TCP receiver to control the amount of data sent by the TCP sender. This is achieved by making the TCP receiver advertise its acceptable window size to the TCP sender. This is done at every ACK segment sent by the TCP receiver. The acceptable window size is its acceptable sequence number after the last segment successfully received. This will indicate how many bytes the TCP sender can have in flight. In flight is transmitted bytes that is not acknowledged.

3.1.5 Congestion Control

The congestion control algorithms for TCP is an extension to TCP as specified in RFC793 [32]. TCP congestion control was first specified in RFC2581 [3], updated in RFC3390 [1] and obsoleted by RFC5681 [2]. These algorithms makes TCP dynamically adjust its sliding window size so that it doesn't cause congestion in the network.

RFC5681 [2] specifies four intertwined congestion control protocols: Slow-start, congestion avoidance, fast retransmit and fast recovery.

Slow-start and congestion avoidance adds two additional state values to the TCP protocol, namely congestion window (cwnd) and slow start threshold (ssthresh). The cwnd state value is the sender-side limit on the amount bytes the sender can have in flight. The ssthresh state value is used to determine if Slow-start or Congestion avoidance algorithm is used to control the data transmission.

Fast retransmit and fast recovery algorithms is used to quickly detect and repair loss by retransmitting the lost packet before the retransmission time-out.

These four algorithms is discussed in more detail below.

Slow-start

To determine the available capacity in the network, TCP must slowly probe the network at the beginning of the transmission or after loss is detected by the retransmission time-out. The slow-start algorithm uses TCP ACKs as a clock to probe new packets into the network. This is a self balancing clock since the TCP receiver can't generate TCP ACKs faster then packets received. During slow-start the TCP senders cwnd is increased by at most the senders maximum segment size (SMSS) every clock cycle as shown in equation 3.1. This will double the TCP senders cwnd every round-trip-time (RTT). It will continue to do so in this way until it reaches/exceeds the ssthresh or when congestion is observed. The initial value of ssthresh is set arbitrarily high and it is reduced in response to congestion as shown in equation 3.2 where FlightSize is the amount of outstanding data in the network.

$$cwnd = cwnd + SMSS \tag{3.1}$$

$$ssthresh = max(FlightSize/2, 2 \times SMSS) \tag{3.2}$$

Congestion Avoidance

During congestion avoidance the TCP senders cwnd is increased by at most SMSS every RTT as shown in equation 3.3.

$$cwnd = cwnd + ((SMSS \times SMSS)/cwnd) \tag{3.3}$$

Fast Retransmit

The fast retransmit is used to detect and repair loss based on the incoming duplicate ACKs. It uses the arrival of three duplicate ACKs as an indication that the TCP segment has been lost. It should then perform an early retransmission (before the retransmission timer-out) and then let fast recovery govern the transmission of new data until a non-duplicate ACK has arrived. The reason for not let slow-start govern the transmission is that the duplicate ACKs does not only indicate that a segment has been lost, it may also indicate that the segment has been reordered due to different routes with different delay in the network.

Fast Recovery

When TCP sender is entering fast recovery it must first set the ssthresh with equation 3.2 which is the same equation used in slow-start. TCP sender then reduces its cwnd to ssthresh plus three times the SMSS to artificially inflate the cwnd with the number of segments buffered by the receiver as shown in equation 3.4. For each additional duplicate ACK the cwnd is increased by SMSS shown in equation 3.1.

$$cwnd = ssthresh + (3 \times SMSS) \tag{3.4}$$

3.1.6 Selective Acknowledgement (SACK)

TCP as specified in RFC793 [32] only provides a cumulative acknowledgement mechanism, and this may preform poorly when multiple packets are lost or reordered in the network within a flight window. The selective acknowledgement (SACK) mechanism provides improvements to this scenario.

SACK is specified in RFC2018 [28] and the extension that enables TCP to handle duplicate SACK (DSACK) is specified in RFC2883 [15].

SACK enables TCP receiver to inform TCP sender with all segments that has been received successfully. This will prevent unnecessary retransmits of segments which have already been received. In this way, TCP sender only needs to retransmit the segments that has actually been lost. Also when multiple packets are lost, and we are only using the cumulative acknowledgement mechanism, TCP will generally lose the ACK-based clock resulting in reduced overall throughput.

The SACK extension to TCP uses two TCP-options. First option is the SACK-permitted which may be sent in the TCP SYN segment to signal the counterpart that SACK option can be used once the connection is established. This will ensure that SACK is permitted in one of the flow directions, the data flowing in the reverse direction can be treated independently. The other option is the SACK-option itself, which may be sent over an established connection once permission has been given by SACK-permitted.

When a TCP receiver is sending a duplicate ACK in response to a lost or reordered packet in the network, a SACK option should be

included if previously permitted. The SACK-option contains up to four 32 bit sequence number (three if time-stamp option is included, as it usually is), whereas the first sequence number is telling the sender which segment that was actually received. The next sequence numbers is a history of previously SACKed segments that is still not ACKed by the cumulative acknowledgement mechanism. This will ensure that non-continues segments buffered by the receiver is reported in at lest three successive SACK options. This will make it more robust to lost ACKs.

When the TCP sender receives an ACK including a SACK option, the TCP sender will turn on the SACKed bits for segments that have been selectively acknowledged. The TCP sender will then skip these segments during any later retransmission. After a retransmission time-out, the TCP sender should turn off all of the SACKed bits, since the time-out may indicate that the TCP receiver has reneged. The TCP sender must also retransmit the segment at the left edge of the window after a retransmission time-out, whether or not that segment has been SACKed.

To enable the TCP receiver to accurately report the reception of duplicate segments, the TCP receiver can include a duplicate-SACK (DSACK) in the ACK segment sent to the TCP sender. If TCP sender doesn't understand DSACKs, the TCP sender will simply discard the DSACK block and continue to process the other SACK blocks as it normally would. Because of this, the use of DSACK doesn't require a separate negotiation between the TCP sender and receiver. When DSACK is used, the first block of the SACK option should be the DSACK block. For the TCP sender to check if the first DSACK block of an ACK segment is acknowledging a duplicate segment, it should compare the first SACK block to the cumulative ACK in the same ACK segment. If the SACK space is less then the cumulative ACK, it is an indication that the segment identified by the first SACK block has been received more than once by the TCP receiver. If the SACK space is greater then the cumulative ACK, then the TCP sender compares the first SACK space with the second SACK space if there is one. If they are equal, the first SACK block is reporting a duplicate segment over the cumulative ACK.

3.1.7 Forward Acknowledgement (FACK)

The forward acknowledgement algorithm (FACK) as described by Mathis and Mahdavi in [27] is an algorithm that improves the TCP congestion control during recovery by keeping an accurate estimate of the amount of data outstanding in the network. With this accurate estimate FACK attempts to preserve TCP's self-clock and reduce the overall burstiness. FACK uses the additional information obtained by the TCP SACK option (see 3.1.6) to measure the outstanding data in the network. Opposed to TCP Reno and TCP Reno + SACK which both attempts to estimate this by assuming that each duplicate ACK received represents one segment that has left the network.

The FACK algorithm adds two new TCP state variables: $snd.fack$ and $retrans\_data$. The $snd.fack$ is updated to reflect the forward-most segment

held by the TCP receiver (hence the name forward acknowledgement). By forward-most we mean the correctly received segment with the highest sequence number. The *retrans_data* is the quantity of outstanding retransmitted segments in the network.

In non-recovery states, the $snd.fack$ variable is updated with the use of the cumulative ACK sent by the TCP receiver. The $snd.fack$ will then be the same as the *snd.una* variable (Send unacknowledged) which is the first unacknowledged segment sent by the TCP sender. During recovery the TCP sender continue to update *snd.una* variable with the use of the cumulative ACK from the TCP receiver, but it utilizes the information contained in the TCP SACK options to update the $snd.fack$ variable.

In [27] *awnd* is defined to be the TCP senders estimate of the actual quantity of data outstanding in the network. In a non-recovery state the computation of *awnd* is shown in equation 3.5. If we are in recovery, retransmitted segments must also be included in the computation of *awnd* as shown in equation 3.6. Using this estimate of outstanding data, the FACK algorithm can regulate the amount of outstanding data in the network to be within one MSS of the current *cwnd*.

$$awnd = snd.nxt - snd.sack \tag{3.5}$$

$$awnd = snd.nxt - snd.sack + retrans\_data \tag{3.6}$$

FACK derives its robustness from the simplicity of updating its state variables. If TCP sender is retransmitting old data, the *retrans_data* variable is increased. If TCP sender is sending new data, TCP sender advances the *snd.nxt* variable. If the TCP sender receives an ACK from the TCP receiver it will either decrease *retrans_data* or advance $snd.fack$. If the TCP sender receives an ACK which advances $snd.fack$ beyond *snd.nxt* and we have a unaccounted retransmitted segment, the TCP sender knows that the segment which has been retransmitted has been lost.

3.1.8 TCP Timers

TCP uses timers to detect inactivity on remote nodes. When the local node is waiting for a response or action from the remote node, TCP will set a time out where to recover in case the response or action is not received. There are seven different timers in TCP in total, and we will briefly described their usage in this section:

Connection Establishment Timer

This timer is set when transmitting the initial SYN segment. If no response is received before the timer runs out, the connection is aborted.

Retransmission timer

This timer is set when transmitting a segment. If no acknowledgement is received for that segment before the timer runs out, TCP will retransmit the

segment and enter loss recovery. To calculate the Retransmission TimeOut (RTO) described in RFC6298 [31], TCP uses the Round Trip Time (RTT) measurement which is measured with the use of a TCP timestamp option described in RFC7323 [7]. Due to the variation of the RTT measurements, RTT measurements are not used directly to calculate RTO, instead the TCP sender maintains two new state variables: Smoothed Round Trip Time (SRTT) and Round Trip Time Variation (RTTVAR). $SRTT$ is an estimation of RTT and $RTTVAR$ is the variance of RTT. Initially RTO is set to 3 seconds, or more then 1 second. When the first RTT measurement arrives, $SRTT$ is set to this value and $RTTVAR$ to be half this measured value. For subsequence RTT measurements $RTTVAR$, $SRTT$ and RTO is calculated as follows in equation 3.7, 3.8 and 3.9 and exclusively in that order. In equation 3.9 G is the clock granularity of seconds and K is the constant value of 4.

$$RTTVAR = \frac{3}{4} \times RTTVAR + \frac{1}{4} \times |SRTT - RTT| \tag{3.7}$$

$$SRTT = \frac{7}{8} \times SRTT + \frac{1}{8} \times RTT \tag{3.8}$$

$$RTO = SRTT + max(G, K \times RTTVAR) \tag{3.9}$$

RFC6298 [31] states that TCP must use Kern's algorithm [24] for taking RTT samples. This means that we can't take samples from retransmitted and reordered segments. However it also says that this is not true if we are using the TCP timestamp option [7], since it removes the uncertainty of which segment that triggered the acknowledgement.

RFC6298 [31] also states that if the timer runs out and we need to retransmit a segment, we must also do a exponential timer back-off. A maximum value of 60 seconds may be used to provide a upper bound to the doubling operation: $RTO = (RTO \times 2 < 60sec)RTO \times 2 : 60sec$.

Delayed ACK Timer

Delayed ACKs makes it possible for the TCP receiver to bundle together ACKs back to the TCP sender to fill up the segments to MSS with ACKs. The TCP receiver can also piggyback ACKs if data is also flowing from TCP receiver to TCP sender. Delayed ACKs will reduce the amount of segments sent from the TCP receiver to the TCP sender, but it will also introduce additional delay. Therefore the TCP sender needs to have a delayed ACK timer which adds additional duration to the retransmission timer, the delay must not exceed 500ms as specified in RFC1122 [8].

Persist Timer

This timer is set when the remote node is advertising a window size of zero. If the timer times out, the local TCP needs to probe the remote node to check if the window size is still zero; if it is, the persist timer is restarted.

Keep-alive Timer

With the keep-alive timer, the TCP sender can set a threshold to the amount of time with inactivity it would allow before it determines that the connection has expired. If set set by the user that the connection is going to remain open, a special segment is sent when the timer runs out to keep it open.

FIN WAIT 2 Timer

This timer is set when TCP is in the FIN-WAIT-2 state where both endpoints is waiting for a FIN segment. If a FIN segments is not received before the timer runs out, the connection is dropped.

TIME WAIT Timer

This timer is set when TCP enters the TIME-WAIT state, and it will expire after twice the Maximum Segment Lifetime (MSL). When the timer runs out, the connection is released and state variables deleted. The reason for this timer is to occupy the connection pair so that no late segments can be received if the connection pair is reused after the connection is released.

3.1.9 Forward Retransmission TimeOut (F-RTO)

F-RTO as specified in RFC5682 [35], is an algorithm for detecting spurious retransmission timeouts. The F-RTO algorithm only affect TCP sender during RTO. It uses timestamps and/or D-SACKs to detect spurious retransmissions.

When the retrasmission timer expires it first retransmits the unacknowledged segment followed by a new unsent segments if there is more data to send. It then monitors the next incoming ACK to detect if the retransmission was spurious.

If so, RFC5682 [35] doen't specify actions to be taken in this situation, but there is a dicussion in which a TCP sender should not countinue retransmitting segment based on the timeout which is proved spurious. It should revert back to the previous phase before the timeout.

If not, normal RTO should be applid. This sets the `cwnd` to `3 * MSS` and countiue with slow-start.

3.1.10 Different Flavours Of TCP Congestion Control

There is a lot of different flavours of TCP congestion control out there. Some of which are loss based, in the meaning that they relay on loss detection for estimating the congestion in the network. TCP Reno [38], TCP NewReno [20] and TCP CUBIC [18] are all loss based congestion control mechanism which we are going to cover later in this section. There are also other congestion control such as TCP Westwood [10] which relay end-to-end bandwidth estimation when setting *cwnd* and *ssthresh* after congestion is detected, and TCP Vegas [9] which is a delayed based congestion control

mechanism. The latter mechanisms and all the other congestion control mechanisms which is not already mentioned will not be covered.

TCP Reno

TCP Reno is the first congestion control mechanism which incorporates all the algorithms discussed in section 3.1.5 and first specified in RFC2001 [38]. It derives its name from the operating system that the algorithm first appeared, namely 4.3BSD Reno release.

TCP NewReno

TCP NewReno as specified in RFC6582 [20] is an improvement to TCP Reno's fast recovery algorithm and it derives its name from TCP Reno because of this. The improvement resides in the fact that TCP Reno in the absence of the SACK option performs poorly when multiple segments are lost within the same flight window, and will often result in a time-out. However TCP NewReno's improvement is its ability to respond to what we call a partial acknowledgement. In cases where a segment or multiple segments has been lost within the same flight window, the first information available to the TCP sender is the ACK received from the segment that has been retransmitted during fast retransmit. If there is only one segment that has been lost and there isn't any reordering, this ACK will acknowledge all segments transmitted before fast retransmit was entered. However, if there are multiple segments lost this ACK will only acknowledge the segments up to the next hole in the sequence space of the current flight window. The latter is called a partial acknowledgement.

If TCP NewReno receives a partial acknowledgement during fast recovery, the first unacknowledged segment is retransmitted and the *cwnd* is deflated with the amount of new data acknowledged by the cumulative ACK field. If the partial ACK acknowledges at least one *SMSS* of new data, it will artificially inflate the *cwnd* with *SMSS* to reflect the additional segment that has left the network. To attempt to end fast recovery with *ssthresh* amount of data in flight, a segment is sent if permitted by the new value of *cwnd* to partly deflate the window. The retransmission timer is also restarted when the first partial ACK arrives.

TCP CUBIC

TCP CUBIC was first presented by Ha, Rhee, and Xu in [18] and is a congestion control mechanism which aims to improve the TCP-friendliness and RTT-fairness. It does this by changing the window growth function to a cubic function of time since the last congestion event, which makes it RTT independent and therefore allows more fairness between flows.

