UNIVERSITY OF OSLO
Department of Informatics

A Client-side Split ACK Tool for TCP Slow Start Investigation

Master Thesis

Rolf Erik G. Normann

2011-08-01

A Client-side Split ACK Tool for TCP Slow Start Investigation

Rolf Erik G. Normann

2011-08-01

Abstract

The start-up phase in TCP is called Slow Start, and is followed by Congestion Avoidance. The Slow Start phase is becoming a bottleneck of communication in the Internet today. There are several proposals to improve TCP's Slow Start phase and it has recently gotten much attention because Google proposes to use a larger starting point (Initial Window, IW).

We present a tool that, by splitting acknowledgements of TCP into multiple pieces at the beginning of a connection, to trick a host into quickly sending a larger number of packets than normally intended. We tested the method against different Operating Systems, commercial companies and the top 600 most visited web sites in the world. Test results indicate that, while many hosts do not react to our tool, some do, and they are probably enough to use the tool for measurements.

Acknowledgments

I would like to thank my supervisor Dr. Michael Welzl for his guidance, inspiring attitude and the friendly atmosphere. Without his invaluable feedback, this work would not have been possible.

I would also thank my brother Fredrik Normann and my friend Vegard Lunde for proofreading the thesis. I would also thank Chris Carlmar for the support with the IPtables modification.

My special thanks goes to my girlfriend Camilla Heslien for emotional and moral support throughout the whole master period. Without her patience and support, this thesis would never been finished. And at last, a big thanks to my parents for believing in me.

Thank you all so much!

Oslo, July 31. 2011
Rolf Erik G. Normann

Contents

Chapter 1

Introduction

The Internet is a collection of different servers, computers and devices. They all run some kind of an operating system, and most of them support IPv4 / IPv6 and utilize the Transmission Control Protocol (TCP) to communicate. In fact 85-95% of the packets sent on the Internet are TCP packets [1], and this has not changed over the course of time, since file sharing (e.g. BitTorrent), HTTP, and video streaming (e.g. YouTube, Justin.tv) are utilizing TCP to communicate.

The start-up phase in TCP is called the Slow Start phase. The current Slow Start algorithm was developed by Van Jacobson in 1988 [2], and was described in the article «Congestion avoidance and control» [3]. The 1988 version of Slow Start is still used today, but with some improvements and modifications [4]. The classic Slow Start algorithm starts off with 3 segments and doubles the sending rate for every Round Trip Time (RTT), giving the connection an exponential growth. This approach was a good choice when the the maximum bandwidth was around 56 Kb with a dial-up modem. Today the average connection speed has drastically increased, and a common connection can range from 1 MBit up to 100 MBit. In fact, 99% of all Norwegians got access broadband of 1000 KBit or more [5]. So the Slow Start approach today is too slow, when compared to the higher speeds of the connections.

To illustrate this with an example, we have a 56 Kb/s connection, and a packet

size of 1500 bytes (i.e maximum MTU), and a delay of 100 ms. To achieve the maximum bandwidth of the 56 Kb/s connection, it would only require one RTT. To achieve the maximum bandwidth for a 10 MBit connection, with the same delay and packet size, it would take around 6 RTTs before the connection would reach the maximum bandwidth. Note that in any of the two scenarios, there are no network congestion, drops or delays.

There are different standards that tell a client how to communicate, but the implementations of the standards are not the same in every Operating System (OS). The different Linux distributions (e.g. Ubuntu [6]) have a tendency to implement most of the new standards in the kernel. This gives the end user the opportunity to choose from the different standards, features and algorithms. But there is a difference between implementing a feature/algorithm, and turning the feature/algorithm on by default. The distribution does come with predefined algorithms and features, and it is up to the end user to turn on (or off), and configure the other features. In the end, they give the end user the opportunity to use the different features and the more experimental algorithms. To give an example, we can look at the Stream Control Transmission Protocol (SCTP) [7]. This protocol is implemented in the more recent Linux kernels, but is by default turned off [8].

The big commercial companies (e.g. Microsoft, Apple, Google, Oracle) are following the standards, but they may have different implementations of the same standards. There may be similarities, but its hard for the end-user to see the differences in the implementations and configurations. An OS could have a source code that is restricted (i.e. Windows, OS X), and most OSs do have configurations that the end-user needs to setup. Even in Linux, where the source code is open and free, this is still a problem. The source code could be changed, and as a client, there is no easy way to find out what was changed. The implementation of Slow Start could also differ from OS to OS. And because it is not an easy task to see how the different algorithms behave between OSs, we need a tool to probe the different servers.

In this thesis we implemented a tool that can utilize split Acknowledgments (ACK) - ACKs which acknowledge less than a whole packet. Such ACKs have been reported to provoke unusual behavior by a server [9]. And the split ACK tool probes

a server to see if we achieved any results and/or benefits from this approach. We wanted to know in this thesis how Slow Start was implemented in the different operating systems, and how the different operating systems reacted to a client side split ACK approach. The goal was to use the split ACK approach to observe the difference in the behavior of the servers, and also to see if we could achieve a faster Slow Start phase by utilizing client side split ACKs.

The implementation of the split ACK tool was targeted for a real world scenario, and was done only by modifying TCP ACKs on the client side. The results showed that split ACK approach did increased the traffic flow, and that the split ACK tool was successful in splitting ACKs, and measuring the server response. The split ACK tool could therefore be utilized as en efficient method to measure the response of a split ACK approach in a real world scenario.

Chapter 2

Background and Related Work

In this chapter we will give some background on the Internet Protocol (IP) and Transmission Control Protocol (TCP). The chapter will also go into some detail about Appropriate Byte Counting (ABC), congestion avoidance, Slow Start, acknowledgments (ACKs), and split ACKs. This background information is important in order to understand the design and implementation of the Split ACK tool and to understand the final results.

2.1 The Internet Protocol

Internet Protocol (IP) is a connectionless protocol that communicates between two endpoints on a packet-switched network. The protocol is defined in the Request for Comments (RFC)791 [10].

For a device to be able to communicate on an IP network, it requires an IP address. Each connected device is required to have its own unique addresses, whether it be a intranet or the Internet. This is because the IP address is the identity of the connected devices, and a router is using this identity to send the correct packets to the correct device. So in a network where two devices have the same IP address an IP conflict will occur. A router will not be able to determine where the data should be sent to and the devices will lose the connection.

The IP work on the best effort principle, and will try to deliver a packet, but give no guarantees for that the packet will be lost, duplicated, delayed or out of order. The main task for IP is to provide addresses to identify the route to the destination.

A device will never need to know the path to the receiver, but it only assumes that the receiver exists and sends the packet to a router until it arrives at the target. If there is a path to the receiver, the router will deliver the packet, if not, the packet may be dropped or forwarded to the next router.

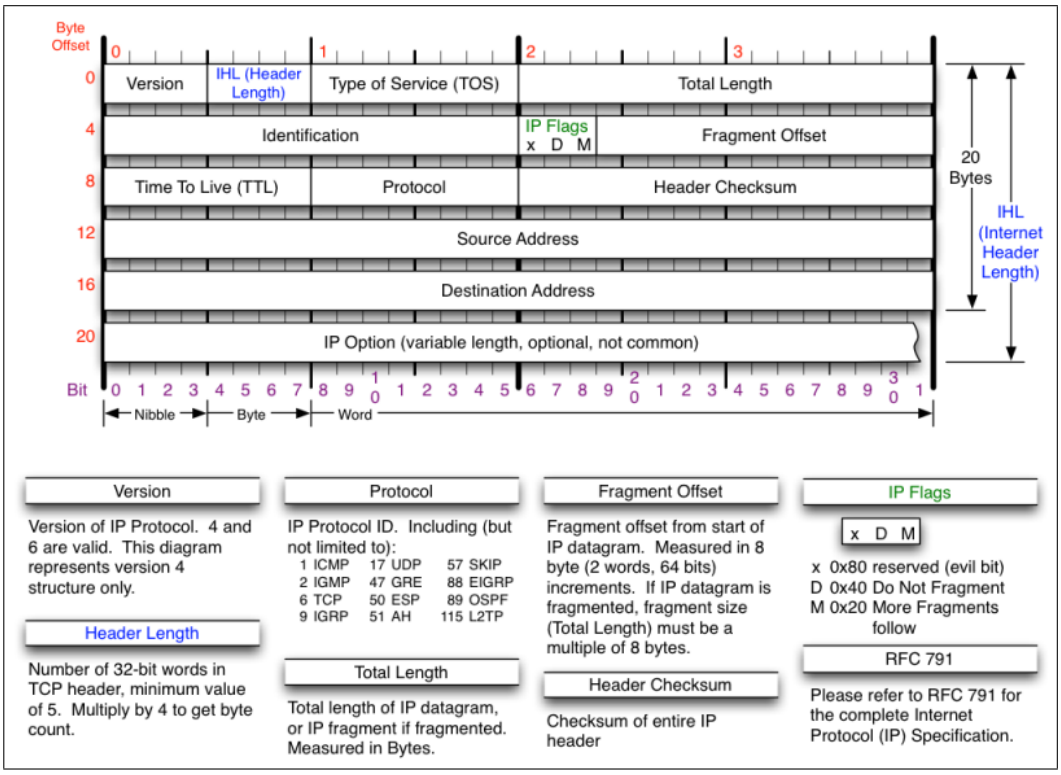


Figure 2.1: Ipv4 Header. This illustration is used by permission of author Matt Baxter.

The IP header is shown in illustration 2.1. In this thesis we are only going to do small changes to the IP header. The modifications are going to be done to the TTL value in the IP header.

Note that the are two different versions of the IP, and the most widespread version of IP is version 4 (IPv4). The other IP version is the successor, IP version 6 (IPv6). IPv6 is in most cases utilized in combination with IPv4, and has not yet taken over.

Because of the worldwide use of IPv4, we are going to focus on the IPv4 version of the protocol in this thesis.

2.1.1 Time To Live

The IP header is split up in different header fields that have different properties. The IP destination field has the IP address to the destination and the source field holds the clients address. As seen in figure 2.1 on the preceding page the IP header has a TTL field that is utilized to eliminate loops in a network. The time to live (TTL) value is a mechanism to eliminate unwanted traffic in a network. TTL is a counter that keeps track of the number of hops the packet has left to go, and if the TTL value reaches zero, the packet will get dropped by the next router. These mechanisms are used to avoid that packets go in an endless loop, and to limit the queue size in routers.

2.1.2 IP Checksum

In figure 2.1 on the facing page of the IP header and in figure 2.2 on page 9 there is a field that is responsible for verifying any changes in the header or payload. In IPv4 it is called the IP Header checksum. A checksum can be defined like this [11]:

> "A checksum or hash sum is a fixed-size datum computed from an arbitrary block of digital data for the purpose of detecting accidental errors that may have been introduced during its transmission or storage."

Therefore, if any changes were done to the header in the transmission, a new calculation of the checksum would differ from the value stored in the header. Routers do verify the checksum of incoming packets in the same way, they recalculate the checksum and compare it with the one stored in the header. If the incoming packet was modified or corrupted the router would simply drop the packet and wait for a retransmission.

2.2 Transmission Control Protocol

Transmission Control Protocol (TCP) is layered between the IP and the application layer above, and act as the transport layer. This is the same as described in the OSI reference model, where the main tasks of the transport layer is to handle the end to end communication and the reliability of a connection. The Transmission Control Protocol is described in the RFC793 [12], and updated by the RFC1122 [13] and RFC3168 [14]. TCP is one of many different transport layer protocols, but TCP is by the far most wide spread and implemented protocol in the Internet today. In fact as early as 1997, 95% of all packets on the Internet were transmitted using TCP [1].

2.2.1 TCP connection

To put it very simple, a TCP connection is a connection from a device to another device, over multiple (or single) underlying computers. This gives the illusion of a direct connection from device A to device B.

A device does not need to know if the connection is a local connection, or if the connection is routed through multiple servers over the Internet. All a devices needs to know is the address to the server destination (and the port). To establish a TCP connection, the first step is to initialize the connection to the specific IP and port. If the host does not respond the connection will time out. This differs from e.g email where the email address has to be valid, but the receiver does not need to be online. With TCP, the host needs to be online, and the address and port need to be correct.

As mentioned before, the IP is merly in charge of addressing, and because IP cannot guarantee reliability of segments, the client needs a transport protocol that can handle reliable delivering. TCP is based on best effort delivery and will try to provide a reliable and ordered delivery of a stream of bytes. It cannot guarantee this feature, but it will go very far to try to keep this guaranty. This is why TCP has algorithms that will go far to keep a reliable connection and to deliver/get the

data that the client wants to send/receive.

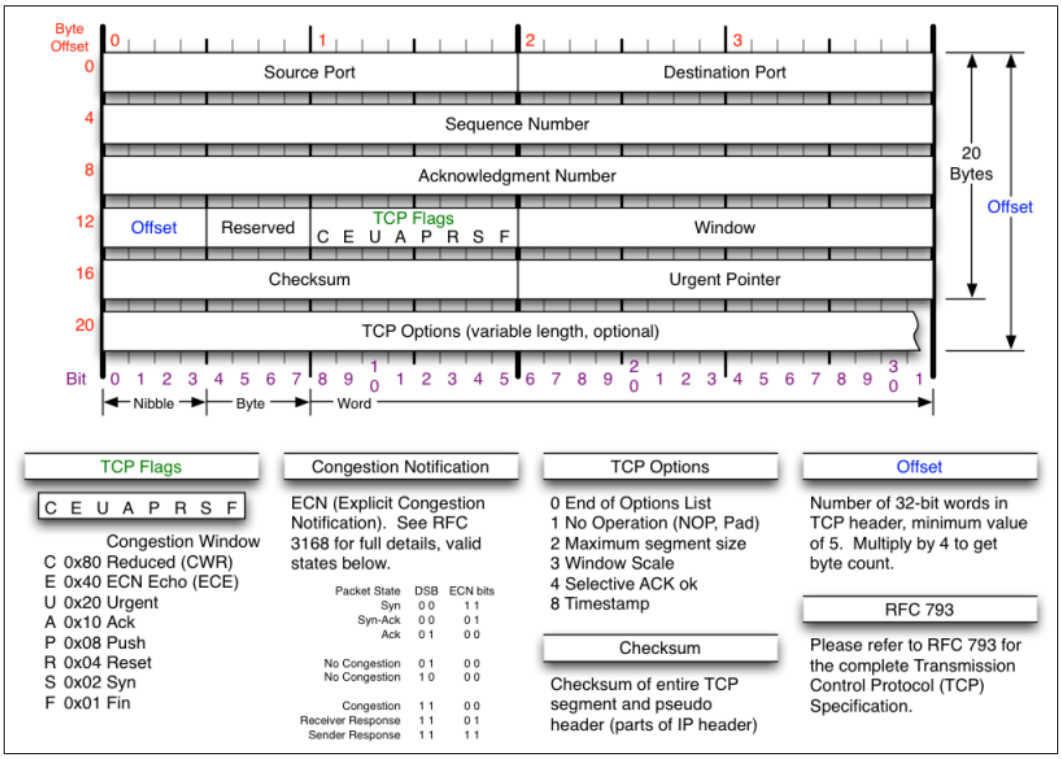


Figure 2.2: TCP Header. This illustration is used by permission of author Matt Baxter.

TCP is the main protocol for Internet communication today, and without TCP the Internet would not be the same as we know it today. The reason why TCP has survived for so long, is the constant improvement of the protocol, and the mechanisms it features. But TCP still has a few drawbacks that are not updated to match the speed of the modern Internet.

Note that TCP and IP are commonly known and referred to as TCP/IP [15], but in this thesis we will refer to TCP and IP as standalone protocols. This is done to avoid any confusion when we are describing TCP or IP in the design and implementation part of this thesis.

2.2.2 Acknowledgment of Packets

Reliability is one of the fundamental functions in TCP. The Internet gives no guarantee for data transfers, and works only on the best effort principle. So to be able to detect packet loss, congestion or reordering, we need reliability from the transport layer. TCP uses different mechanisms to guarantee reliability of a connection; two core functions are sequence numbers and acknowledgment numbers.

Sequence numbers identify each byte in a packet, and identify which stream data belong to (i.e. multiple connections from a single host). The sequence numbers are unique to every connection, regardless of any fragmentation, disordering or packet loss that might happen to a segment. The sequence number is stored in every TCP segment and it is a continuously growing number based on a random generated number that only the sender and receiver know.