- **TCP-friendliness:** as defined by Floyd and Fall in [14], a flow is TCP-friendly if its arrival rate does not exceed the arrival of a conferment TCP connection in the same circumstances.

- **RTT-fairness:** as defined by Miras, Bateman, and Bhatti in [29], evaluates the fairness of TCP flows which shares the same bottleneck but traverse network paths with different RTT.

3.2 Linux TCP

The part of the Linux kernel which implements the TCP/IP stack is called the TCP engine. The main part of the TCP engine code is located in various files listed below with a brief description their content. In we will talk about how packets are stored in the TCP engine and in how the TCP output engine work.

- **tcp.c**
  This file contains code for the TCP connection procedures and an entry point for data coming from user space.

- **tcp_output.c**
  This file contains code for handling outgoing packets.

- **tcp_input.c**
  This file contains code for handling incoming packets, and to handle events triggered by ACKs.

- **tcp_cong.c**
  This file contains a pluggable TCP congestion control, allowing it to switch between congestion control algorithms. It also contains the code for TCP NewReno congestion control.

- **tcp_<*>.c**
  <*> is replaced with: bic, cubic, highspeed, htcp, hybla, illinois, scalable, vegas, veno, westwood or yeah. These files contains alternative congestion control algorithms.

3.2.1 The Socket Buffer (SKB)

The socket buffer (SKB) defined as `struct sk_buff` is the most fundamental data structure in Linux networking code. All packets sent or received is handled using this data structure.

SKB contains a `next`, `prev` and `list` pointer as well as all the necessary information to build the packets.

`next` and `prev` pointer is used to implement the list handling. `list` points to the head of the list the SKB is in, which is of type SKB head structure.

SKB head defined as `struct sk_buff_head` is not contained any information to build a packet, since its purpose is to contain only list information. `next` and `prev` is members of SKB head like in SKB, but it also contains a `qlen` variable which is the amount of SKBs in the list, and a `lock` variable which is used to ensure mutual exclusion.

3.2.2 TCP Output Engine

The TCP output engine is responsible to store outgoing data for each TCP socket in SKBs which forms a doubly linked list called TCP output queue.

As we can see in figure 3.5 TCP socket has two members used in TCP output engine: `sk_write_queue` and `sk_send_head`. All pending outgoing SKBs is stored in `sk_write_queue` which is of type SKB header. `sk_send_head` points to the next SKB in the `sk_write_queue` not sent. When an ACK arrives from the receiver and more send window space becomes available, we walk `sk_write_queue` from `sk_send_head` towards `sk_write_queue` head. When `sk_send_head` points to `NULL`, it means that all SKBs in `sk_write_queue` has been sent once. Note that retransmitting SKBs isn't handled here.

In figure 3.6 we can see the call graph of TCP output engine where all paths lead to `tcp_transmit_skb()`, handing over the SKB to the network layer. `tcp_sendmsg` and `tcp_sendpage()` gather up data from user space or the page cache into SKB packets which is then stacked onto `sk_write_queue`. They then invoke either `tcp_write_xmit()` or `tcp_push_one()` to try to output the data.

When an ACK arrives, the TCP input engine calls `tcp_data_send_check()` to see if we have any data left in `sk_write_queue`. If so, `tcp_write_xmit()` is invoked to try to output the data.

When TCP is going to retransmit data in response to either RTO or fast retransmit, `tcp_retransmit_skb` is invoked which passes the SKB to `tcp_transmit_skb`.

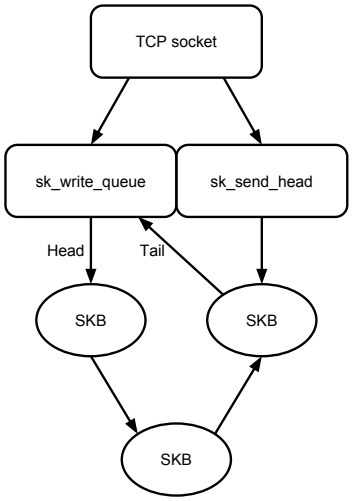


Figure 3.5: TCP output queue

When the receivers sends back an ACK packet covering the sequence space of one or more SKBs in `sk_write_queue`, this allows TCP to unlink and free the SKBs in the `sk_write_queue` using `kfree_skb()`.

3.2.3 TCP Input Engine

The TCP input engine is responsible to store and process inbound data for each TCP socket before it is delivered to user space in the order it was sent out by TCP sender. In figure 3.7 we can see the call graph of TCP input engine where all paths either lead to data delivered to user space, an ACK back to TCP sender or both. All the main function in the TCP input engine is briefly described in the following list:

- `tcp_v4_rcv()` retrieves a TCP socket from the TCP socket hash table with the address and port number from the SKB received from network layer as key. The incoming SKB is discarded if it fails the

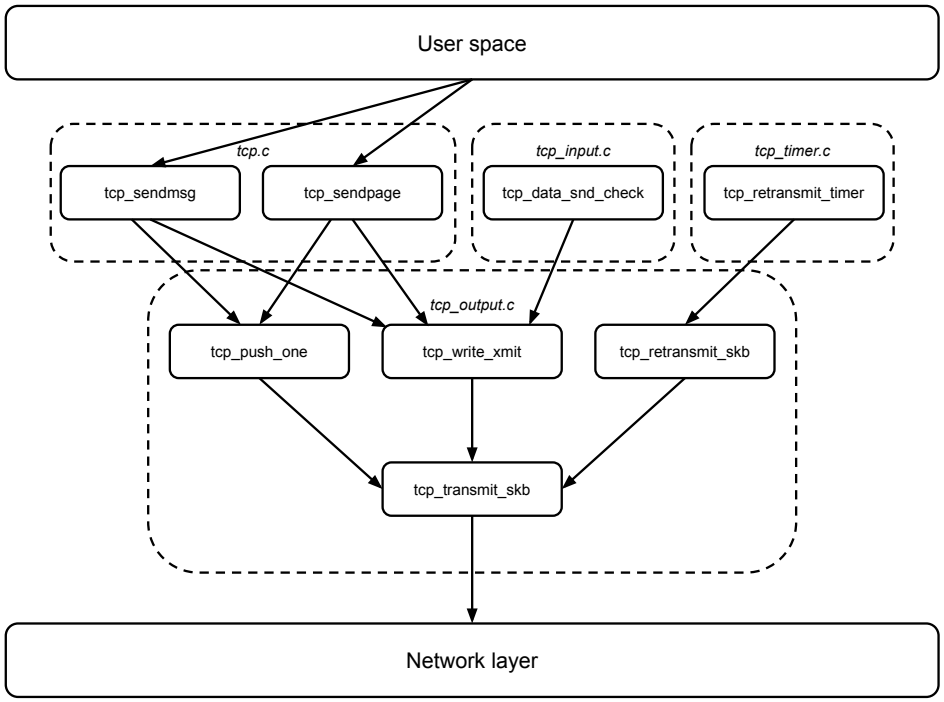


Figure 3.6: TCP output engine

following: retrieve the TCP socket, Time-To-Live (TTL) test, policy filter, socket filter, CRC check in time wait processing or if it fails to add a SKB to the backlog queue. If the TCP socket state is in `TIME_WAIT` time wait processing is performed. If the TCP socket is owned by the user, it first tries to add the SKB to prequeue. If this fails, the SKB is handed over to `tcp_v4_do_rcv()`. If the TCP socket is not owned by the user, it tries to add the SKB to the backlog queue.

- `tcp_v4_do_rcv()` checks the TCP socket state and handing the SKB over to either `tcp_v4_established()` if the TCP socket state is in `ESTABLISHED` or `tcp_rcv_state_process()` for all other states.

- `tcp_rcv_established()` is divided into two processing paths: fast and slow path. Slow path processes the incoming packets as defined in RFC793 [32] whereas fast path is a TCP optimization which skips unnecessary packet handling when deep packet inspection is not needed. By default fast path is disabled, before fast path can be enabled, four criteria listed below must be met:

  1. The out of order queue must be empty.
  2. Receive window cannot be zero.
  3. Memory must be available
  4. An urgent pointer has not been received.

Even after the fast path has been enabled, TCP segments must be verified before they are accepted to be treated in fast path. If a TCP segment does not pass verification, it is processed in the slow path.

To verify the TCP segments, a technique known as header prediction is used. Header prediction compares certain bits in the incoming TCP segment header, to ensure that there is no spacial condition requiring additional processing. The header flags is compared against prediction flags (`pred_flags`), header `seq` against `rcv_nxt`, header `ack_seq` against `snd_nxt`. So to disable fast path, TCP only needs to zero out the `pred_flags` causing header prediction to always fail.

Fast path first tries to deliver the data segments received directly to user space, if this fails the SKB is handed over to `tcp_queue_rcv()` before scheduling an ACK for the received data, processing the received ACK and finally check if the is data to be sent by calling `tcp_data_snd_check()`.

Slow path first validates the incoming segment by calling `tcp_validate_incoming()`, which checks the following: PAWS (defined in RFC1323 [22]), sequence number, RST bit and SYN bit. checking of RST bit and SYN bit applies improvements defined in RFC5961 [34]. If the validation is passed, the received ACK and urgent pointer is processed before handing the SKB over to `tcp_data_queue()`.

If somehow the TCP header has an incorrect size or validation of the TCP segment fails in slow path, the TCP segment is discarded.

Both fast and slow path makes RTT measurements with and without time stamp option and storing of recent time stamps used by PAWS when TCP segments are in sequence.

- `tcp_rcv_state_process()` implements all the receiving procedures defined in RFC793 [32] for all states except `ESTABLISHED` which is implemented in `tcp_rcv_established()` and `TIME_WAIT` which is implemented in `tcp_v4_rcv()`.

- `tcp_queue_rcv()` first tries to merge the incoming SKB with the previous SKB. This will reduce the overall memory use and queue length. If the merge was successful, the function will return back that the SKB was eaten, otherwise the SKB is added to the receive queue.

- `tcp_data_queue()` first checks if the incoming TCP segment has any data to be processed. If not, the SKB is dropped. Then the ECN flag is processed before checking if the TCP segment is in sequence.

  If the TCP segment is in sequence, it first checks if there is more space in the receive window. If not, an immediate ACK is scheduled and the SKB is dropped. Otherwise, it first tries to deliver the TCP segment data to user space. If this fails, it tries to add the SKB to the receive queue after it has called `tcp_try_rmem_schedule()` to check

if more memory has to be allocated to the socket memory pool before queuing the SKB. If this fails, it tries to squeeze out some memory by pruning the receive and out of order queue. If this also fails, the SKB is dropped. Otherwise, if the data segment has been delivered to user space or the SKB has been queued, an ACK is scheduled for the received data. Next the FIN flag is checked, if set the socket state is moved to `TCP_CLOSE_WAIT`, delayed ACK is disabled, all the SKBs in the out of order queue is then dropped and SACK is reset if enabled. Next it checks if the out of order queue is empty, if not `tcp_ofo_queue()` is called, which checks if there is data in the out of order queue which can be added to the receive queue, a DSACK is generated in cases where new segments cover partially of fully any segment in the out of order queue. It then checks if the out of order queue is empty again. If it is empty all the gaps in the queue has been filled and this results in an immediate ACK. Since `RCV.NXT` now advances, some SACKs are now eaten by calling `tcp_sack_remove()`. It then checks if fast path is now possible and setting the prediction flag if so.

If the TCP segment is out of sequence, it first checks if the incoming TCP segment is a retransmission by checking if the end sequence is less then what we expect to receive. In this case, it generates a DSACK by calling `tcp_dsack_set`, scheduling an immediate ACK and dropping the SKB. Otherwise, it checks if the TCP segment is a zero window probe which will result in an immediate ACK before dropping the SKB. It will then enter quick ACK mode, which disables delayed ACK. Next it checks if this is a partial segment which generates a DSACK for the overlapping segment. It also checks if there is no more space in the receive window, resulting in an immediate ACK and dropping the SKB. Otherwise, it will try to add the SKB to the out of order queue after `tcp_try_rmem_schedule()` is called. If `tcp_try_rmem_schedule()` fails, the SKB is dropped. Otherwise, header prediction is disabled and the SKB is merged or added to the out of order queue. If this is the very first segment going into the out of order queue, SACK is also initialized with the first SACK block. If not, it needs to find the proper position in the queue depending on the sequence space of the incoming segment. Then it needs to either expand the existing SACK block or create a new one. It is only possible to expand the existing one if the following conditions are met:

1. The new segment is in sequence with the last segment in the out of order queue.
2. The number of SACKs must be greater than zero.
3. The last segment in the out of order queue must be the latest one to arrive.

If any of the above condition are `FALSE`, a new SACK block is created. If the new segment partly overlaps or is completely covered by some

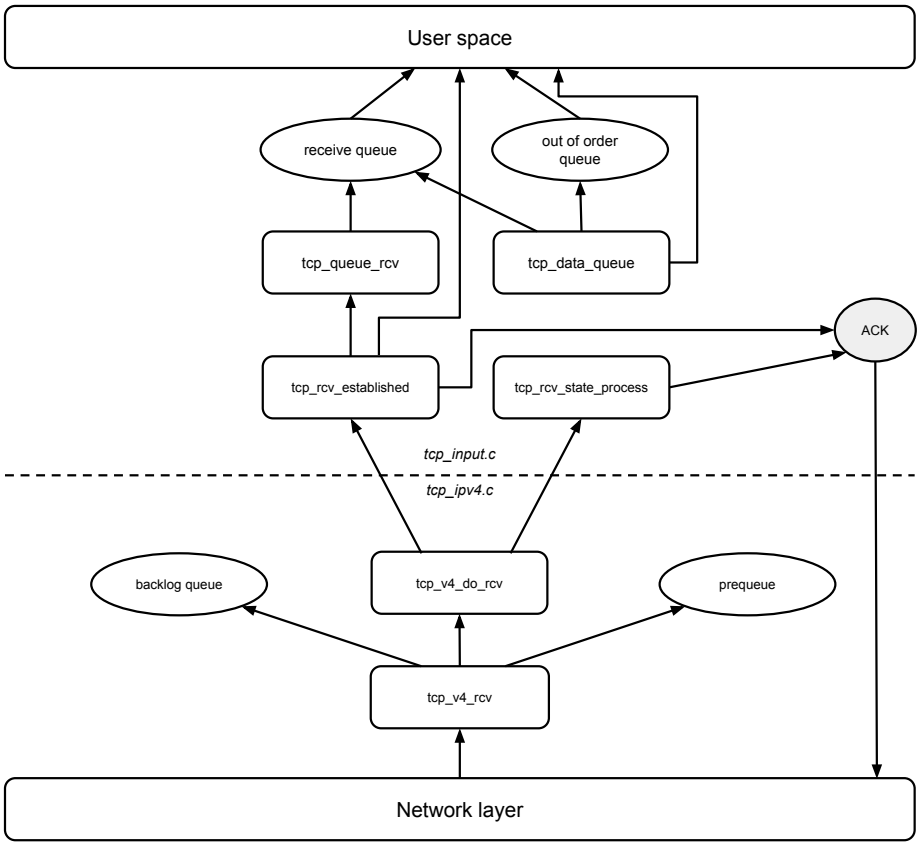


Figure 3.7: TCP input engine

of the segments in the out of order queue, a DSACK is created.

When the user tries to read data from the TCP socket, it calls `tcp_recvmsg`. This will process the queues in the following order:

1. receive queue
2. prequeue will be waited.
3. backlog queue is copied to receive queue when the process is ready to release the socket.

This will ensure that the data is copied to user space in the same order as it was sent out.

Chapter 4

Robustness Against Network Reordering In Linux TCP

The fast retransmit and fast recovery algorithms makes TCP perform poorly when there is a significant amount of reordering in the network. The reason for this is that fast retransmit will mark segments as lost after three successive dupACKs. If we then have segments with a reordering length longer three segments, TCP will misinterpret out-of-order segments as loss. This will result in a false fast retransmit, and repeated false fast retransmit will limit the senders `snd_cwnd` which will result in a severely degraded overall throughput.

In Linux TCP however, there is two mechanisms to address this problem, a proactive and recovery mechanism for false fast retransmit.

In section 4.1 we will look into how Linux TCP protectively prevents false fast retransmit, in section 4.2 we will look into how Linux TCP recovers from false fast retransmit and in section 4.3 we will look into how Linux TCP responded to SACKs during a test where we aggregated two links with different delays.

4.1 Proactively Prevent False Fast Retransmit

Linux TCP is preventing false fast retransmits proactively by introducing three new state variables shown in table 4.1.

To adaptively adjust its duplicate ACK threshold (dupthresh) to fast retransmit when network reordering is detected, Linux TCP has replaced the dupthresh variable which was the static number 3 to fast retransmit with the `reordering` variable.

The `reordering` variable is a heuristic of the maximum reordering length detected. It grows from `tcp_reordering` to `tcp_max_reordering`. Both `tcp_reordering` and `tcp_max_reordering` can be adjusted with the sysctl tool, but their default values is 3 and 300.

To detect network reordering, Linux TCP uses either the SACK-option if permitted or emulate SACKs for SACK-less connections.

When emulating SACKs, Linux TCP can only guess the reordering length based on dupACKs expected compared to dupACKs received. If it

Variable	Description
`reordering`	Maximum reordering length heuristics
`tcp_reordering`	The min value of `reordering`. The default value is 3. It can be changed with `sysctl`
`tcp_max_reordering`	The maximum value that `reordering` can grow to. The default value is 300. It can be changed with `sysctl`
`sacked_out`	The amount of SACKed segments not cumulatively ACKed
`fackets_out`	The amount of segments between forward most SACKed and the highest cumulatively ACKed segment
`retrans_out`	The amount of segments retransmitted and not acknowledged

Table 4.1: Variables used in Linux TCP to make it reordering robust

received more dupACKs than expected, it counts this as reordering. With SACKs, Linux TCP can accurately calculate the reordering length based on the amount of segments between the forward most SACKed segment and the lowest SACKed or ACKed segment.

In section 4.1.1 we will explain how Linux TCP sender detects and calculates the reordering length and in section 4.1.2 we go into some issues related to the increased dupthreah and the FACK algorithm when we have network reordering.

4.1.1 Reordering Length

On SACK-less connections, Linux TCP sender can only estimate the reordering length based on dupACKs expected compared to dupACKs received. If it receives more dupACKs than expected, it counts this as reordering. The reordering length is than updated with the amount of outstanding packets + packets newly acknowledged.

On SACK permitted connections, Linux TCP sender find the reordering length based on the amount of segments between the forward most SACKed and the lowest SACKed or ACKed segment. As shown in figure 4.1, the reordering length is updated both from `tcp_sacktag_write_queue` and `tcp_clean_rtx_queue`.

- `tcp_sacktag_write_queue` is where SACKs is processed. From this function, the reordering length is updated with the amount of segments between the forward most SACKed and the lowest SACKed segment.

- `tcp_clean_rtx_queue` is where the retransmission queue is cleaned after we have received an ACK. From this function, the reordering length is updated with the amount of segments between the forward

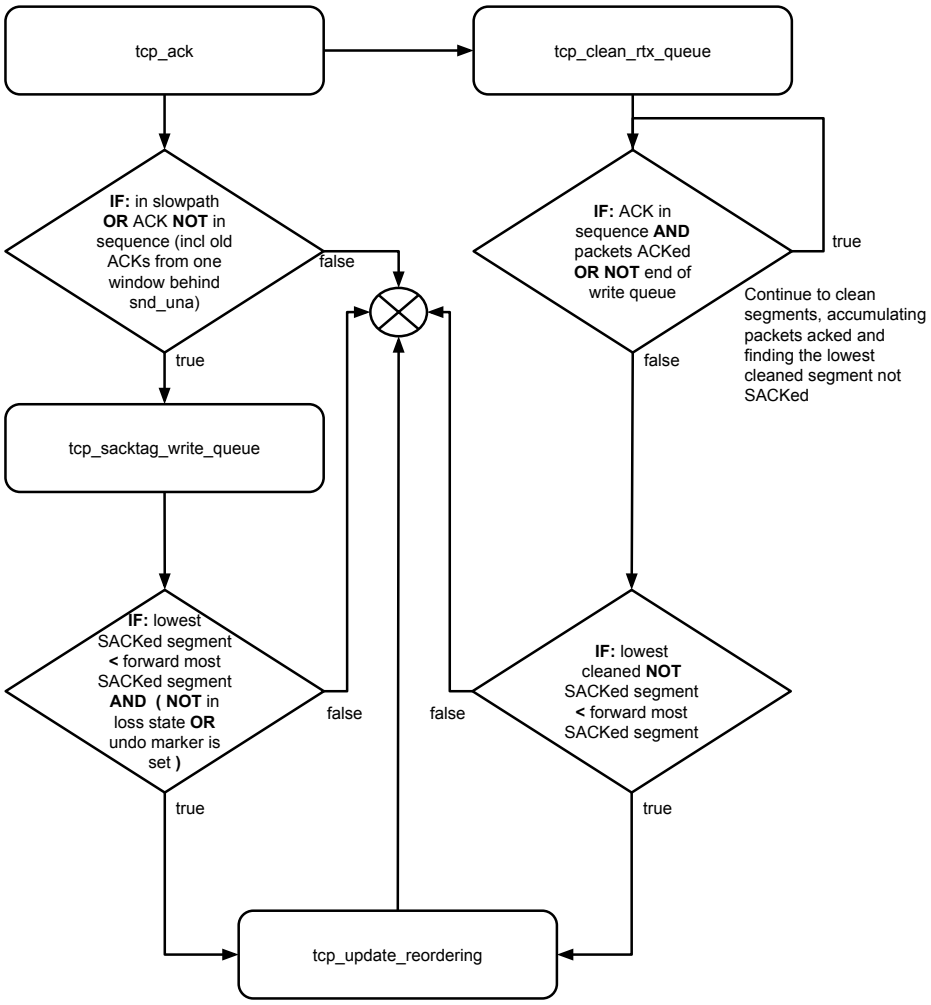


Figure 4.1: Simplified call-graph for updating reordering length in Linux TCP sender for SACK permitted connections.

most SACKed, and the first cleaned segment not SACKed (first previous hole).

4.1.2 Known Issues

When we have an increased dupthresh variable, it makes the TCP sender respond more slowly to packet loss. Thats why Linux TCP resets `reordering` variable back to `tcp_reordering` during a RTO, since the reordering length may have been overestimated at this point.

When it comes to the FACK algorithm, the algorithm only works if there isn't any network reordering. For that reason, Linux TCP disables FACK when reordering is detected: `reordering` > `tcp_reordering`.

4.2 Recover From False Fast Retransmit

To recover from false fast retransmit, Linux TCP uses DSACK as a mechanism to detect spurious retransmission during fast retransmit. If a spurious retransmission is proven, TCP tries to undo the congetion reduction by reverting back to the old `snd_cwnd` logged before entering fast retransmit.

4.3 An Example Of How Linux TCP Updates Its Reordering Length Heuristics

To enable us to see how Linux TCP updates its reordering length heuristics in practice, we needed to look into how the TCP sender processed SACKs received.

To achieve this, we needed to run an experiment on a practical testbed as explained later in chapter 5, where we have introduced network reordering caused by aggregating traffic over two network paths from one TCP connection in a round-robin fashion. The two paths differs in network delay with a magnitude of 10 which will cause packets to naturally be reordered since half of them are delayed with this magnitude.

Table 4.2 shows us which segments are marked ACKed or SACKed and also retransmitted in the TCP senders retransmission queue. Table 4.3 shows us some important state variable related to how TCP sender updates its reordering length heuristics after each ACK segment is received.

As we can see in table 4.2 from ACK segment 1 to 5, we are receiving dupACKs with SACK as expected. But as shown in table 4.3, the reordering length isn't updating from the default value of 3 until we are getting a cumulative ACK in segment 6, which gives us reordering length of 8.

So why isn't the reordering length updated when the TCP sender receives segment 3-5? At segment 3 the reordering length could have been updated to 4 and at segment 4 the reordering length could have been updated to 5.

The reason behind this behavior is that the TCP sender had segments retransmitted and not accounted for while receiving ACK segment: 2, 3, 4, 5 and 6. This caused it to not update its reordering length in `tcp_sacktag_write_queue`. We will walk through the source code related to reordering for `tcp_sacktag_write_queue` in section subsec:tcpsacktagwritequeue

In `tcp_clean_rtx_queue` it didn't update its reordering length for ACK segment 1-5 since it doesn't clean any segments after receiving these ACK segments. After ACK segment 6 however, `tcp_clean_rtx_queue` is cleaning one segment of the retransmission queue causing it to update its reordering length to 8. We will walk through the source code related to reordering for `tcp_clean_rtx_queue` in section 4.3.2.

Seg #	ACK	SACK	Segments											
1	1	3-4	1	2	3	4	5	6	7	8	9	10	11	12
2	1	3-5	1	2	3	4	5	6	7	8	9	10	11	12
3	1	7-8, 3-5	1	2	3	4	5	6	7	8	9	10	11	12
4	1	7-9, 3-5	1	2	3	4	5	6	7	8	9	10	11	12
5	1	7-9, 3-5	1	2	3	4	5	6	7	8	9	10	11	12
6	2	7-9, 3-5	1	2	3	4	5	6	7	8	9	10	11	12
7	5	7-9	1	2	3	4	5	6	7	8	9	10	11	12
8	6	7-9	1	2	3	4	5	6	7	8	9	10	11	12
9	9		1	2	3	4	5	6	7	8	9	10	11	12
10	9	11-12	1	2	3	4	5	6	7	8	9	10	11	12
11	9	11-13	1	2	3	4	5	6	7	8	9	10	11	12
12	9	2-3, 11-13	1	2	3	4	5	6	7	8	9	10	11	12
13	10	11-13	1	2	3	4	5	6	7	8	9	10	11	12
14	13		1	2	3	4	5	6	7	8	9	10	11	12

Table 4.2: Marking of retransmission queue in Linux TCP sender, where green is segments cumulatively ACKed, gray is segments SACKed and red is segments marked retransmitted.

Seg #	`sacked_out`	`fackets_out`	`retrans_out`	`reordering`
1	1	3	0	3
2	2	4	1	3
3	3	7	2	3
4	4	8	2	3
5	4	8	2	3
6	4	7	1	8
7	2	4	0	8
8	2	3	0	8
9	0	0	0	8
10	1	3	0	8
11	2	4	0	8
12	2	4	0	8
13	2	3	0	8
14	0	0	0	8

Table 4.3: State variables in Linux TCP sender

4.3.1 tcp_sacktag_write_queue

Looking at the source code for tcp_sacktag_write_queue in listing 4.1, we can see that the state.reord variable is initially set to packets out at line 4.

The state.reord variable represents the lowest newly SACKed segment not retransmitted or newly DSACKed segment previously SACKed if ever retransmitted.

Looking at line 6, we can see that state.reord needs to be less than the packet count for the forward most SACKed segment and we cannot be in loss state or the undo maker has to be set before the tcp_update_reordering is called at line 8. As input for the reordering length metric to tcp_update_reordering, we can see that the packet count for the forward most SACKed segment minus the packet count for the lowest SACKed segment (tp->fackets_out - state.reord) is used.

Listing 4.1: Linux kernel v4.0 source code

```
1  static int tcp_sacktag_write_queue(struct sock *sk, const
       struct sk_buff *ack_skb, u32 prior_snd_una, long *
     sack_rtt_us)
2  {
3  ....
4  state.reord = tp->packets_out;
5  ....
6  if ((state.reord < tp->fackets_out) && ((inet_csk(sk)->
     icsk_ca_state != TCP_CA_Loss) || tp->undo_marker))
7                  tcp_update_reordering(sk, tp->fackets_out
                       - state.reord, 0);
8  ....
9  return state.flag;
10 }
```

To find the packet count for the lowest newly SACKed or DSACKed segment, we need to look at the source code for tcp_sacktag_one. As shown in figure 4.2, tcp_sacktag_one is called from different helper functions used by tcp_sacktag_write_queue.

As shown in listing 4.2 at line 8 and 19, we can see that state->reord is set to min(fack_count, state->reord) where fack_count is the packet count where the SACKed or DSACKed segment is in the retransmission queue.

As we can see from line 5-8, line 8 is only executed when we have received a DSACK for a segment previously been marked retransmitted and also marked SACKed.

If we then look at line 10-19, line 19 is only executed when the segment is not previously been SACKed and not ever retransmitted. The start sequence must also be before the forward most SACKed segment's start sequence.

Listing 4.2: Linux kernel v4.0 source code

```
1  static u8 tcp_sacktag_one(struct sock *sk,struct
       tcp_sacktag_state *state, u8 sacked, u32 start_seq,
       u32 end_seq, int dup_sack, int pcount, const struct
       skb_mstamp *xmit_time)
2  {
3  ....
4  /* Account D-SACK for retransmitted packet. */
5  if (dup_sack && (sacked & TCPCB_RETRANS)) {
6  ....
7          if (sacked & TCPCB_SACKED_ACKED)
8                  state->reord = min(fack_count, state->
                       reord);
9  }
10 if (!(sacked & TCPCB_SACKED_ACKED)) {
11         if (sacked & TCPCB_SACKED_RETRANS) {
12         ....
13         } else {
14                 if (!(sacked & TCPCB_RETRANS)) {
15                 /* New sack for not retransmitted frame,
16                 * which was in hole. It is reordering.
17                 */
18                 if (before(start_seq,tcp_highest_sack_seq
                       (tp)))
19                         state->reord =min(fack_count,
                               state->reord);
20                         ....
21                 }
22         ....
23         }
24 ....
25 }
26 ....
27 return sacked;
28 }
```

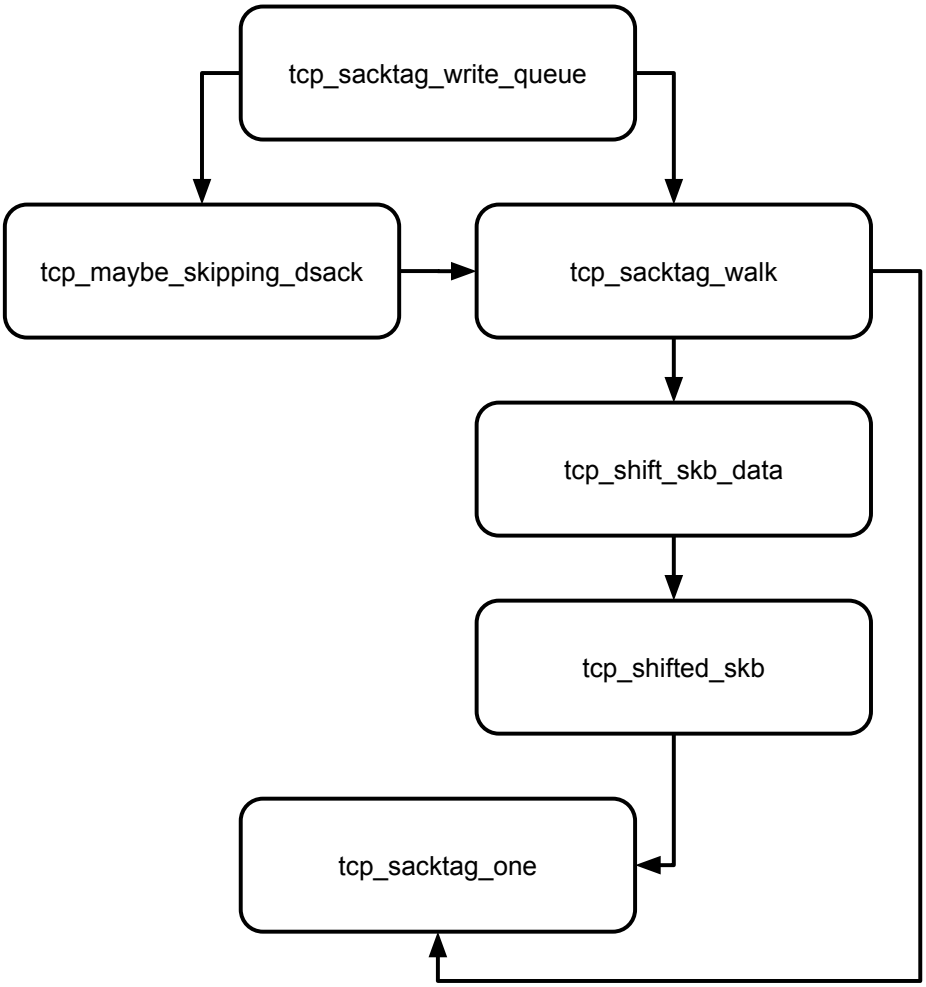


Figure 4.2: Call-graph to `tcp_sacktag_one`

4.3.2 tcp_clean_rtx_queue

Looking at the source code for tcp_clean_rtx_queue in listing 4.3, we can see that the reord variable is initially set to packets out at line 5 and packets ACKed is also set to zero at line 7 since we haven't cleaned any segments of the retransmission queue yet. We than enter the while loop at line 9. The while loop is iterating through the retransmission queue and will break at line 17 or 18 if we have no more segments to be cleaned or it will break at line 38 if we partially cleaned a segment.