The second mechanism is the use of acknowledgments (ACKs). Acknowledgments are used to confirm the arrival of incoming packets, and to verify that packets were not fragmented or disordered. An acknowledgment number is generated by using the sequence number of the incoming packet and the number of bytes that were received. The value is then stored in the TCP packet header, where the Acknowledgment number is the next expected sequence number and the and «ACK» flag is set. The sender would now know that the acknowledgment is for packet X, i.e all packets up to and including X have arrived, and the client can now send the next packet. If a sender did not receive an ACK for a packet, a predetermined timer will expire and a retransmission of the lost packet is triggered. And even if the server did receive the packet, but the ACK got dropped, the sender still needs to retransmit the packet to keep the reliability. The use of ACKs is one of the core functions in TCP, and is used in most of the algorithms like Slow Start, congestion avoidance, the detecting packet loss, and initializing of a new connection.

Selective Acknowledgment

Relying on a pure cumulative acknowledgment scheme, as in the original TCP protocol, could lead to a potential catastrophic effect on the TCP throughput. Con-

sider a scenario where we would send 15,000 bytes in 10 packets (1500 bytes per packet). If the first packet got lost and the other 9 did arrive at the host, in a pure cumulative acknowledgment protocol, the receiver cannot tell the client that it only received bytes 1500 - 15000, and that the 0 - 1499 bytes were lost. This potentially triggers the sender to retransmit all of the 15000 bytes.

Selective Acknowledgment (SACK) [16] is an addition to the TCP protocol as a strategy to eliminate this problem. SACK allows receivers to acknowledge discontinuous blocks of packets that were received correctly, and to alert the sender to the last sequence number of the last continuous packets received. The sender uses this information to retransmit the correct missing packets, so that the receiver gets a complete set of packets. In the previous scenario, where the sender would potentially retransmits all of the data, just because the first packets got lost, the SACK strategy would alert the host that only the first packets was lost. With the SACK option, the sender would see that it was only the first packet that got lost, and that it would only need to retransmit the first 1500 bytes.

2.2.3 Congestion

Congestion in a network can be compared to a traffic jam on a highway. The same problem occurs on the Internet, where a link is overloaded with traffic or where, as a consequence, a router queue is full. A definition of network congestion can be describe as when a link or node is so overloaded with traffic, that the quality of service deteriorates. So network congestion can be caused by queueing delay in a router, packet loss or blocking of a new incoming connection.

TCP will try to control network congestion, via multiple algorithms that can detect it (e.g. the detection of three identical ACKs). This mechanism would indicate that the sender that needs to back off, because the receiver may be flooded by data packets. The TCP congestion algorithms will take necessary steps to avoid further congestion, and will try to keep up the speed of the connection, without causing congestion. This will be explained thoroughly in the section of congestion avoidance.

2.3 TCP algorithms

TCP has different algorithms and mechanisms that are a part of the standard of TCP. Some of these mechanisms and algorithms are Slow Start, congestion avoidance, fast retransmit, fast recovery and – an important element in this thesis – appropriate byte counting.

2.3.1 TCP - Three-Way-Handshake

When a client wants to establish a connection, it needs to verify that the server is online and that it is ready to process the connection. The verification and the initializing of a connection is done by using the three-way-handshake [12]. The algorithm makes sure that a server is ready to process a incoming connection, and secures the initializing of the new connection by exchanging sequence numbers between the client and the server. The procedure of how the algorithm works is split into three steps:

1. The first step is a TCP synchronize packet where the SYN flag in the TCP header is set. The Client sets a sequence number X which is randomly generated. The randomly generated sequence number is for security and verification between the client and the server. The client sends the SYN packet to the server to indicate that the client wants to establish a connection.

2. The second step is when the server responds to the client by sending a packet with a synchronize acknowledgment, where the SYN and ACK flags are set. The server also generates a random Y sequence-number and sets the acknowledgment number to X+1.

3. The third and final step is when the client receives the SYN-ACK and responds with an an acknowledgment packet, where the ACK flag is set. The client also needs to set the acknowledgment number to Y+1 and the sequence-number to X+1, thus indicating that the connection is established.

The tree-way handshake is shown in the figure 2.3 on the facing page. After the third and final step is done, the client can start to send or request data from the

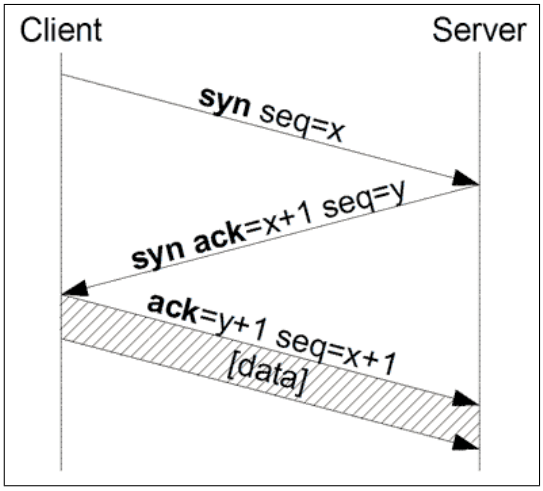


Figure 2.3: Three-way-handshake

server.

2.3.2 TCP - Slow Start

TCP Slow Start is an algorithm to control congestion in the early stages of a connection. This stage of a TCP connection is called the Slow Start phase. The next phase is the Congestion Avoidance, where the next step is to control the congestion. Slow Start is defined in RFC5681 [4] together with congestion avoidance. TCP needs different algorithms for controlling congestion in the early stages of a connection. This is because the knowledge of the network state is very limited.

Basic Slow Start [17] starts after the connection is established with a successful three-way-handshake. As the name Slow Start implies, the client starts the connection slow, and increases the congestion window (cwnd) for each acknowledge segment. In a perfect scenario, where the server has acknowledged every packet it receives, the cwnd will double for every round trip time (RTT), thus giving cwnd an exponential growth. As mentioned before, this behavior is only seen where the segments do not get lost, delayed or retransmitted. Even if the server would use delayed ACKs, the cwnd would still get an exponential growth, but would take longer and use more packets to reach the same size of the cwnd.

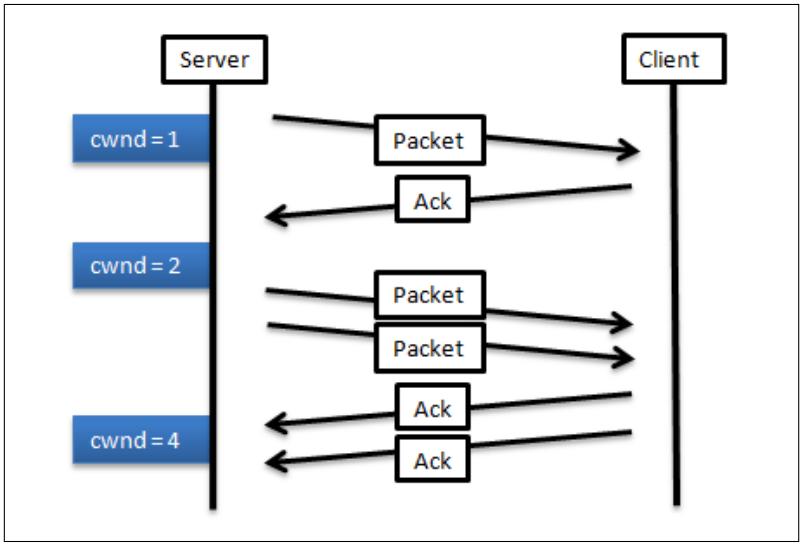


Figure 2.4: Slow Start packet flow

Figure 2.4 shows how Slow Start looks in the first few RTT and how the the first packets are sent, and how the client reacts to each acknowledgment that it receives by increasing the cwnd.

2.3.3 TCP - Congestion Avoidance

Congestion Avoidance is the next algorithm to control congestion. Congestion Avoidance is less aggressive in increasing the cwnd and roughly increases the cwnd with one full-size segment per RTT. The are a few congestion avoidance algorithms (e.g. NewReno, and the non-standard Vegas and Cubic) with different approaches to increasing the cwnd and handling network congestion. RFC5681 [4] has three rules for increasing the cwnd during the congestion avoidance:

- MAY increment cwnd by Sender Maximum Segment Size (SMSS) bytes
- SHOULD increment cwnd per equation once per RTT
- MUST NOT increment cwnd by more than SMSS bytes

The RFC [4] also has a second alternative of increasing the cwnd during Congestion Avoidance, and this equation is given in listing 2.1 on the facing page.