If the segment in the retransmission queue has not been marked retransmitted and also not marked SACKed, the reord variable is set to min(pkts_acked, reord) at line 32. pkts_acked is then incremented at line 37 with the packet count which was cleaned.

After we have exited the while loop, tcp_update_reordering is called at line 48 if reord is less than the packet count for the forward most SACKed segment prior to entering tcp_clean_rtx_queue and the reordering metric passed as an argument to this function in set to the packet count for the forward most SACKed segment minus reord.

When segment 6 was received in the previous experiment we talked about in section 4.3, the reord variable was set to zero in the first iteration of the while loop since pkts_acked was zero and incremented later at line 37. This resulted in a reordering length of 8 since tp->fackets_out is decremented later at line 51. Also the retransmission flag was removed prior to entering tcp_clean_rtx_queue from the function tcp_mark_lost_retrans which is called from tcp_sacktag_write_queue which was called since segment 6 included some SACKs.

Listing 4.3: Linux kernel v4.0 source code

```
1  static int tcp_clean_rtx_queue(struct sock *sk, int
       prior_fackets, u32 prior_snd_una, long sack_rtt_us)
2  {
3  ....
4  u32 prior_sacked = tp->sacked_out;
5  u32 reord = tp->packets_out;
6  bool fully_acked = true;
7  ....
8  u32 pkts_acked = 0;
9  ....
10 while ((skb = tcp_write_queue_head(sk)) && skb !=
       tcp_send_head(sk)) {
11         ....
12         u32 acked_pcount;
13         tcp_ack_tstamp(sk, skb, prior_snd_una);
14
15         /* Determine how many packets and what bytes were
               acked, tso and else */
16         if (after(scb->end_seq, tp->snd_una)) {
17                 if (tcp_skb_pcount(skb) == 1 || !after(tp
                       ->snd_una, scb->seq)) break;
18                 acked_pcount = tcp_tso_acked(sk, skb);
```

```
19                  if (!acked_pcount) break;
20                  fully_acked = false;
21          } else {
22                  /* Speedup tcp_unlink_write_queue() and
                       next loop */
23                  prefetchw(skb->next);
24                  acked_pcount = tcp_skb_pcount(skb);
25          }
26
27          if (unlikely(sacked & TCPCB_RETRANS)) {
28                  ....
29          } else {
30                  ....
31                  if (!(sacked & TCPCB_SACKED_ACKED)) {
32                          reord = min(pkts_acked, reord);
33                          ....
34                  }
35          }
36          ....
37          pkts_acked += acked_pcount;
38          ....
39          if (!fully_acked) break;
40          ....
41  }
42  ....
43  if (tcp_is_reno(tp)) {
44          tcp_remove_reno_sacks(sk, pkts_acked);
45  } else {
46          ....
47          /* Non-retransmitted hole got filled? That's
               reordering */
48          if (reord < prior_fackets) tcp_update_reordering(
               sk, tp->fackets_out - reord, 0);
49          ....
50  }
51  tp->fackets_out -= min(pkts_acked, tp->fackets_out);
52  ....
53  return flag;
54  }
```

Part II

Testbed Design and Set-up

Chapter 5

Network Emulation Testbed

To reconstruct the network emulation testbed used by Kaspar in [25], we needed to set up a testbed as described in section 5.1. To enable the host acting as receiver to communicate over two physical interfaces it had to be reconfigured as described in section 5.2. To add the right characteristics to each link so that one link act as a wireless LAN and the other as HSPA (3G) we had to shape the traffic and add some additional delay to each link as described in section 5.3. To aggregate the traffic over both links we had to diverge some of the packets in the TCP flow at the sender and merge back all the diverged packets back to the original TCP flow at the receiver, how this is done is described in section 5.4. In section 5.5 we will describe which tools we use in our experiments and in section 5.6 we are describing which metrics we use to measure throughput, aggregated throughput and reordering.

5.1 Testbed Set-up

The testbed as shown in figure 5.1 corresponds of a dedicated sender and receiver. They both run Linux version 4.0. The receiver is connected to a bridge (bridge0) using two 100 Mbit/s Ethernet links. The bridge0 is again connected to a router using a 1000 Mbit/s Ethernet link. The sender is connected to a bridge (bridge1) using a 1000 Mbit/s Ethernet link, bridge1 is then connected to the same router as bridge0 using a 1000 Mbit/s Ethernet link.

The idea behind this emulated testbed is to easily configure the paths properties such as throughput and RTT. This is difficult in a practical network since its characteristics fluctuates at each moment. The emulated testbed also makes simulations deterministic, which is not the case when running simulations in a practical network.

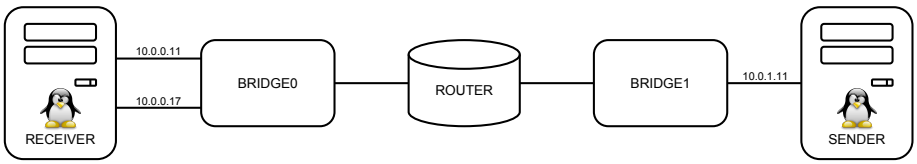


Figure 5.1: Configuration of our network testbed

5.2 Configuring Receiver To Communicate Over Two Interfaces

To enable the receiver to communicate over two interfaces with different addresses, we must tell the kernel to use a specific interface when a source address is used. To achieve this, we must first configure one routing table for each interface identified with a routing table number. We then need to add some new rules in the routing policy table telling the kernel to use a spesicic routing table when a source address is used.

As default the kernel can respond to Address Resolution Protocol (ARP) requests with addresses from other interfaces. The reason for doing this is to increase the chance of successful communication, and IP addresses are owned by the host on Linux and not by the particular interface. In our case this causes some problems and it needs to be configured. To make ARP requests for each interface to be answered based on the source address for that interface we only need to configure the kernel parameter arp_filter to 1, the default value is 0.

Our bash script for configuring the receiver to communicate over two interfaces is shown in listing 5.1.

Listing 5.1: Bash script for configuring receiver

```
1  #!/bin/bash
2  # Adding routing tables
3  ip route add 10.0.0.0/24 dev eth1 src 10.0.0.11 table 1
4  ip route add 10.0.0.0/24 dev eth2 src 10.0.0.17 table 2
5  # Adding default gateways
6  ip route add default via 10.0.0.1 dev eth1 table 1
7  ip route add default via 10.0.0.1 dev eth2 table 2
8  # Adding rules in policy list
9  ip rule add from 10.0.0.11 table 1
10 ip rule add from 10.0.0.17 table 2
11 # Correction to ARP flux
12 sysctl -w net.ipv4.conf.all.arp_filter=1
```

5.3 Rate Control and Network Delay

To change the path characteristics so that we can emulate a wireless LAN and a HSPA connection at the receiver, we must use some mechanisms to control the data rate and add network delay to each path.

In section 5.3.1, we will talk about which tool we use to control the data rate to each path, enlighten a problem with respect to RTT if the buffer size is too large (Bufferbloat) and how we should choose a more correct buffer size with respect to the latter problem.

In section 5.3.2 we will talk about which tool we use to add network delay to each path and why it must be located on a different node than where the data rate control is performed.

5.3.1 Rate Control

To control the data rate to each path, we are using the Linux traffic control (tc) facilities at bridge1 to manipulate the Active Queue Managements (AQM) settings. We are using the classful queueing discipline (qdisc) known as the Hierarchy Token Bucket (HTB), which enables us to classify the traffic based on the destination address so that we can distinguish the data rate between each path.

Since the HTB qdisc is timer based, it will temporarily buffer all the incoming packets and drain it with a given amount of packets within a given time period. If the buffer is full, it will drop the incoming packet.

Because the packets are buffered, it will add some additional delay to each packet from the time it was buffered to the time it is drained. The RTT will then vary based on how full the buffer is and the variation of RTT will therefore vary based on the buffer size.

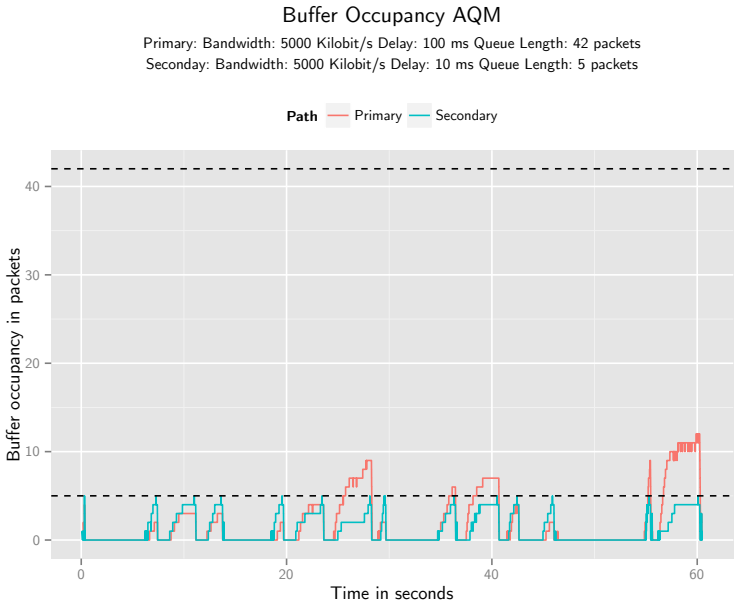
To minimize the RTT variation we must choose a buffer size based on the characteristics of a given link. Traditionally, router buffers have been provisioned according to the Bandwidth Delay Product (BDP) shown in equation 5.1.

However, according to Nichols and Jacobson in [30] this can be problematic since today's links vary in bandwidth and the individual connections vary in delay. Also when we where aggregating the traffic over two links and used BDP for each link to calculate the queue length needed, we experienced loss on the path with the lowest RTT as shown in figure 5.2, resulting in a degraded overall throughput.

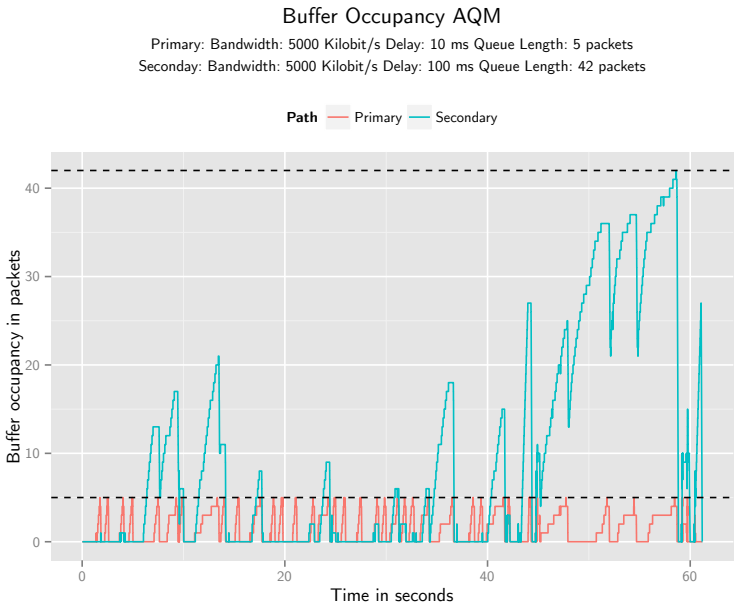
Our hypothesis for this behavior is that the TCP sender looks at the aggregated link as just one link, and therefore we decided to calculate the packet queue length for both paths based on the aggregated path characteristics, using the aggregated bandwidth and the longest RTT in the calculation of BDP. The result of this is shown in figure 5.3, and we can see that we now have a higher utilization of the AQM buffer for both paths. The difference in goodput from old settings to the new ones is shown in figure 5.4, where we can see that the old settings had a much higher variance of goodput causing it to have a lower overall goodput.

To calculate the packet queue length from BDP, we use equation 5.2 (where the Maximum Transmission Unit (MTU) is the largest size of a packet specified in bytes).

Lastly shown in listing 5.2 is our entire bash script for controlling the data rate to each path.

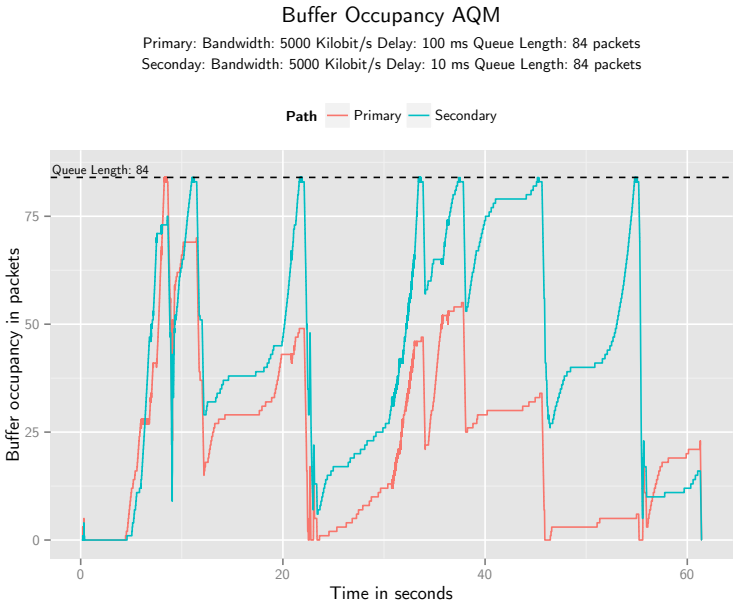


	Packets Received	Packets Dropped
Primary	21282	0
Secondary	21302	16

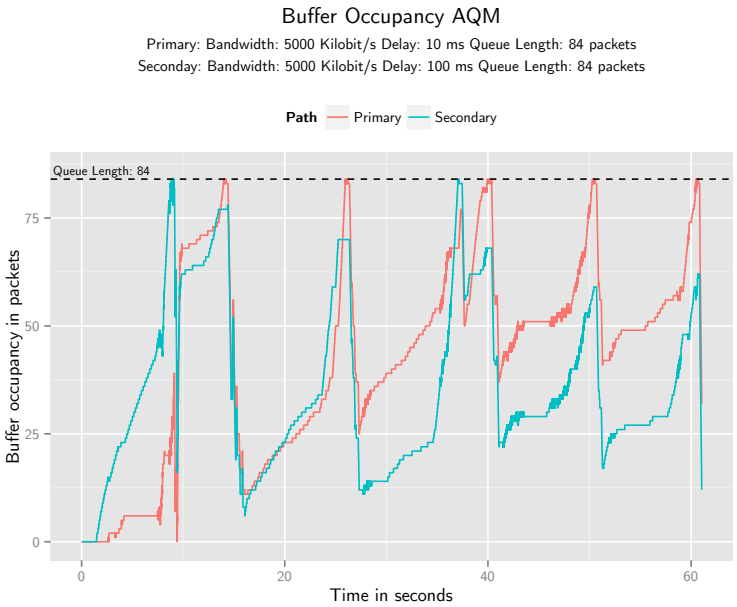


	Packets Received	Packets Dropped
Primary	22518	69
Secondary	22528	1

Figure 5.2: Measurements of buffer occupancy in Active Queue Management with HTB rate control calculating queue length for each path based on their own BDP