Listing 2.1: Congestion Avoidance - increase of cwnd

```
1    cwnd += SMSS*SMSS/cwnd
```

The different algorithms and equations have in common that they are less aggressive than Slow Start in increasing the cwnd.

Slow Start and Congestion avoidance

Slow Start is at the beginning of a connection, and congestion avoidance phase is after Slow Start, and continues for the rest of the connection, unless a time-out occurs which would put a TCP sender back in Slow Start.

- Slow Start starts off with a very small cwnd and lets it grow accordingly to the number of ACKs that the client receive. At this point the cwnd is less than a parameter called the «Slow Start threshold» (ssthresh). $cwnd < ssthresh$
- If the client experiences no loss or no retransmits, the cwnd is going to increase rapidly and would grow exponentially, until cwnd is higher or equal to ssthresh. $cwnd >= ssthresh$
- After cwnd hits ssthresh, Congestion Avoidance is taking over and the cwnd gets a linear growth. The connection continues to use Congestion Avoidance until the connection is terminated, or a time-out occurs.

2.4 Misbehaving Sender/Receiver in TCP

TCP is built on trust, and that both parties in a connection are following the specification of the different TCP standards. The problem is that not every client on a network can be trusted to follow the standard for TCP communications. A client may want to increase the performance compared to the other connected clients, which can be unfair to the other connected clients

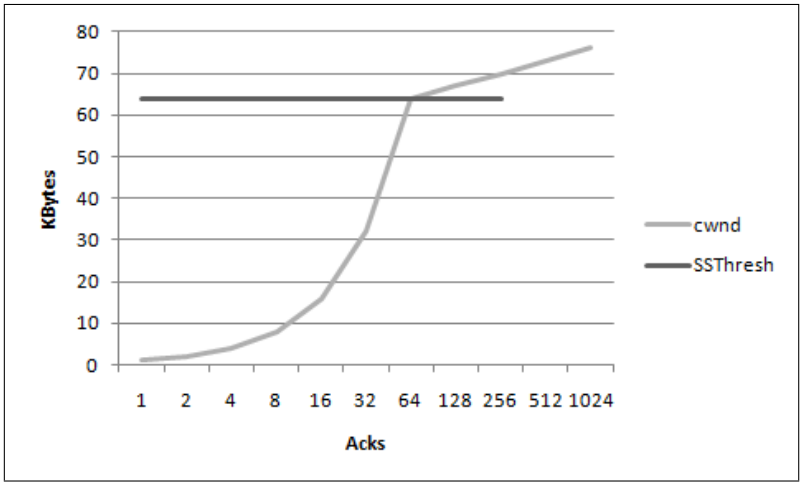


Figure 2.5: Slow Start versus Congestion avoidance

2.4.1 TCP Split ACK in Slow Start

TCP Split ACK is a client side modification to TCP. The split ACK modification can increase the growth of the cwnd in Slow Start and congestion avoidance. Stefan Savage et al. described this in the article, «TCP congestion control with a misbehaving receiver» [18], where a client would trick TCP to increase the cwnd faster than intended in the original algorithm.

As we will discuss in section 2.4.3 on page 20, split ACKs to not have an effect on a modern TCP in Congestion Avoidance. TCP split ACK can, however potentialy be utilized to improve the Slow Start phase in TCP. This can lead to a faster Slow Start, so that a client can accelerate the growth of the cwnd and better utilize the available bandwidth.

To generate Split acknowledgments, a client would need to split an ACK up into one or more pieces and send them all to the sender in the correct order. Because the server essentially lets cwnd grow for every ACK it receives, this would trick the server to increase the cwnd with more than the one segment per ACK.

To understand how split ACK works, we can describe a scenario where a client would utilize split ACK to increase the growth of cwnd. A client receives two packets, and would normally respond with two ACKs. With split ACK, the client

splits each ACK up in multiple ACKs. Each split ACK has a lower acknowledgment number than the original ACK. The client sends out the split ACKs with the lowest acknowledgment number first and and the original ACK last. The host will respond by increasing the cwnd for each incoming ACK. By splitting each ACK into two, the client would be tricking the server to send eight packs the next time, instead of the expected four packets. This is because the server increases the cwnd for each incoming ACK. The results would be even higher if the client would split the first ACKs into smaller pieces, thus triggering the server to increase the cwnd by an even larger value. Figure 2.6 illustrates how split ACK could increase the cwnd over one RTT.

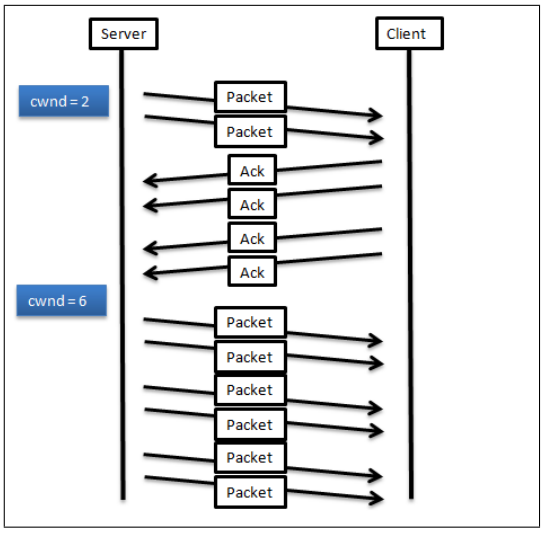


Figure 2.6: Slow Start with 2x SplitAcks

Benefits of TCP Split ACK in Slow Start

Split ACK is beneficial in scenarios where the Slow Start phase is the bottleneck and the bandwidth is not. The utilization of split ACK is a method to get more data faster, and with a shorter Slow Start phase. The client can get the server to adapt to the desired threshold in the Slow Start phase without doing any modification on the server side.

E.g in a scenario where we would have a large number of connections to a single

server, and each connected client would only need a small amount of data, the utilization of split ACK could increase the performance of the clients and the server. Note that this is true if the file size is less than $(ssthresh * 2) - sizeof(IW)$, (i.e. MSS 1460 bytes, IW 3 packets, ssthresh 64 Kb), file size less than 123,7 Kb. With a small file size, the client would be limited by TCP Slow Start phase, and not by the bandwidth.

If, in the same scenario, all the clients would utilize Split ACK to decrease the overall connection time, the server would actually be able to handle more connections. This is because each client would use fewer RTTs to download the file, and the overall connection time per client would decrease.

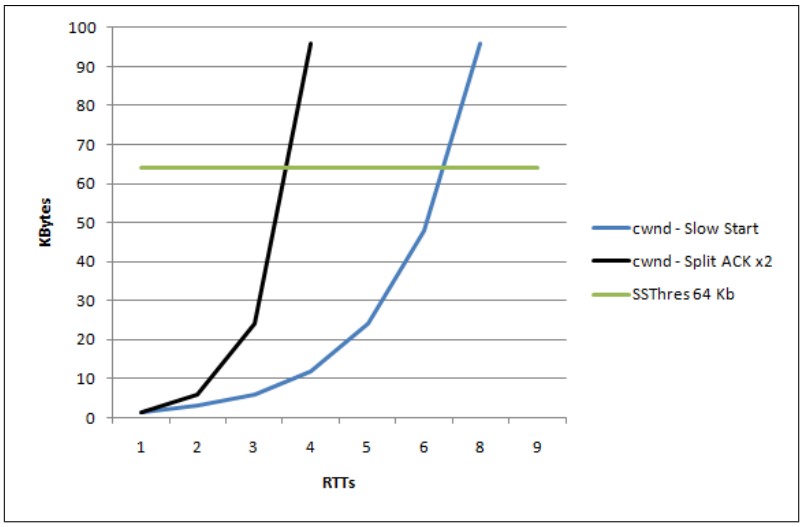


Figure 2.7: Theoretical increase of cwnd with Slow Start and with SplitAck. MSS = 512 Bytes

To illustrate this theoretical performance increase, figure 2.7 shows the theoretical speed of Slow Start and Slow Start with split ACK. Theoretically a client would need fewer RTTs to transfer the same amount of data as the standard Slow Start, thereby decreasing the download time (and connection time).