	Packets Received	Packets Dropped
Primary	24623	19
Secondary	24743	39



	Packets Received	Packets Dropped
Primary	24836	34
Secondary	24820	9

Figure 5.3: Measurements of buffer occupancy in Active Queue Management with HTB rate control calculating queue length for each path based on BDP from their combined bandwidth and the longest RTT

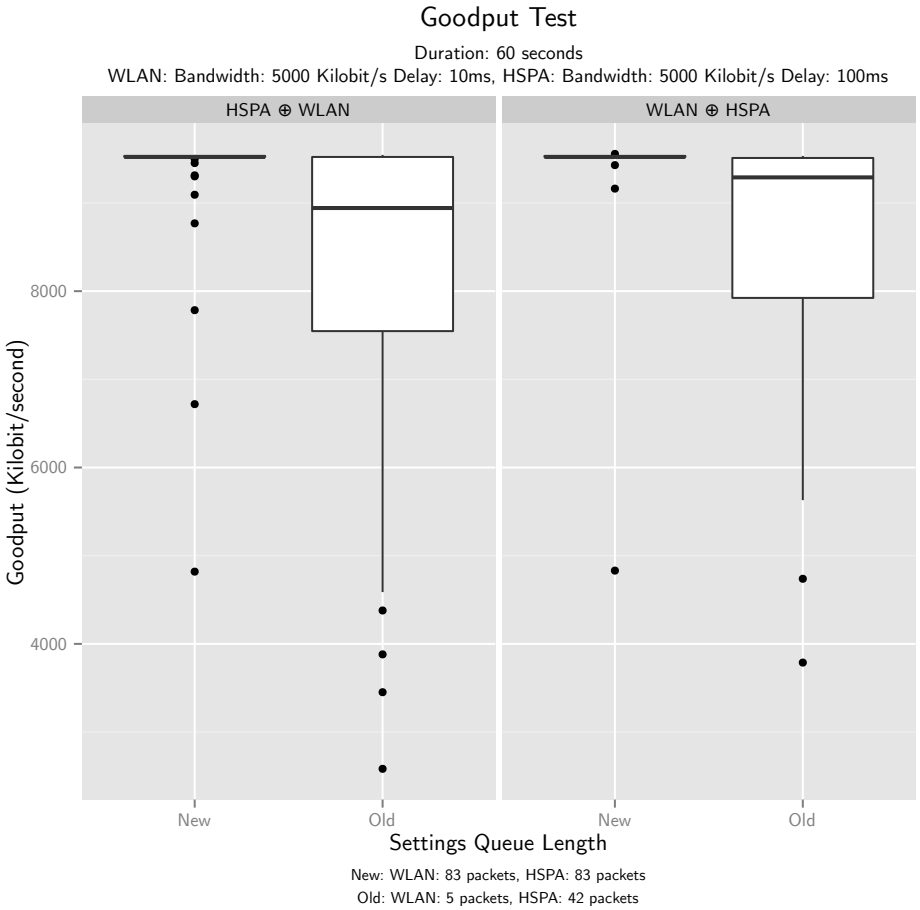


Figure 5.4: Comparison of goodput with different packet queue length settings in AQM over the aggregated path: HSPA $\oplus$ WLAN and WLAN $\oplus$ HSPA

$$BDP = Bandwidth(Bytes) \times RTT(Seconds) \tag{5.1}$$

$$QLEN = BDP/MTU \tag{5.2}$$

5.3.2 Network Delay

To add network delay to each path, we are using a network emulator called netem at bridge0 and the receiver. Netem is an enhancement of the Linux traffic control (tc) which we are using also usning to control the data rate as descibed in section 5.3.1. This allows us to add network delay to outgoing packets from a selected network interface.

As stated by the bufferbloat project in [33] the netem qdisc does not work in conjunction with other qdiscs. A combination of netem and any other complex qdiscs will misbehave, and it will make our data suspicious. An entire separate machine is therefore required for adding network delay in conjunction with rate control. That is why we control the data rate at bridge1 and add network delay at bridge0. At bridge0, we also need to simulate delay in both directions. If the delay is suppose to be 100ms for a path, we need to add 50ms at bridge0 in each direction of the path. Our entire bash script for adding network delay is shown in listing 5.3.

5.4 Diverge and Merge TCP Packets

To make the sender aggregate data as Kaspar did in [25] over both available paths (referred to as primary and secondary path) to make use of all the available bandwidth on both of them, the sender must diverge some of the packets belonging to the TCP connection using the primary path to the secondary path. The receiver must then merge back all packets previously diverged back onto the original TCP flow.

As shown in figure 5.5 this is performed with the use of Destination Network Address Translation (DNAT) at both the sender and receiver. The sender must also include a packet scheduler since we are only diverging a subset of the packets within the TCP connection. To mitigate reverse path reordering (which as shown by Leung, Li, and Yang in [26] causes additional challenges to TCP) and to reduce the need of a packet scheduler at the receiver, we are only translating addresses in the direction towards the receiver. The result of this is that all ACKs only traverse the primary path, which is what we want.

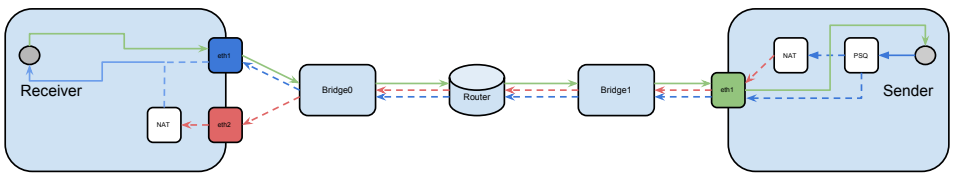


Figure 5.5: Overview of our network emulation Test-bed

Listing 5.2: Bash script for rate control

```
#!/bin/bash
#==============================#
# Primary path characteristics #
#==============================#
pri_limit=5000
pri_delay=100
pri_addr=10.0.0.11
#==============================#
# Secondary path characteristics #
#==============================#
sec_limit=5000
sec_delay=10
sec_addr=10.0.0.17
#========================================#
# Bandwidth Delay Product and queue length #
#========================================#
delay=$pri_delay
if [ $pri_delay -lt $sec_delay ]; then
    delay=$sec_delay
fi
bdp=$(bc <<< "(($pri_limit+$sec_limit)*1000)/8)*($delay*0.001)")
mtu=1500
qlen=$(echo "$bdp␣$mtu" | awk '{printf("%d\n",($1/$2))}')
# FIX rounding UP if we have rest in BDP % MTU
if [ 0 -lt $(echo "$bdp␣$mtu" | awk '{printf("%d\n",($1 %
    $2))}') ]; then
    qlen=$(echo "$qlen" | awk '{printf("%d\n", ($1 + 1))
        }')
fi
#==============================#
#           Rate control       #
#==============================#
tc qdisc add dev eth1 root handle 1: htb
# Adding primary class with rate limit
tc class add dev eth1 parent 1: classid 1:1 htb rate "
    $pri_limit"kbit
# Adding secondary class with rate limit
tc class add dev eth1 parent 1: classid 1:2 htb rate "
    $sec_limit"kbit
# Adding primary queue length
tc qdisc add dev eth1 parent 1:1 pfifo limit $qlen
# Adding secondary queue lengt
tc qdisc add dev eth1 parent 1:2 pfifo limit $qlen
# Attach filter to primary class
tc filter add dev eth1 protocol ip prio 1 u32 match ip
    dst $pri_addr flowid 1:1
# Attach filter to secondary class
tc filter add dev eth1 protocol ip prio 1 u32 match ip
    dst $sec_addr flowid 1:2
```

Listing 5.3: Bash script for adding network delay at bridge0

```
#!/bin/bash
#==============================#
# Primary path characteristics #
#==============================#
# Network interface
pri_dev=eth3
# Network delay in ms
pri_delay=50
#==============================#
# Secondary path characteristics #
#==============================#
# Network interface
sec_dev=eth4
# Delay in ms
sec_delay=5
#==============================#
# Reverse path characteristics #
#==============================#
# Network address
pri_addr=10.0.0.11/32
sec_addr=10.0.0.17/32
#======================#
# Adding network delay #
#======================#
tc qdisc add dev $pri_dev root netem delay "$pri_delay"ms
tc qdisc add dev $sec_dev root netem delay "$sec_delay"ms
tc qdisc add dev eth1 root handle 1: prio
tc qdisc add dev eth1 parent 1:1 handle 10: netem delay "
    $pri_delay"ms
tc qdisc add dev eth1 parent 1:2 handle 20: netem delay "
    $sec_delay"ms
tc filter add dev eth1 protocol ip parent 1:0 prio 3 u32
    match ip src $pri_addr flowid 1:1
tc filter add dev eth1 protocol ip parent 1:0 prio 3 u32
    match ip src $sec_addr flowid 1:2
```

To implement the DNAT with the additional packet scheduler at the sender, we first looked at what was possible to do with iptables in Linux. Iptables is an administration tool for IPv4/IPv6 packet filtering and NAT. We found out that by it self iptables cannot apply rules to translate network addresses only to a subset of the packet stream within a TCP connection, which is what we want. Iptables uses connection tracking only applying rules on a connection basis. This makes sense since what we are doing actually breaking up the connection. We found out that it is two possible approaches to solve this.

The first approach is to use iptables to filter out and queue all the packets belonging to the given TCP connection and send it to user space for further processing using libnetfilter_queue. The libnetfilter_queue is a user space library providing an API to packets that have been queued by the kernel packet filter. In user space we could implement a packet scheduler to only translate network addresses to a subset of the TCP packets queued. The drawback of manipulation packets in user space is that it will add some extra processing time. Unless we are trying to filter large bandwidths, this approach will work just fine.

The second approach is to make a new netfilter module, which is a Linux module where we register a netfilter hook. By doing this we could write a packet scheduler in kernel space. For the module to receive commands from user space to dynamically add new rules we could use IOCTL which enables us to receive commands from user space by writing to a device file.

We decided to go for the second approach. We implemented a kernel module for the sender and a slightly different one for the receiver. A more detailed description is given in section 5.4.1.

5.4.1 Netfilter Modules

To receive packets inside the kernel which we want to further process, we must register a netfilter hook inside our Linux module to express which packets we are interested in. As shown in figure 5.6 these hooks can be either PREROUTING, POSTROUTING, FORWARD, INPUT, or OUTPUT. These hooks are briefly described in the following list.

- In the PREROUTING hook it is allowed to alter (mangle, DNAT) incoming packets before they are routed.
- In the POSTROUTING hook it is allowed to alter (mangle, SNAT) outgoing packets after they are routed.
- In the FORWARD hook it is allowed to alter (mangle) or filter forwarded packets.
- In the INPUT hook it is allowed to alter (mangle) or filter incoming packets addressed to the local host before they are handed over to the local process.

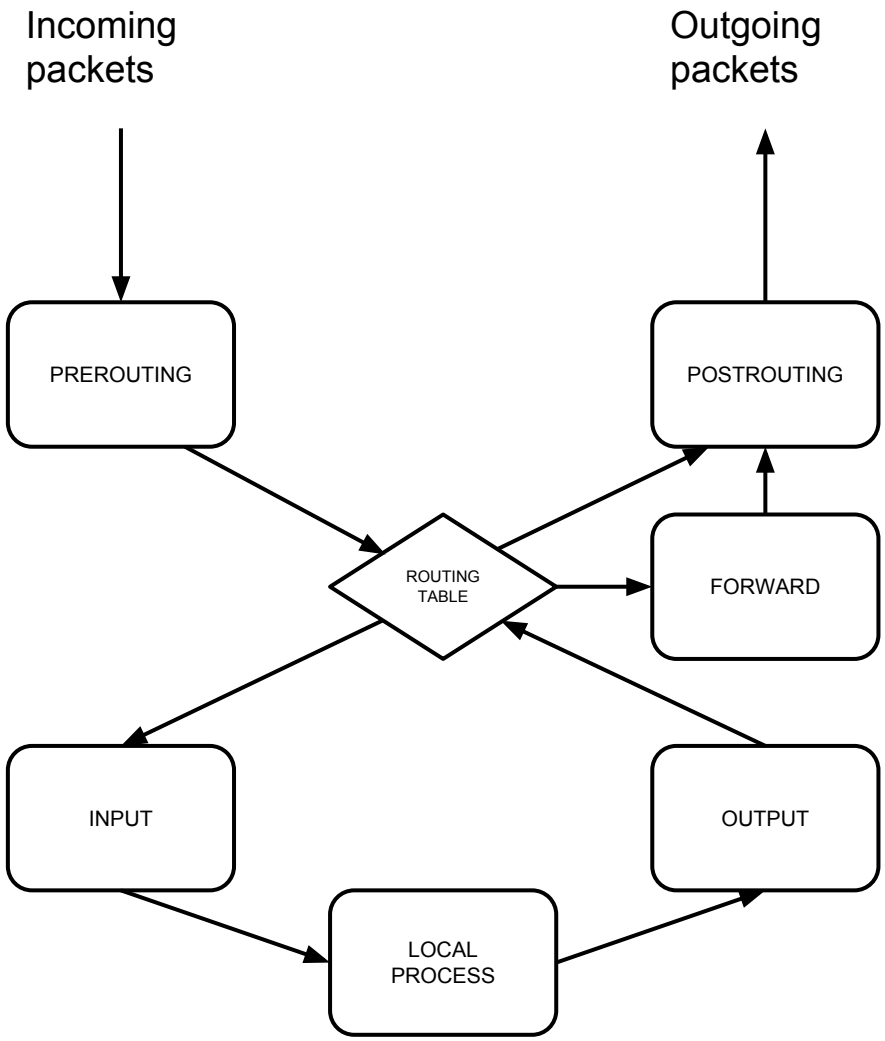


Figure 5.6: Overview netfilter packet flow

- In the OUTPUT hook it is allowed to alter (mangle, DNAT) or filter locally generated packets before they are routed.

Sender Module

The sender module registers to the OUTPUT hook since its going to alter the destination address and maybe the destination port number to some of the locally generated packets. We have implemented a rule chain within the module which contains the destination address and destination port number to the primary and secondary path. We then filter the outgoing packets with the primary destination address and primary port number to select which packets we may have to alter. From this flow of selected packets, the path is selected based on a the bandwidth ratio between

the paths, which is predefined in the rule chain. To do this we have implemented a packet scheduler which uses a send vector like Kaspar did in [25] to select in a round-robin fashion which path the next packet will traverse, altering the destination address and destination port number if needed.

The send vector is represented as a bit field in the kernel module. A look up in the send vector determines which path to take. A 0 bit represents a packet which is to be sent on the primary path and a 1 bit represents a packet which is to be sent on the secondary path. To traverse the send vector, we have a variable which tells us the bit position to look up in the send vector for the next outgoing packet. Since the send vector has a finite bit length, the bit position variable must wrap around using the modular operation.

In order to dynamically load rules from user space into the kernel module, we use an IOCTL call to the module, which sends over a command and the data containing the rule. Every rule needs to include the following data set as shown in table 5.1. As we can see in the data set, the send vector is represented as a 64 bit field and must be generated a priori in user space before it is loaded into the kernel module. Algorithm 1 shows how we generate the binary send vector. However when n is small, like 64 in our experiments, the send vector is often unable to accurately represent the ratio between the two weights. Error of such will translate directly into loss in aggregation efficiency. However, the error is bounded and will never exceed $\frac{1}{2n}$.