File Size	SS	CA	SS + CS
15 Kb	Improves	no-op	Improves
150 Kb	Improves	no-op	Improves
1.5 Mb	Mostly degrades	OK	Mostly degrades

Table 2.1: Results from the simulated study of TCP ACK divisions [9]

2.4.2 Simulated Study of TCP ACK Division

A more recent study of split ACK (i.e. ACK Division) was done to see if split ACK was always beneficial in any scenario (i.e. does greed always pay off?). The study is a part of a PhD thesis [9]. The study simulated a network with 8 computers, where half the computers were senders, and the other half were receivers. The computers were set up to accept split ACKs by default. They measured three different files that were sent over a TCP connection. The file sizes were 15 Kb,150 Kb, and 1.5 Mb.

The results were split up into nine different cases. The three different files split up in three different scenarios, Slow Start (SS), congestion avoidance (CA), and Slow Start plus congestion avoidance (SS+CA). Table 2.1 shows the results of the study.

As we can see from the results, the split ACK approach improved the TCP connection in the Slow Start for the small files. Split ACK also improved in the SS+CS phase, for the small files. With the larger file, the results showed that the split ACK approach triggered congestion and got a decrease in performance.

This was a simulated test to see how split ACK would preform. Our goal in this thesis is also to find out how split ACKs work in the Slow Start phase of TCP, but in a real world scenario. Based on the results of the study, the split ACK approach could be at least sometimes an efficient method to improve the Slow Start phase.

2.4.3 Appropriate Byte Counting

Appropriate Byte Counting (ABC) is defined in RFC3465 [19] as an experimental algorithm. RFC3465 outlines a proposal of how the ABC algorithm could increase the cwnd in TCP. The proposal states that, by utilizing the ABC algorithm for increasing cwnd in TCP, the ABC algorithm could increase performance and security in TCP. The older RFC2581 [17] increased the cwnd by counting the number of arriving of ACKs, and increased the cwnd with one full segment size (MSS) per ACK in Slow Start. The problem with this approach is a misbehaving receiver. A misbehaving receiver could ACK a single packet with two ACKs – because of fragmentation, where a packet got split in two parts, or if a client is utilizing split ACKs. In a scenario where the incoming ACKs only acknowledge a small amount of data, the sender would increase the cwnd faster than it should, based on the actual amount of data that was acknowledged. This is because the RFC2581 rules would allow to increase the cwnd with a full segment even if it the ACK only acknowledges a small amount of data. This could allow the receiver to trick the sender into increasing the cwnd too fast, and therefore send out more data than intended by the original algorithm.

The ABC algorithm aims to solve this by appropriately counting the bytes that are acknowledged in each ACK. Even if a client generates a large amount of ACKs per segment, the ABC algorithm will not increase the cwnd by more than the one segment that was intended.

This part of the algorithm is recommended in the new TCP congestion control RFC5681 [4], and is utilized when the TCP algorithm is increasing the cwnd in the Slow Start phase. The RFC5681 states that:

> "..we RECOMMEND that TCP implementations increase cwnd, per:
>
> cwnd += min (N, SMSS)
>
> where N is the number of previously unacknowledged bytes acknowledged in the incoming ACK. This adjustment is part of Appropriate Byte Counting and provides robustness against misbehaving receivers that may attempt to induce a sender to artificially inflate cwnd using

Algorithm	Current cwnd	ACKs	Bytes per ACK	cwnd inc.	New cwnd
ABC	1000 bytes	2	730	cwnd + (730 * 2)	2460 bytes
Old Slow Start	1000 bytes	2	730	cwnd + MSS + MSS	3920 bytes

Table 2.2: Illustrates how the cwnd could potentially grow, MSS = 1460 bytes

a mechanism known as “ACK Division”

As mentioned in the last section, the older conventional way for increasing the cwnd is to count the number of ACKs that the server received per RTT and increase the cwnd with a factor X of the number of ACKs. As mentioned in RFC5681 [4] the factor X can change depending on the algorithm that is in use (i.e Slow Start or congestion avoidance).

The ABC algorithm changes the way TCP increases the cwnd in Slow Start and congestion avoidance. The ABC algorithm still uses the same algorithms, but instead of increasing the cwnd with number of ACKs, it would increase cwnd based on the number of bytes that were acknowledged.

The RFC states that the ABC algorithm increases the cwnd with the number of unacknowledged bytes for each incoming acknowledgment, provided that the increase is less than or equal to a threshold L. The RFC states that the threshold L could be set to 2 * SMSS bytes, but should not be set to L > 2 * SMSS bytes. So instead of increasing cwnd with 1 * SMSS per ACK, as documented in the older RFC2581 [17], the ABC algorithm increases the cwnd with all of the unacknowledged bytes, as long as the unacknowledged bytes is less than the predefined threshold L.

In table 2.2 we can see the difference between the ABC approach and old approach in slow-start. When we get multiple ACKs per segment, the old Slow Start algorithm increases the cwnd with two MSS, whereas Slow Start with ABC counts the number of bytes acknowledged, and increases the cwnd with one MSS (i.e. 730 bytes * 2).

The ABC algorithm would be more aggressive than the TCP algorithm in RFC2581 [17] in the Slow Start phase with delayed ACK (where the receiver sends at most one ACK for every other packet). The difference would be that the ordinary TCP al-

gorithm would increase cwnd with 1*cwnd, even if the ACK would acknowledge two packets. The ABC algorithm would see the amount of bytes that the ACK acknowledges, and increase the cwnd with this amount. The threshold L prevents ABC from reaching a too severe rate jump in such a situation.

ABC was designed to counteract the different approaches of split ACK, where the idea was to exploit how TCP increases the cwnd. With this in mind, the intended benefits of split ACK would be nearly impossible to achieve, but since these are standards and not the law of the Internet, this is not the case. The first thing to remember, is that it takes time to get standards implemented into newer systems, and even when they are implemented, there is a chance that they are not turned on by default. It is often up to the end user to actually turn the feature on, and therefore it is still interesting to see how Split ACK would work in a real world scenario.

2.5 More on Slow Start

In this section we take a deeper look at other possible solutions to improve the start-up behavior in TCP. This section is split up the different approaches on the client/receiver side and the server/sender side. Additionally each side is split up in to proposals for the application layer, and proposals for the TCP stack.

2.5.1 Client / Receiver

Modifications to the client side are mostly done in the applications, but there are solutions to change the behavior of the start-up procedure in the TCP stack (e.g. split ACK).

Application - Internet Browsers

Internet browsers have implemented functions into the program to improve the start-up behavior of TCP. Instead of opening a single connection to a web page,

the browser opens several TCP connections to a single domain. This includes several popular web browsers like Internet Explorer 8 [20], Mozilla FireFox and Google Chrome. Browsers opens up to 6 connections [21] per domain to increase the start-up performance in TCP and therefore increases the speed of downloading the web page. The browser can, by utilizing multiple TCP connections, reduce the limitations of the Slow Start behavior in TCP.

One drawback of multiple connections is that the congestion can be unfair among the connected clients. This is because the different browsers have different approaches of how many connections they use, so this could potentially be unfair among the connected clients.

Application - Persistent Connections (HTTP 1.1)

Persisting connections [22] is a method that can handle multiple HTTP request in one TCP connection. A client that uses persistent connections can fetch multiple files in the same TCP connection, without the need to open and close the connection for each file. A browser / server that utilize a persistent connections can avoid multiple Slow Start phases for multiple files. With the use of a persistent connection, a server can pipeline requests (i.e. the client can send multiple request without the need to wait for a response), but the server needs to respond to the requests in the same order as they arrived.

The benefits of persistent connections are due to the reduction of overhead from opening and closing TCP connections. With fewer connection openings, the client (and server) gets fewer Slow Start phases, and can utilize the available bandwidth with multiple files in a single connection. Routers, clients and servers save CPU time with the reduction of opening / closing connections. The server/-client also saves memory by the decrease of TCP control blocks (e.g three-way-handshake).

Persistent connections is an application approach to improve the Slow Start phase. But not every application protocol uses persistent connections. The File transfer protocol (FTP) is an application layer protocol that utilizes TCP to transfer files.

FTP does not use a persistent connection, and therefore needs to open and close each TCP connection after a successful transfer.

2.5.2 Server / Sender

Server side modification is done in the application and in the TCP stack. There are different approaches to improve Slow Start in both ways.