Proof. The approximation error, which is the difference between the true ratio: $r_w = w_0/(w_0 + w_1)$ and the send vector's approximated ratio: $r_v = m/n$, where m is the number of zeros and n is the vector length.
The approximation error is defined as:

$$err = |r_w - r_v|$$

In the worst case, m will differ its ideal value $r_w * n$ by $\pm\frac{1}{2}$ (adjusting of r in algorithm 1). We can therefore write m as $r_w * n \pm \frac{1}{2}$, and replace it in the equation:

$$err = \left|r_w - \tfrac{m}{n}\right| = \left|r_w - \tfrac{r_w * n \pm \frac{1}{2}}{n}\right|$$

To find the upper limit, we must set r_w and n such that m differs from $r_w * n$ by $\pm\frac{1}{2}$ making this a worst case scenario. An example would be: $r_w \leftarrow \frac{1}{2}$, $n \leftarrow 3$ and $m \approx (r_w * n)$. The maximum error in the worst case scenario would then be:

$$err = r_w - \frac{m}{n} = \frac{1}{2} - \frac{1}{3} = \frac{3}{6} - \frac{2}{6} = \frac{1}{6} = \frac{1}{2n}$$

The upper limit then simplifies to:

$$err \leq \frac{1}{2n}$$

□

Variable name	Data type	Description
`pri_daddr`	`u32`	Primary destination address
`sec_daddr`	`u32`	Secondary destination address
`pri_dest`	`u16`	Primary destination port
`sec_dest`	`u16`	Secondary destination port
`send_vector`	`u64`	A 64 bit binary send vector

Table 5.1: List of data needed for a valid rule in the sender netfilter module

Algorithm 1: generateSendVector(n, w_0, w_1)

Data: Vector length $n \in \mathbb{N} > 0$
Primary weight $w_0 \in \mathbb{N} \geq 0$
Secondary weight $w_1 \in \mathbb{N} \geq 0$
Result: Send vector V of length n
$V = zero(n)$; `// initialize` V `with` n `zeros`
$r = w_1/(w_0 + w_1)$; `// calculate weight ratio`
$r = round(r * n)/n$; `// adjust r such that` $r * n$ `is an integer`
for $i = 1$ **to** $r * n$ **do**
  $V[i/r] = 1$
end

Receiver Module

The receiver module registers to the PREROUTING hook since its going to alter the destination address and maybe the destination port number to some of the incoming packets.

We are also using a rule chain here, much like the one we are using in the sender module, the only difference is the it lacks the send vector since the receiver module doesn't contain any packet scheduler.

We filter the incoming packets with the secondary destination address and port number to select which packets we must alter.

The rules are also dynamically loaded from user space as we did in the sender module.

There is also two kernel parameters we have to configure to successfully forward packets from one interface to another.

Firstly we have to configure `ip_forward` to allow forwarding between interfaces. In Linux forwarding is by default disabled.

Secondly we have to configure The reverse path filter: `rp_filter`, which chooses the reverse path forwarding mode as defined in RFC3704 [4]. By default the reverse path filter is configured to strict mode. In our experiments the packets arriving at the secondary interface would be discarded in this mode. This is why we need to reconfigure the reverse path filter. Based on our traffic, loose mode would be a correct way to do this. These modes are briefly described in the following list

- Reverse path forwarding strict mode test each incoming packet against the Forwarding Information Base (FIB) to check if the

Listing 5.4: Bash script to allow our receiver to forward packets form the secondary interface to the primary interface

```
#!/bin/bash
#============#
# Interfaces #
#============#
# Primary
pri_dev=eth1
# Secondary
sec_dev=eth2
#====================#
# Enable forwarding #
#====================#
# 0 - Disabled      #
# 1 - Enabled       #
#====================#
sysctl -w net.ipv4.ip_forward=1
#============================================#
# Configure reverse path filter to loose mode #
#============================================#
# 0 - No source validation                   #
# 1 - Strict mode                            #
# 2 - Loose mode                             #
#============================================#
sysctl -w net.ipv4.conf.$pri_dev.rp_filter=2
sysctl -w net.ipv4.conf.$sec_dev.rp_filter=2
sysctl -w net.ipv4.conf.all.rp_filter=2
```

interface which received the packet would be the one used to forward packets to the source address of the packet received. If it fails the packet is discarded.

- Reverse path forwarding loose mode test each incoming packet against the FIB to check if there exist a route via any interface to the source address of the packet received. If it fails the packet is discarded.

Our script to allow the receiver to forward packets form the secondary interface to the primary interface is shown in listing 5.4.

5.5 Tools

For our experiments, we use a variety of tools to measure network performance and to analyse TCP. These are briefly described in the following list.

- **iperf** is a tool for performing network throughput measurements. We use it to generate traffic from the sender to the receiver.

- **tcpdump** is a tool which prints out a description of the contents of packets traversing the network interface its attached to. We use it to capture packets between the sender and receiver to a packet capture (PCAP) file for later analysis.
- **mergecap**, **tcpprep** and **tcprewrite** is used to merge two PCAP files created by **tcpdump** at receiver. We translate the destination address to the diverged packets to mimic merging performed by receiver module since PCAP doesn't pick that up. When all of this is done, the merged PCAP file is ready to be used by tools to retrieve network performance metrics.
- **captcp** is used for TCP analysis of PCAP files.
- **tcp_probe_reordering** is a modification we made to the original **tcp_probe** which is a Linux kernel module for obtaining detailed information about TCPs state. Our modification adds some additional TCP state variables to be tracked.
- **tdn-aqmprobe** is an in-house kernel module that probes qdiscs and extracts drop statistics. We use it extract buffer occupancy and drop statistics in the Active Queue Management at bridge1.
- **R** is used in conjunction with **ggplot2** to create plots.

5.6 Metrics

- **Throughput** is the link utilization or flow rate in bits per second between sender and receiver. Throughput can include packet header size in addition to the payload size for each packet.
- **Goodput** is a subset of throughput consisting of useful traffic. It omits lost or retransmitted packets.
- **Displacement (D)** is a metric defined in RFC5236 [23] to measure the displacement of a packet from where it was expected to be received. A negative values of displacement indicates the earliness of a packet and a positive value of displacement indicates the lateness of a packet. It omits lost or retransmitted packets
- **Reorder density (RD)** is defined in RFC5236 [23] as the distribution of displacement frequencies normalized with respect to the number of packets received. It omits lost or retransmitted packets

Part III

Results and conclusion

Chapter 6

Results

In section section 6.1 we are showing some results which relates to the degree of reordering to give us a better picture of how much reordering we experience at the TCP receiver and how robust Linux TCP sender is against network reordering. In section 6.2 we will show some results related to the performance at different network configurations when aggregating the available paths with our own network layer aggregation or aggregating them with MPTCP.

6.1 Network Reordering

6.1.1 Displacement (D)

Figure 6.1 shows us the displacement and duplicated packets for the first 60 seconds received by the TCP receiver. During the run for the aggregated path WLAN $\oplus$ HSPA, we can see that during slow-start (the beginning of the connection) we experience a high magnitude of displacement. After this, the two aggregated paths behaves more or less the same. We also believe that the spikes we see in the negative directions relates to the high points of senders congestion window. As we can see right after each spike of displacement, we receive duplicate packets.

6.1.2 Reorder Density (RD)

Looking at figure 6.2 which shows us the Reorder Density (RD) over the entire connections for both aggregated paths HSPA $\oplus$ WLAN and WLAN $\oplus$ HSPA, we can see that the 50% of the displacement is between first quantile (Q1) and third quantile (Q3) for each connection. The median for both connection is 0, given that the data rate is the same for both paths. We can also see that WLAN $\oplus$ HSPA has over twice as much maximum displacement in both directions, it gains this during slow-start.

Also most of the packets are actually not on time, only around 2% is. There is also slightly more late than early packets for both connections, but this is such a small number that we could say that the ratio between early and late packets is almost one to one.

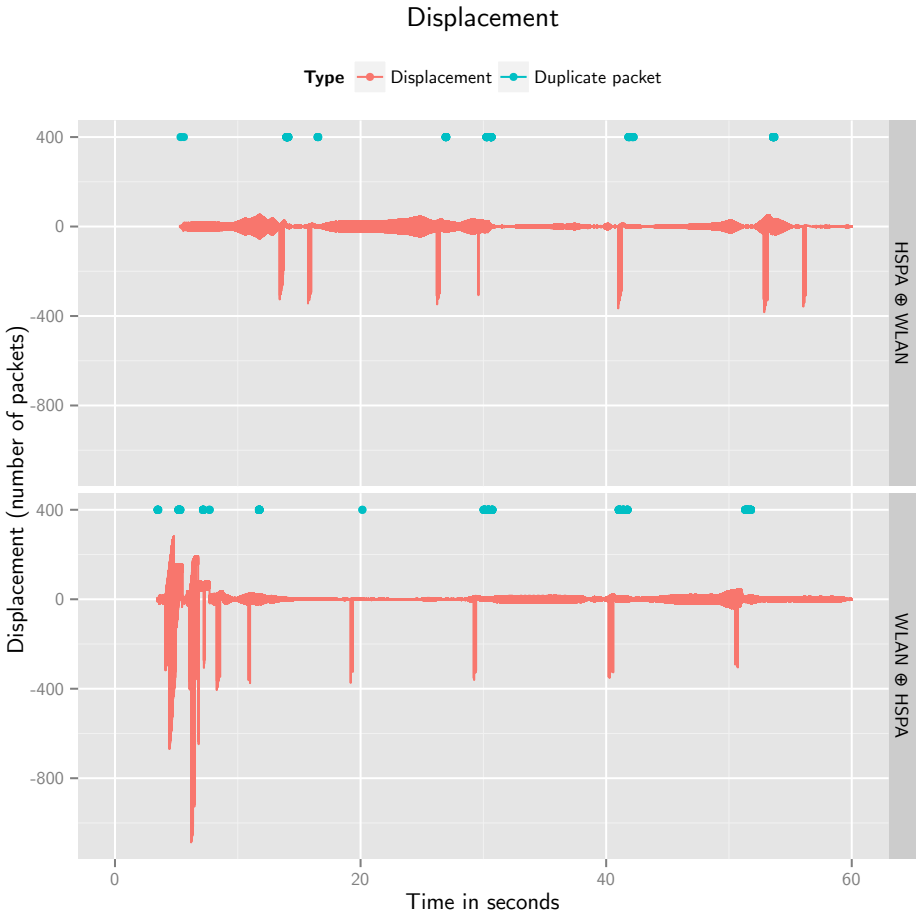
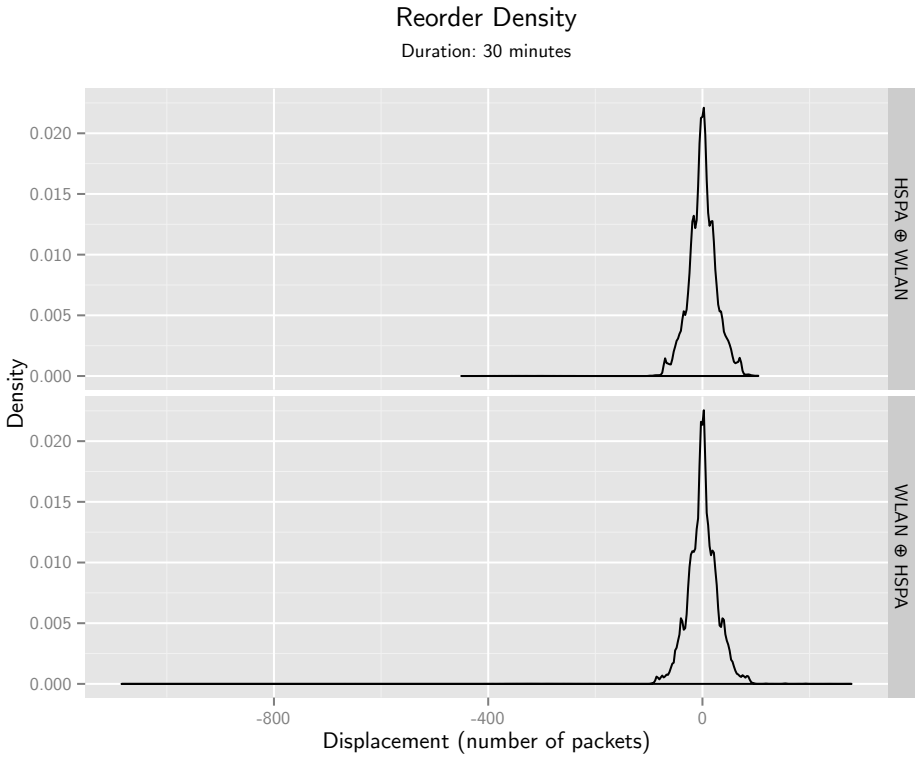


Figure 6.1: Displacement of packets and duplicated packets received at TCP receiver over the network aggregated paths



	pkts rcv	max	min	Q1	Q3	Early(%)	Late(%)	On Time(%)
HSPA ⊕ WLAN	1462314	105	-452	-15	16	48.49	49.54	1.97
WLAN ⊕ HSPA	1466336	281	-1087	-16	17	48.63	49.4	1.96

Figure 6.2: Reorder density of packets received at TCP receiver over the network aggregated paths

6.1.3 Linux TCP's robustness against network reordering

Looking at figure 6.3 and 6.4, we can see how Linux TCP is handling the network reordering in both test cases where we measure the amount of selectively acknowledged packets in the retransmission queue compared to the reordering metric, senders congestion window and which congestion avoidance state we are in.

We can see that we are mostly in the disorder state, and after entering the recovery state when `sacked_out >= reordering`, we can see that `snd_cwnd` is being halved because loss is detected and that is an indication of congestion in the network.

In WLAN $\oplus$ HSPA the RTTM is oscillating between 10 and 55 ms and in HSPA $\oplus$ WLAN its oscillating between 55 and 100 ms. In WLAN $\oplus$ HSPA the initial SRTT after the connection is established is 10 ms where as in HSPA $\oplus$ WLAN its 100 ms because we don't start diverging packets before the connection is established. We can see the effect of this if we compare the first five seconds of the `snd_cwnd` plot in both figures. In figure 6.4 we can see that the `snd_cwnd` is oscillating two times in this time period. If we then look at the congestion avoidance state plot in the same time period, we can see that we enter loss state exactly the same time the `snd_cwnd` is dropped. If we than compare it to what we see in figure 6.3, we can see that we have a much smoother start and we do not enter loss state in any given moment.

In both cases the reordering metric is growing to a stable value much faster than what previously observed by Kaspar in [25], specially for WLAN $\oplus$ HSPA, which was growing very slowly.

6.2 Performance

In terms of performance, we first looked at the effect in relation to throughput and goodput when it comes to the difference in data rate to each network path. This is shown in figure 6.5 and 6.6. Both figures shows that having a higher data rate for the network path with the highest RTT has a small but positive effect. The opposite will have a negative effect in terms of small but periodic loss in throughput and goodput.

We then looked at the impact in relation to the throughput and goodput when the data rate was equal for each network path, and this time we let the test run for 30 minutes. This is shown in figure 6.7 and 6.8. Both figures shows that the network aggregated paths has periodically dropped in throughput and goodput. MPTCP however has a stable and consistent throughput and goodput. Despite this, the network aggregated paths outperforms MPTCP.

We also notice that the overall throughput for the network aggregated paths is slightly higher than the combined data rate for both paths they are aggregated with. This is probably due to the accuracy of the HTB queuing discipline in Linux traffic control. As mention by Benita in [5], its possible to control the throughput accuracy by changing the cburst/burst values.