Application - Server side archives

Compressed archives on the server side are a simple solution to improve (or avoid) the Slow Start phase for multiple files. With one big file compared to X single files, we can avoid the Slow Start phase X-1 times. With a compressed archive a client can easily utilize the available bandwidth and download the file faster.

To give an example, we can use the Linux kernel, linux-2.6.38.4.tar.bz. The compressed kernel archive is 73057 KB (71 MB) in size, but when extracted it is 35777 files and a total size of 503808 KB (492 MB). The total overall file size of the single files, compared to the archive, is 7 times bigger, and the average file size of the single files is about 14KB per file.

The first obvious downside is the difference in file size between the archive and the single files. But even if we ignore the different file sizes, and look at the transfer time of a single file (492 MB) compared to the transfer time of multiple files with the same overall file size (492 MB), on nonpersistent connections (e.g. FTP), there is a difference in speed. On an FTP connection, the single file download rapidly increases in speed, until the maximum available bandwidth is reached. In this scenario, the bottleneck would be the bandwidth of the connection. But when downloading N (35777) single files with the same overall size (492 MB), where the average file size is 14KB (503808 KB / 35777 files), the bottleneck is not the bandwidth. The bottleneck of the connection is the three-way-handshake and the Slow Start phase.

The first bottleneck is to open and close N TCP connections compared to the

single opening and closing with the single file. This uses extra CPU time and memory on the client side, and the server side. The second problem is that each single file (on average) never leaves the Slow Start phase, and thus the client never gets to utilize the available bandwidth.

We can see, that something as simple as a compressed archive on an FTP server gives a great performance improvement for a client, even if the overall file size is the same as with the single files.

The disadvantage to this approach is that if a single byte in the compressed archive is damaged, the whole archive is corrupted. Additionally, if a client only needs a single file in a big archive, it is very time consuming to download the whole archive just for that

TCP Stack - Jump Start

Another approach to increase the performance in Slow Start is to simply skip it. Jump Start [23] is an aggressive approach to increase the performance in the Slow Start phase, where the approach is to guess the start value for congestion avoidance instead of executing Slow Start. The overall function of the Slow Start phase is to get a reasonable start value for the congestion avoidance phase, but since Jump Start skips the Slow Start phase, it has to guess a reasonable start value. Jump Start starts off directly with the congestion avoidance phase, and depending on the difference between the guessed value and the available bandwidth, there can be a performance increase compared to ordinary Slow Start. But it can also lead to a decrease in performance. All this depends on the initial guessing value of Jump Start and the current congestion. The guessing value is determined after the three-way handshake, and the value is determined by the number of data packets that can be transmitted as the minimum of the receiver's advertised window and the amount of data queued locally for the transmission. The Jump Start phase is a short phase that finishes after the first RTT, where TCP switches to the normal congestion avoidance phase and continues the connection (if there is more data to send).

The benefits of using Jump start is the increase in performance in some specific scenarios, where the connection had more than 2 packets and no drops occurred. In other scenarios, Jump Start has shown to decrease performance when congestion occurs on a link, and the drop rate of packets increased compared to Slow Start.

Jump Start can be good in a specific scenario where most of the variables are known, to speed up Slow Start. Jump Start aims to utilize the bandwidth faster in the initial phase of a connection, but at a higher risk compared to TCPs Slow Start.

TCP Stack - Quick-Start

Quick-Start [24] is an experimental RFC to improve the Slow Start phase of TCP. Quick-Start uses custom options in the IP header in the TCP packet. This approach requires modifications to the server, client and the routers. The Quick-Start option in the IP header is set by the TCP receiver (the client), and it specifies the desired sending rate in bytes per second. The idea is to get every router along the path to accept the desired sending rate, so that the receiver can utilize the available bandwidth. When all the routers accept (or reduce) the desired sending rate, the TCP sender (the server) responds with a transport-level Quick-Start response to confirm the final sending rate. The TCP sender can then immediately send up to the desired sending rate per window.

In a scenario where a router cannot support the sending rate (e.g. no available bandwidth) the router reduces the sending rate or refuses Quick-Start all together. But if a router does not understand or refuses Quick-Start, the connection reverts back to the default congestion control behavior.

The difference between Quick-Start and Slow Start without Quick-Start is the router feedback. Slow Start probes the network to see if more and more data can be sent, but Quick-Start ask every router in the path for the maximum data rate. This is much faster, and Quick-Start can determine the threshold of the bottleneck within one RTT.

The problem with this approach is the extra header. The extra Quick-Start option in the IP header requires a firmware upgrade on every deployed router (even small routers in the common home). This is because if there is only a single router in the path that does not understand the Quick-Start option, the connection reverts back to default congestion control, thus making every previous processing step useless and a waste of time.

Another problem with the extra header is the additional processing time to determine the Quick-Start value for each new connection. With the use of Quick-Start the router would use CPU cycles on determining the Quick-Start value instead of processing data packets. This could result in a lower throughput and a decreased overall performance.

Quick-Start seems to be an approach that could improve the start-up phase in TCP. The problem is that Quick-Start is not feasible to implement in the whole Internet, where every router would need a firmware update. Even if new equipment supports Quick-Start by default, it still would take a long time to actually get an Internet path where every hop supported the Quick-Start option.

TCP Stack - Limited Slow Start

Limited Slow Start for TCP with Large Congestion Windows [25] is an experimental algorithm that limits the growth of the cwnd before it reaches the ssthresh. The proposal states that during Slow Start, a large increase in the cwnd can result in a large number of packets dropped in the network. This will result in a large number of unnecessary retransmit timeouts for a TCP connection. This could eventually result in a very small cwnd when TCP reaches the congestion avoidance phase, and it would take the congestion avoidance phase a large number of RTTs to recover its old cwnd.

Limited Slow Start introduces a new parameter called $max\_ssthresh$. During Slow Start when,

$cwnd <= max\_ssthresh$

the cwnd is increased as with the ordinary Slow Start aalogrithm, by one MSS for every arriving ACK. But when Slow Start reaches

$max\_ssthresh < cwnd <= ssthresh$

the cwnd is no longer increased by one MSS for every arriving ACK. In this phase of Slow Start, the cwnd is at most increased by $max\_ssthresh/2MSS$ per RTT. The algorithm is described in listing 2.2.

Listing 2.2: Limited Slow Start algorithem

```
1 If (cwnd <= max_ssthresh){
2     cwnd += MSS;
3 }else{
4     K= (int)(cwnd / (0,5 * max_ssthresh);
5
6 cwnd += (int)(MSS/K);
7 }
```

During Limited Slow Start the cwnd is increased by $1/K * MSS$ for each arriving ACK, compared to the original Slow Start algorithm that increases cwnd with 1 MSS per ACK. RFC3742 recommends that $max\_ssthresh$ is set to 100 MSS.

As an example from the RFC3742 – to reach a cwnd of 83,000 packets it would take 836 RTTs with the use of Limited Slow Start. With the original Slow Start algorithm it would take 16 RTTs to reach the same cwnd [25].

TCP Stack - Larger initial congestion window

Other proposals that aim to improve the Slow Start behavior in TCP have a tendency to change the algorithm of Slow Start (or even remove it), where as the initial congestion window (IW) proposal aims to increase the starting value in Slow Start. The key difference with the larger initial congestion window proposal is that this is not a modification to the algorithm, but more of a tweaking of the deployed Slow Start algorithm. Google has published an argument for increasing TCPs initial Congestion Window [21] to a higher value than the recommended values in RFC5681. The initial congestion window (IW) is by default set to 3 segments [26], but the RFC5681 allows the IW to be larger than 3 segments [4].