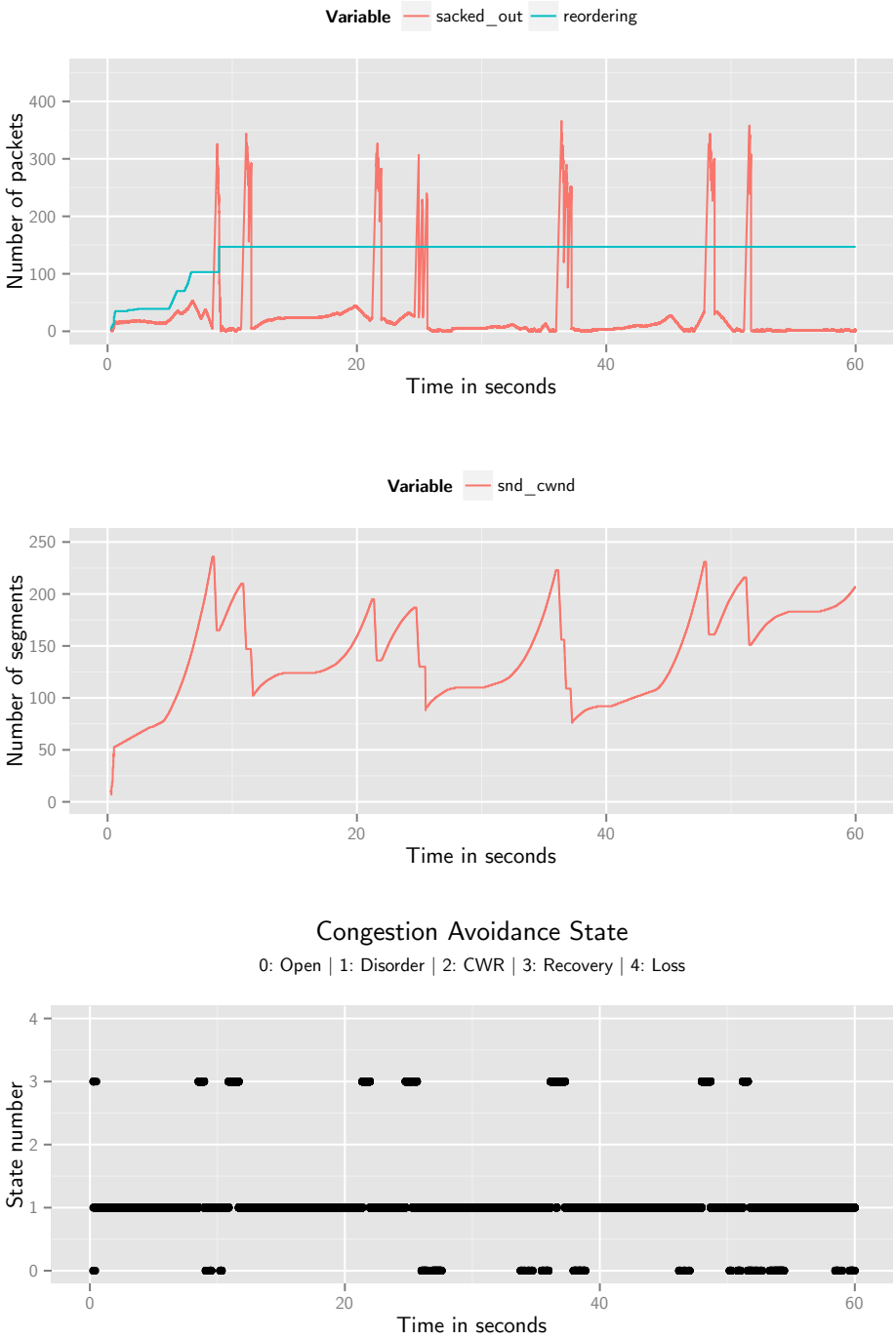


Figure 6.3: Test of Linux TCP's robustness against network reordering over the aggregated network path HSPA $\oplus$ WLAN

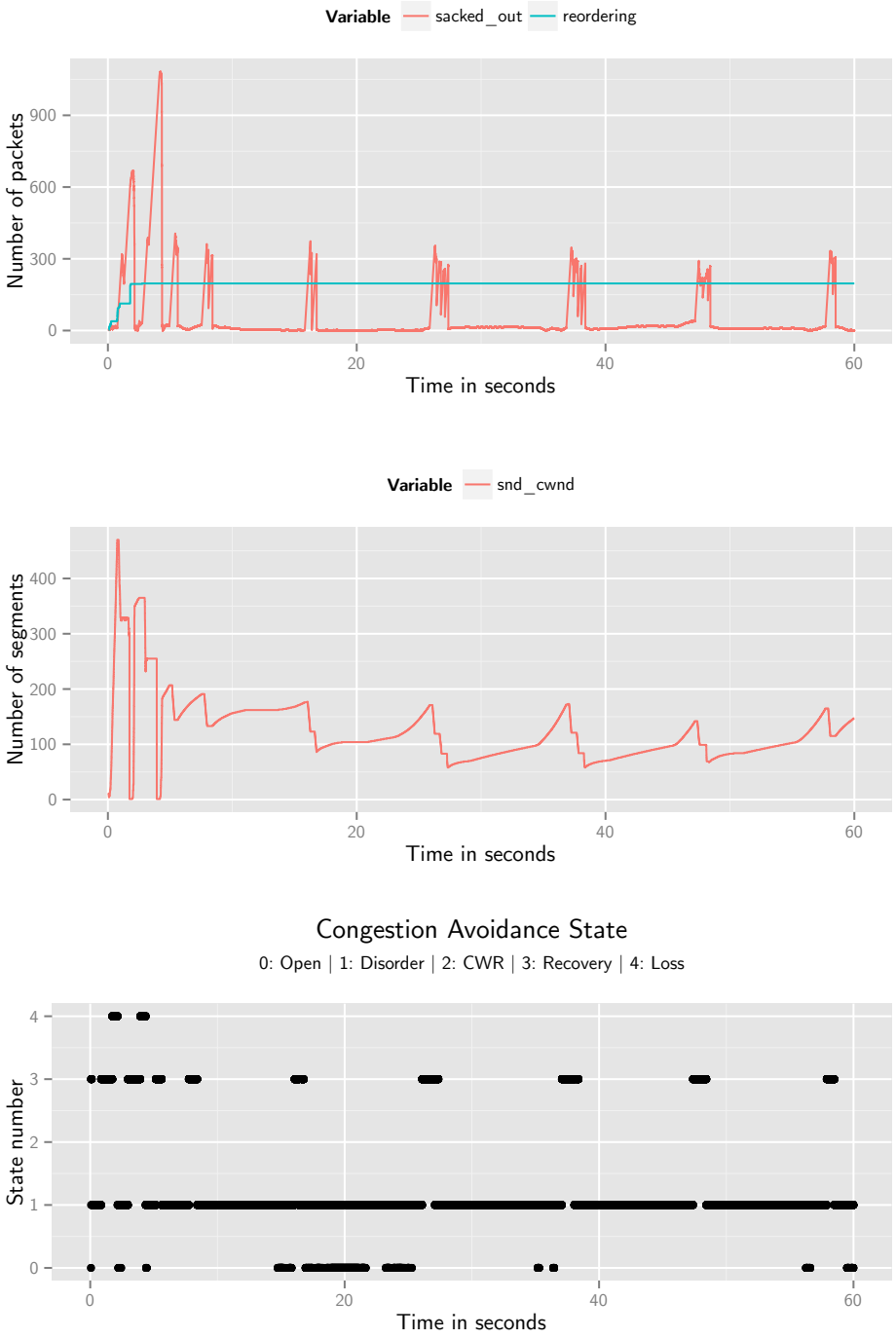
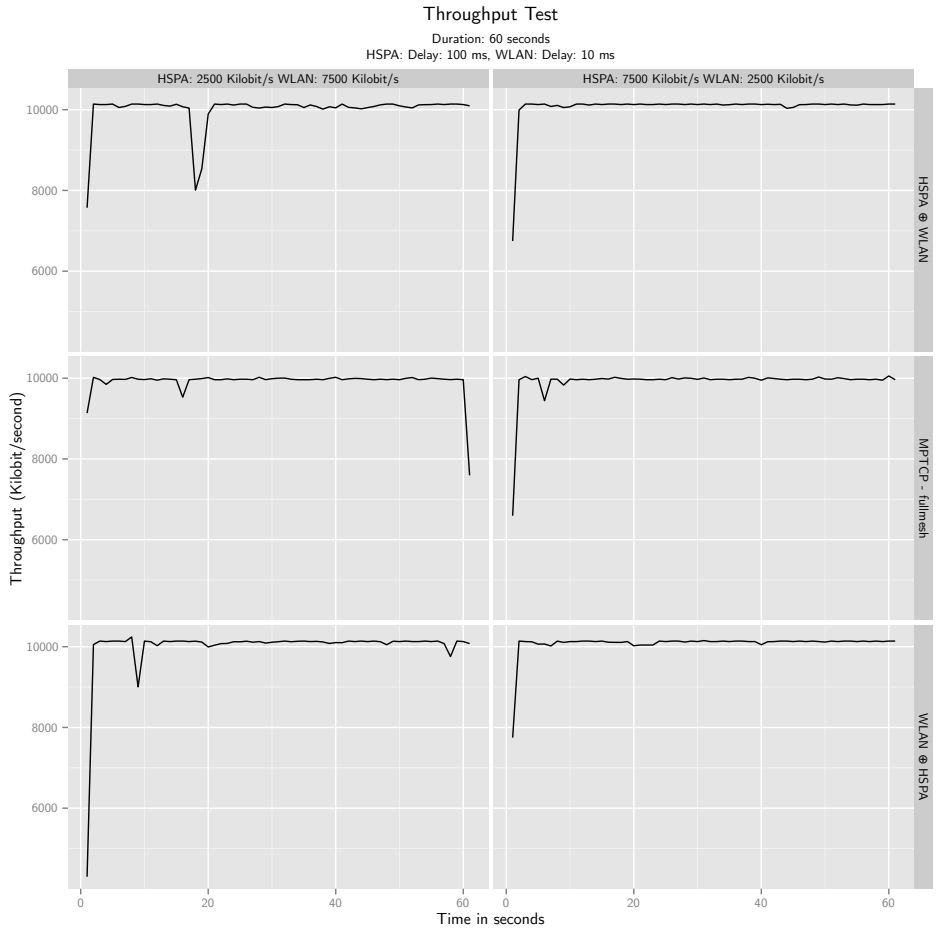
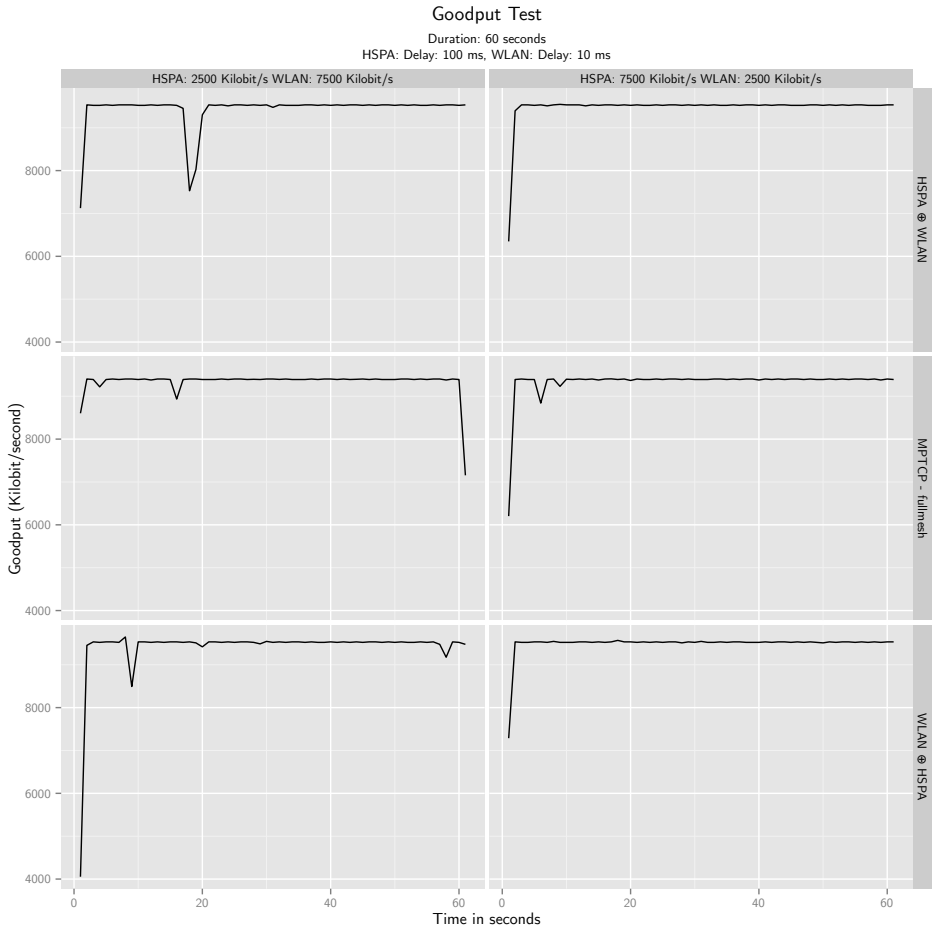


Figure 6.4: Test of Linux TCP's robustness against network reordering over the aggregated path WLAN $\oplus$ HSPA



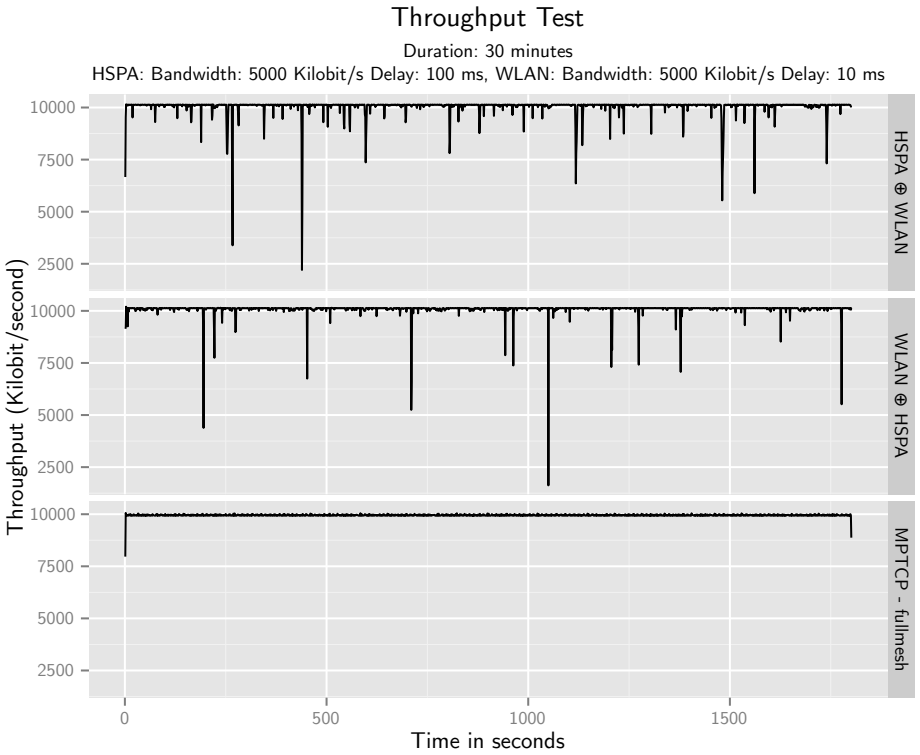
	Total data (MegaByte)	avg. Throughput (Megabit/s)	Configured Data Rate
WLAN ⊕ HSPA	76.25369	9.988701	HSPA: 2500Kbit/s WLAN: 7500Kbit/s
HSPA ⊕ WLAN	76.32675	9.979128	HSPA: 2500Kbit/s WLAN: 7500Kbit/s
MPTCP - fullmesh	75.61482	9.892854	HSPA: 2500Kbit/s WLAN: 7500Kbit/s
WLAN ⊕ HSPA	77.12707	9.989337	HSPA: 7500Kbit/s WLAN: 2500Kbit/s
HSPA ⊕ WLAN	76.95589	10.054855	HSPA: 7500Kbit/s WLAN: 2500Kbit/s
MPTCP - fullmesh	75.76427	9.857657	HSPA: 7500Kbit/s WLAN: 2500Kbit/s

Figure 6.5: Throughput test with different data rate and network path aggregation configurations