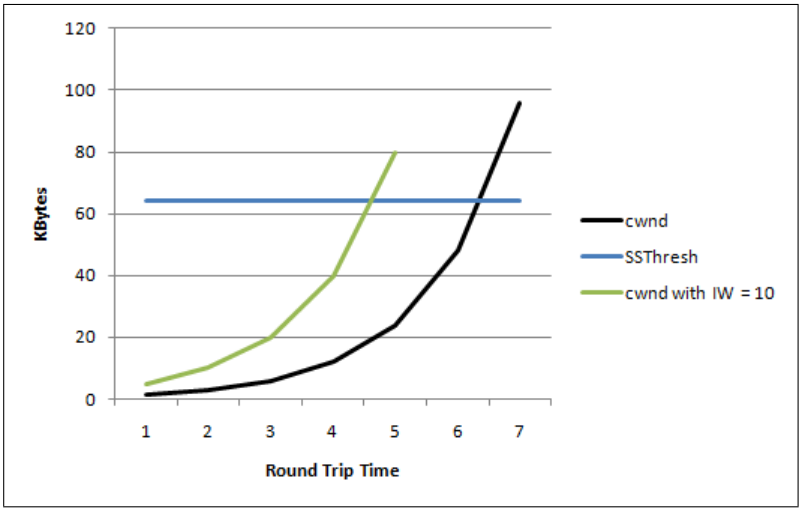


Figure 2.8: Classic Slow Start vs Slow Start with an initial congestion window of 10 segments. The Maximum Segment Size (MSS) is set to 512 bytes

The Google proposal aims to improve the Slow Start behavior in TCP, with an even larger IW, and uses the same exponential growth to hit the ssthresh. So by increasing the IW in the TCP stack, the server can kick-start the TCP Slow Start phase, and skip a few RTTs. And by increasing the initial window, Slow Start can reach the ssthresh with fewer RTTs than the original initial Slow Start value. This would allow a connection to utilize the maximum bandwidth earlier. Google proposes that the initial value should be set to 10 segments, and argues the following benefits for increasing the IW:

- Reduce Latency
- Keep up with the growth in Web page sizes
- Allow short transfers to compete fairly with bulk data traffic
- Allow faster recovery from losses

The reduced latency is the latency for a transfer completing in the Slow Start without losses on Google web search.

The negative impact could, for instance be an increase of retransmissions for slow connections, but the results showed that the increase was less than 1%. The worst case scenario discussed in [21] is with multiple concurrent TCP connections with

a large IW. Since web browsers open up to 6 concurrent connections, we get an effective IW of 80 segments or more. Even in this case, the increased IW does not cause a degradation relative to the baseline IW. The difference between ordinary Slow Start and Slow Start with a larger IW is illustrated in figure 2.8 on the previous page. As we can see, the larger IW is increasing faster than the ordinary Slow Start, and hits the ssthresh around two RTTs before the classic Slow Start.

The Google proposal was one of the major motivating factors behind this thesis, as the investigation documented in [21] only focused on web traffic that originated from the Google server (which was the server that the authors were able to change). This may not be enough: proposing a larger initial window as a standard means that every up-to-date host on the planet could eventually change its behavior, whether it is behind a fast corporate Internet connection or a behind a slow private one in a rural area. On slow connections, the larger initial window could cause problems with different applications, e.g. increased delay or loss. As we have discussed, splitting ACKs can provoke different hosts to send out a larger number of packets in slow start, hence creating an effect similar to a larger initial window, at least for one round-trip time. Our tool could therefore be used to measure the impact of a larger initial window on hosts different from Google's web server. Note that the larger initial window is already used in practice by many end hosts, as Linux has adopted the larger initial window in kernel 2.6.39 [27] [28].

Chapter 3

A Split ACK Tool

3.1 Introduction

In this chapter we will explain the design choices we made for the Split Acknowledgment tool (Split ACK Tool), and explain how the the implementation of the split ACK tool was done.

The chapter is split in three parts, where the first part explains the design choices of the implementation. The second part of this chapter explains the dependencies that are required to compile and run the split ACK tool (i.e. libraries, tools, and configurations). The last part describes the different parts of development and implementation.

3.2 Design

Our goal in this thesis was to implement and test the split ACK approach from a client point of view. This would require a modification to the operating system, either as a kernel modification, or by modifying the data that was already sent out by the kernel. To be able to generate split ACKs, we had to decide where to implement the split ACK functionality. The design choice was a question on

which layer the Split ACK function should be implemented on. The two choices to implement split ACK were in the user level, or the kernel level. In the user level the implementation would need to be run as the root user, but in the kernel level, it would need to be built into the kernel of the operating system.

3.2.1 Kernel Mode

A kernel implementation would require a custom kernel, where the kernel source code would be modified to allow split ACKs. The kernel would split the outgoing ACKs, without any interference from the user. With this custom kernel, we could create a kernel patch, so that other users could implement the split ACK functionality into their kernels.

The downside with the kernel approach could be that the implementation itself could be harder to achieve. This is because of the length of the source code for the Linux kernel. The TCP stack itself in the kernel is somewhat confusing, with limited commenting or other information. Another problem was that we did not want to split ACKs on every connection going in or out of the computer, and we did not want to lock down the split ACK functionality to a specific port or address. A second problem would be testing the implementation. For each test, the kernel would require recompiling, and then reinstalling, rebooting and finally testing. This would be the same for all modifications done to the source code. This could potentially be very time consuming, and the finished kernel could then only work on one type of system. As mentioned, the user cannot interfere with the generation of split ACKs, and this could make it harder to use the split ACKs only for a specific user and program. Some such isolation would be needed to avoid that every connection in the operating system generates split ACKs. It might otherwise be almost impossible to use e.g. an SSH connection to the computer, and would require hands-on interaction. This alone could increase the development time, because of the limited access to the testbed.

3.2.2 User Mode

A user layer approach would give the user more control with the different arguments and variables, by allowing the user to change the input to fit the specified task. Each argument could be specified at the command line e.g user id, port and the number of split ACKs. The user could also change the source code and recompile the program with the new features without the need of rebooting. This would allow the program to act more like a portable tool for testing split ACK. The split ACK tool would require a few dependencies installed, but the split ACK tool should be able do the same split ACK as a kernel implementation.

The downside with the user mode approach is that the program will have limited control of the kernel generated ACKs. The split ACK tool would need to rely on third party programs to control the kernel ACKs. Without these third party programs, the split ACK tool would be unable to do the necessary modifications to the kernel ACKs.

3.2.3 Design Conclusion

With the kernel mode approach, we could easily make a dedicated split ACK server, where all outgoing connections would be utilizing split ACK in the Slow Start phase. With the user mode approach, it would be easier to make a tool that splits the ACKs as the «man in the middle». Running two users (root and a second user), the root user could modify the ACKs of the second user and generate split ACKs.

In this thesis, we implemented a Split ACK tool that ran in user mode. The pros in the kernel approach did not outweigh the cons, like the extra development and testing time. The user mode approach gave us time to implement more advanced features (i.e. threads and speed optimization).

3.3 Libraries and Tools

The SplitAck tool was developed and tested on Ubuntu Linux 32Bit with kernel version 2.6.31-18. The split ACK tool requires to run as superuser to be able to capture the packets. The split ACK tool is portable, but requires a few dependencies to work on a new Linux system. Most of these dependencies are installed by default in the Linux kernel, except Libpcap. The Libpcap library is available in the repositories for the Linux distribution, and a simple install command is all that is needed to get it up and running. The different tool dependencies that are required to run and compile the split ACK tool are:

- Linux Kernel Version 2.6.31-18 (or higher)
- Superuser
- libpcap [29]
- IPtables [30]
- A second user account
- Wget [31] or any other third party program that utilizes TCP

3.3.1 Superuser

The split ACK tool modifies packets that leave the network card, with help of the pcap liberary. To be able to access this information from the network card, the split ACK tool has to run as the superuser. The superuser in the Linux environment is known as the root user. The downside with running a program as root is that the split ACK tool has access to potentially change anything and everything in the Operating System (OS). The reason why we need to run the split ACK tool as root is because of the pcap library. The pcap library needs to be run as root to be able to access and copy the correct data from the network card. Pcap listens for incoming and outgoing network traffic, and copyies the data that was specified by the split ACK tool. To be able to get direct access to the network card, so that the split ACK tool can listen to all traffic, pcap has to run as root.

Security is another good reason for running the split ACK tool as root. If the pcap library could be utilized on a public server without root access, the library could capture packets from other users. By restricting the pcap library to only work with the root user, a user would need the permission to run it on the given server, or the user is the administrator of the test server / client.

3.3.2 Second user - The Client

The second user is the download user in the split ACK tool, where the superuser is the hijacker of the TCP connection. The second user acts as the client and the split ACK tool is «the man middle». The split ACK tool runs on multiple accounts, to simulate the hijacking and establishing of a TCP connection, but this will be explained more thoroughly later in this chapter.