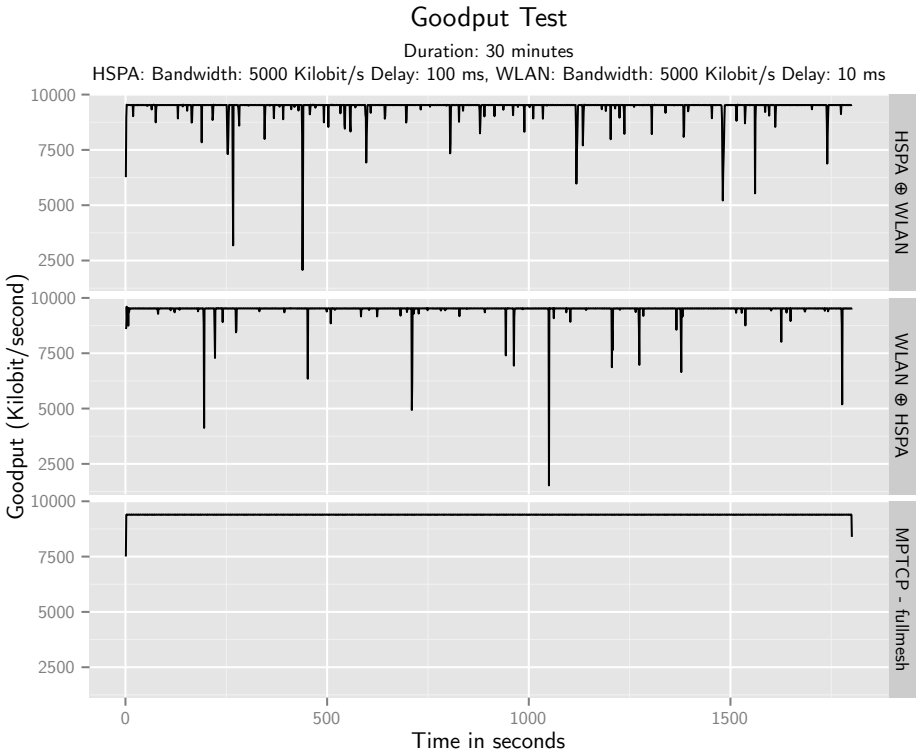
	Total data (MegaByte)	avg. Goodput (Megabit/s)	Configured Data Rate
WLAN ⊕ HSPA	71.78613	9.403481	HSPA: 2500Kbit/s WLAN: 7500Kbit/s
HSPA ⊕ WLAN	71.96794	9.409247	HSPA: 2500Kbit/s WLAN: 7500Kbit/s
MPTCP - fullmesh	71.18354	9.313099	HSPA: 2500Kbit/s WLAN: 7500Kbit/s
WLAN ⊕ HSPA	72.62616	9.406389	HSPA: 7500Kbit/s WLAN: 2500Kbit/s
HSPA ⊕ WLAN	72.41550	9.461620	HSPA: 7500Kbit/s WLAN: 2500Kbit/s
MPTCP - fullmesh	71.30319	9.277228	HSPA: 7500Kbit/s WLAN: 2500Kbit/s

Figure 6.6: Goodput test with different data rate and network path aggregation configurations



	avg. Throughput (Kilobit/s)	Total Data (GigaByte)
WLAN ⊕ HSPA	10075.53976	2.270711432
HSPA ⊕ WLAN	10045.7968	2.2625501
MPTCP - fullmesh	9950.20713	2.240153532

Figure 6.7: Throughput test with equal data rate for each network path



	avg. Goodput (Kilobit/s)	Total Data (GigaByte)
WLAN ⊕ HSPA	9486.16091	2.13788388
HSPA ⊕ WLAN	9454.29791	2.129330616
MPTCP - fullmesh	9392.46103	2.1145846

Figure 6.8: Goodput test with equal data rate for each network path

Chapter 7

Conclusion

In section 1.2 we stated that we needed to solve the problems listed below to find the true origin of how Linux TCP is robust against network reordering how good it is doing that job in relation to sending traffic over an aggregating path which includes two network paths with differences in both delay and bandwidth.

1. Recreate the testbed Kaspar used in [25] and test this out on a newer version of the Linux kernel.
2. Investigate how Linux TCP is robust against network reordering.
3. Compare the performance of our network layer aggregation against MPTCP with different test cases where we experiment with the ratio and order of bandwidth and delay to the aggregated paths.

We have successfully recreated the testbed Kaspar had in [25] and tested it on the newest stable version of the Linux kernel which is now 4.0.

We have found out how Linux TCP is robust against network reordering, which is explained more in detail in chapter 4 and shortly summarized in the list below:

- To proactively prevent false fast retransmit, Linux TCP uses SACK to find and store the maximum detected reordering length in a new variable named `reordering`. The `reordering` variable is than used as the new dupThresh variable to the fast retransmit algorithm.
- To recover from false fast retransmits, Linux TCP uses D-SACK as a mechanism to detect spurious retransmissions. If a spurious retransmission is proven, Linux TCP tries to the congestion reduction by reverting back to the old `snd_cwnd` logged before entering fast retransmit.

Our tests has also shown that Linux TCP is now so robust against network reordering that it outperformed MPTCP in relation to overall throughput and goodput. Linux TCP now also manage to utilize approximately the sum of all the available bandwidth regardless of how its configuration in relation to low and high RTT for the primary and secondary path.

Chapter 8

Future Work

- Deploy and test our network layer aggregation on a practical testbed where we have competing traffic. The receiver must here be extended with some NAT punching mechanism in order to make it possible. Otherwise would the packets traversing the secondary path be dropped by middle boxes such as NATs.
- Make a TCP extension or an application layer protocol to auto configure multi-homed end hosts to transparently send data over multiple network paths.
- Make the bandwidth ratio adaptively adjust in the sender side module.
- Find out how to fix the periodically small amount of loss in throughput over the network aggregated paths.

Bibliography

[1] 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.

[2] M. Allman, V. Paxson, and E. Blanton. *TCP Congestion Control*. RFC 5681 (Draft Standard). Internet Engineering Task Force, Sept. 2009. URL: http://www.ietf.org/rfc/rfc5681.txt.

[3] M. Allman, V. Paxson, and W. Stevens. *TCP Congestion Control*. RFC 2581 (Proposed Standard). Obsoleted by RFC 5681, updated by RFC 3390. Internet Engineering Task Force, Apr. 1999. URL: http://www.ietf.org/rfc/rfc2581.txt.

[4] F. Baker and P. Savola. *Ingress Filtering for Multihomed Networks*. RFC 3704 (Best Current Practice). Internet Engineering Task Force, Mar. 2004. URL: http://www.ietf.org/rfc/rfc3704.txt.

[5] Yaron Benita. "Kernel Korner: Analysis of the HTB Queuing Discipline". In: *Linux J.* 2005.131 (Mar. 2005), pp. 13–. ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=1053490.1053503.

[6] S. Bhandarkar et al. *Improving the Robustness of TCP to Non-Congestion Events*. RFC 4653 (Experimental). Internet Engineering Task Force, Aug. 2006. URL: http://www.ietf.org/rfc/rfc4653.txt.

[7] D. Borman et al. *TCP Extensions for High Performance*. RFC 7323 (Proposed Standard). Internet Engineering Task Force, Sept. 2014. URL: http://www.ietf.org/rfc/rfc7323.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] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. "TCP Vegas: New Techniques for Congestion Detection and Avoidance". In: *SIGCOMM Comput. Commun. Rev.* 24.4 (Oct. 1994), pp. 24–35. ISSN: 0146-4833. DOI: 10.1145/190809.190317. URL: http://doi.acm.org/10.1145/190809.190317.

[10] Claudio Casetti et al. "TCP Westwood: End-to-end Congestion Control for Wired/Wireless Networks". In: *Wirel. Netw.* 8.5 (Sept. 2002), pp. 467–479. ISSN: 1022-0038. DOI: 10.1023/A:1016590112381. URL: http://dx.doi.org/10.1023/A:1016590112381.

[11] Dah Ming Chiu et al. “Can Network Coding Help in P2P Networks?” In: *Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks, 2006 4th International Symposium on*. Apr. 2006, pp. 1–5. DOI: 10.1109/WIOPT.2006.1666439.

[12] D. E. Comer et al. “Computing As a Discipline”. In: *Commun. ACM* 32.1 (Jan. 1989). Ed. by Peter J. Denning, pp. 9–23. ISSN: 0001-0782. DOI: 10.1145/63238.63239. URL: http://doi.acm.org/10.1145/63238.63239.

[13] M. Duke et al. *A Roadmap for Transmission Control Protocol (TCP) Specification Documents*. RFC 4614 (Informational). Obsoleted by RFC 7414, updated by RFC 6247. Internet Engineering Task Force, Sept. 2006. URL: http://www.ietf.org/rfc/rfc4614.txt.

[14] Sally Floyd and Kevin Fall. “Promoting the Use of End-to-end Congestion Control in the Internet”. In: *IEEE/ACM Trans. Netw.* 7.4 (Aug. 1999), pp. 458–472. ISSN: 1063-6692. DOI: 10.1109/90.793002. URL: http://dx.doi.org/10.1109/90.793002.

[15] S. Floyd et al. *An Extension to the Selective Acknowledgement (SACK) Option for TCP*. RFC 2883 (Proposed Standard). Internet Engineering Task Force, July 2000. URL: http://www.ietf.org/rfc/rfc2883.txt.

[16] A. Ford et al. *Architectural Guidelines for Multipath TCP Development*. RFC 6182 (Informational). Internet Engineering Task Force, Mar. 2011. URL: http://www.ietf.org/rfc/rfc6182.txt.

[17] A. Ford et al. *TCP Extensions for Multipath Operation with Multiple Addresses*. RFC 6824 (Experimental). Internet Engineering Task Force, Jan. 2013. URL: http://www.ietf.org/rfc/rfc6824.txt.

[18] Sangtae Ha, Injong Rhee, and Lisong Xu. “CUBIC: A New TCP-friendly High-speed TCP Variant”. In: *SIGOPS Oper. Syst. Rev.* 42.5 (July 2008), pp. 64–74. ISSN: 0163-5980. DOI: 10.1145/1400097.1400105. URL: http://doi.acm.org/10.1145/1400097.1400105.

[19] Junghee Han and F. Jahanian. “Impact of path diversity on multi-homed and overlay networks”. In: *Dependable Systems and Networks, 2004 International Conference on*. June 2004, pp. 29–38. DOI: 10.1109/DSN.2004.1311874.

[20] T. Henderson et al. *The NewReno Modification to TCP's Fast Recovery Algorithm*. RFC 6582 (Proposed Standard). Internet Engineering Task Force, Apr. 2012. URL: http://www.ietf.org/rfc/rfc6582.txt.

[21] C. Hopps. *Analysis of an Equal-Cost Multi-Path Algorithm*. RFC 2992 (Informational). Internet Engineering Task Force, Nov. 2000. URL: http://www.ietf.org/rfc/rfc2992.txt.

[22] V. Jacobson, R. Braden, and D. Borman. *TCP Extensions for High Performance*. RFC 1323 (Proposed Standard). Obsoleted by RFC 7323. Internet Engineering Task Force, May 1992. URL: http://www.ietf.org/rfc/rfc1323.txt.

[23] A. Jayasumana et al. *Improved Packet Reordering Metrics*. RFC 5236 (Informational). Internet Engineering Task Force, June 2008. URL: http://www.ietf.org/rfc/rfc5236.txt.

[24] P. Karn and C. Partridge. "Improving Round-trip Time Estimates in Reliable Transport Protocols". In: *SIGCOMM Comput. Commun. Rev.* 17.5 (Aug. 1987), pp. 2–7. ISSN: 0146-4833. DOI: 10.1145/55483.55484. URL: http://doi.acm.org/10.1145/55483.55484.

[25] Dominik Kaspar. "Multipath Aggregation of Heterogenous Access Networks". Doctoral thesis. University of Oslo, 2011. URL: http://urn.nb.no/URN:NBN:no-30595.

[26] Ka-Cheong Leung, Victor O. K. Li, and Daiqin Yang. "An Overview of Packet Reordering in Transmission Control Protocol (TCP): Problems, Solutions, and Challenges". In: *IEEE Trans. Parallel Distrib. Syst.* 18.4 (Apr. 2007), pp. 522–535. ISSN: 1045-9219. DOI: 10.1109/TPDS.2007.1011. URL: http://dx.doi.org/10.1109/TPDS.2007.1011.

[27] Matthew Mathis and Jamshid Mahdavi. "Forward Acknowledgement: Refining TCP Congestion Control". In: *SIGCOMM Comput. Commun. Rev.* 26.4 (Aug. 1996), pp. 281–291. ISSN: 0146-4833. DOI: 10.1145/248157.248181. URL: http://doi.acm.org/10.1145/248157.248181.

[28] M. Mathis et al. *TCP Selective Acknowledgment Options*. RFC 2018 (Proposed Standard). Internet Engineering Task Force, Oct. 1996. URL: http://www.ietf.org/rfc/rfc2018.txt.

[29] D. Miras, M. Bateman, and S. Bhatti. "Fairness of High-Speed TCP Stacks". In: *Advanced Information Networking and Applications, 2008. AINA 2008. 22nd International Conference on*. Mar. 2008, pp. 84–92. DOI: 10.1109/AINA.2008.143. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4482693.

[30] 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.

[31] V. Paxson et al. *Computing TCP's Retransmission Timer*. RFC 6298 (Proposed Standard). Internet Engineering Task Force, June 2011. URL: http://www.ietf.org/rfc/rfc6298.txt.

[32] J. B. Postel. *Transmission Control Protocol*. RFC 793. Internet Engineering Task Force, Sept. 1981, p. 85. URL: http://www.rfc-editor.org/rfc/rfc793.txt.

[33] The Bufferbloat project. *Best Practices for Benchmarking CoDel and FQ CoDel (and almost any other network subsystem!)* 2014. URL: https://www.bufferbloat.net/projects/codel/wiki/Best_practices_for_benchmarking_Codel_and_FQ_Codel (visited on 04/03/2015).

[34] A. Ramaiah, R. Stewart, and M. Dalal. *Improving TCP's Robustness to Blind In-Window Attacks*. RFC 5961 (Proposed Standard). Internet Engineering Task Force, Aug. 2010. URL: http://www.ietf.org/rfc/rfc5961.txt.

[35] P. Sarolahti et al. *Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP*. RFC 5682 (Proposed Standard). Internet Engineering Task Force, Sept. 2009. URL: http://www.ietf.org/rfc/rfc5682.txt.

[36] W. Simpson. *The Point-to-Point Protocol (PPP)*. RFC 1661 (INTERNET STANDARD). Updated by RFC 2153. Internet Engineering Task Force, July 1994. URL: http://www.ietf.org/rfc/rfc1661.txt.

[37] K. Sklower et al. *The PPP Multilink Protocol (MP)*. RFC 1990 (Draft Standard). Internet Engineering Task Force, Aug. 1996. URL: http://www.ietf.org/rfc/rfc1990.txt.

[38] 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.

[39] R. Stewart et al. *Stream Control Transmission Protocol*. RFC 2960 (Proposed Standard). Obsoleted by RFC 4960, updated by RFC 3309. Internet Engineering Task Force, Oct. 2000. URL: http://www.ietf.org/rfc/rfc2960.txt.

[40] D. Thaler and C. Hopps. *Multipath Issues in Unicast and Multicast Next-Hop Selection*. RFC 2991 (Informational). Internet Engineering Task Force, Nov. 2000. URL: http://www.ietf.org/rfc/rfc2991.txt.

[41] Kang Xi, Yulei Liu, and H.J. Chao. "Enabling flow-based routing control in data center networks using Probe and ECMP". In: *Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on*. Apr. 2011, pp. 608–613. DOI: 10.1109/INFCOMW.2011.5928885.

[42] Ming Zhang et al. "RR-TCP: a reordering-robust TCP with DSACK". In: *Network Protocols, 2003. Proceedings. 11th IEEE International Conference on*. Nov. 2003, pp. 95–106. DOI: 10.1109/ICNP.2003.1249760.