3.3.3 Packet capturing library - Libpcap

Libpcap [29] (pcap) is an open source library for Linux, BSD and Windows. The library was designed for the program language C/C++. Pcap was designed to be a high level interface to the packet capture system that can be utilized in C/C++. Pcap has a well written man page and was easy to use in the implementation. In this thesis the pcap library was used straight out of the box, without any modifications to the source code or configurations files.

Pcap can capture all packets on the network, and can even capture those packets that are destined for other hosts. We did not utilize this functionality in the thesis, and are only capturing packets generated from the same computer as the split ACK tool. Pcap has the opportunity to capture incoming packets and outgoing packets, and the split ACK tool only captures the outgoing packets, and counts the incoming data packets to measure the response from the server.

Some pcap functions that are used in this thesis are:

- pcap_open_live() – is a function to open a packet capture descriptor.

- pcap_lookupnet() – looks up the IP and the network mask.
- pcap_setfilter() – sets a filter expression and binds it to the the capture descriptor.
- pcap_next() – returns a packet.

These are only a few of the functions that are utilized in the split ACK tool, but they are the core functions. The function `Pcap_lookupnet()` verifies that you have specified a correct device and that the device is online. The split ACK tool will exit if the device is not present or if it is disconnected. This was done to make the split ACK tool more user friendly and to prevent segmentation faults. `Pcap_next()` returns a packet that has been filtered out and delivers it as a char array. The packet is supplied with complete MAC, IP and TCP headers that later can be modified or removed. The function `pcap_setfilter()` is a very strong filter command expression. It will bind the filter to the capture descriptor and will return the packets that are specified in the filter and ignore everything else. By utilizing the filter expression, a client can connect with an SSH connection, or multiple users could use the same system, without interfering with the packet capturing. In a given scenario the filter expression could be defined as in listing 3.1.

Listing 3.1: Pcap Filter Expression

```
1 char filter_exp[] = "(tcp[tcpflags]&tcp-ack!=0)and(ip[8]<=1)"
```

In listing 3.1 the filter expression returned only TCP acknowledgment packets with a TTL value of less than or equal to one. Later in this chapter we will explain the reasons for choosing the different values.

3.3.4 IPtables

IPtables [30] is the default firewall in Linux, and it is turned off by default. IPtables is a command-line based program, and works with a set of rules that the user specifies. IPtables works on the application layer, so it does not stop packets

from entering the network card, but it does stop outgoing packets from going to the lower levels of the TCP stack. The split ACK tool modifies the IPtables rules on the run, and does not interfere with the running root user. The split ACK tool also double checks that any given rule was removed after completion of a test. In a given scenario the IPtables rule could be defined as in listing 3.2.

Listing 3.2: IPTables Rule

```
1 iptables -t mangle -A OUTPUT -m owner --uid-owner 2000 -p tcp --dport 80 --tcp-↩
      flags ALL ACK -j TTL --ttl-set 1
```

Listing 3.2 shows an add-rule command for outgoing traffic. The rule only applies for the user with User ID (UID) 2000 and should only affect a TCP connection on port 80. The action of the rule was to modify the TTL value in the IP header. The value was set to 1 on the IP header and the packet was then forwarded out on the link. This modification was critical to the packet, because the first router would take action and drop the packet. The main goal was to get the packet off the network card, so that pcap could capture a copy, and return the packet to the split ACK tool.

3.3.5 Wget

The program Wget [31] was utilized as the default third party TCP program in the split ACK tool. Wget is a command-line web browser that is installed by default in most Linux distributions, making it the perfect program for easy tests with the Split ACK tool. Note that any third party program that utilizes TCP to communicate could be combined with the split ACK tool. Wget has a simple command line interface, and is reliable and easy to combine with the split ACK tool.

3.4 Implementation

The development of the split ACK tool (from now on referred to as the tool) was done in Ubuntu 2.6.31-18. The source code was written in C with support from the pcap library. The basics of pcap were explained in section 3.3.3 on page 35 and will be further explained in this section.

The goal of the implementation was to see if we could use the Split Acknowledgments to achieve benefits for a client in the TCP Slow Start phase. Split ACK was mentioned before in section 2.4 on page 15, but to be able to test this method in a real world scenario, we needed a tool for collecting data and for probing different hosts. The split ACK tool was developed for a real world scenario, where the goal of the split ACK tool was to use an active TCP connection and split the outgoing ACKs. The tool was designed for splitting acks in the Slow Start phase, and was not intended to be used in Congestion Avoidance. Such use of the split ACK tool could potentially trigger heavy congestion, and this could severely damage the connection and give the user a very low quality of service. The complete split ACK procedure is done in multiple steps, and was divided into capturing, forwarding, blocking and modifying ACKs.

3.4.1 Basic use of Pcap

To understand how the tool works, we need to understand the basics of the pcap library. The pcap library can only capture packets on the link layer - in other words, in or outgoing packets on the network card. The first step in the development of the tool was to actually capture a TCP packet. The listing 3.3 illustrates the code for a function that is capturing packets.

Listing 3.3: Capture of a TCP packet

```
1 #include <pcap.h>
2 #include <errno.h>
3
4 void CapturerPacket(){
5 char *dev = eth0;                  /* The Capture device */
6 char *filter_exp = "tcp port 80"; /* The filter expression */
```

```
7  char errbuf[PCAP_ERRBUF_SIZE];
8  pcap_t *descr;                    /* pcap descriptor */
9  const u_char *packet;             /* The captured packet */
10 struct pcap_pkthdr hdr;            /* The captured packet header */
11 struct bpf_program fp;             /* The compiled filter expression */
12
13 bpf_u_int32 mask;                  /* The netmask of our device */
14 bpf_u_int32 net;                   /* The IP of our  device */
15
16 pcap_lookupnet(dev, &net, &mask, errbuf);
17 descr = pcap_open_live(dev,BUFSIZE,5,-1,errbuf);
18 pcap_compile(descr, &fp, filter_exp, 0, net);
19 pcap_setfilter(descr, &fp);
20
21 packet = pcap_next(descr,&hdr);
22 if(packet != NULL){
23     printf(``Got a TCP packet on port 80 from device eth0\n'');
24 }
25
26 return 0;
27 }
```

The filter expression is given as a string on line 6. `Pcap_lookupnet` is the function that looks up the IP address and the network mask. Next `pcap_open_live` starts the packet capturing on a device (i.e. eth0, wlan0), and the filter expression is only allowing pcap to capture TCP packets on port 80. The final step is the `pcap_next`, which returns a copy of a packet with full network, IP and TCP headers. To capture other packets than TCP on port 80, we only need to alter the filter expression, and `pcap_next` will return the designated packets (i.e. if there is something to capture). Listing 3.3 on the preceding page is a code example of how we can use the pcap library in C, and how we were able to capture specific packets just by altering the filter expression.

3.4.2 Capturing and forwarding of ACKs

The first basic design step of the tool was to capture outgoing TCP ACKs from our client, and forward the ACKs to the host. The first test was done between two local computers to verify that the forwarded ACKs did arrive. We saw that the connection was kept alive, but when we forwarded the ACKs, we did not know if

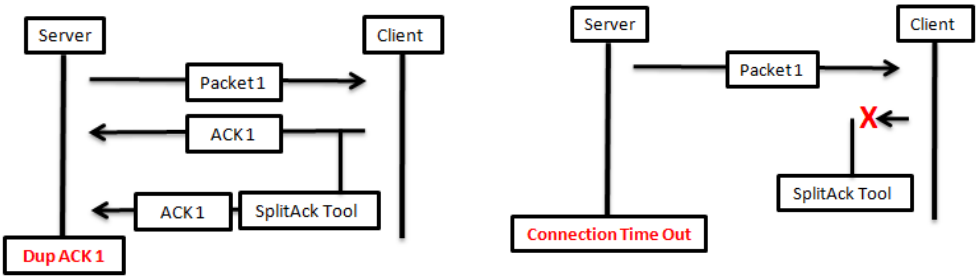
it was the forwarded or the original ACKs that kept the connection running.

The first obvious problem with this implementation was that the original ACKs were not stopped from the client, and the host detected duplicated ACKs. And our forwarded ACKs were dropped by the host, and ignored. This would not be a problem if the generated ACKs from the tool arrived first, but because of the extra delay between the copying and forwarding of the ACKs, this was not the case. As seen in figure 3.1(a) on the facing page we get a duplicated ACK when the tool copies and forwards the ACK, and at the same time the original ACK is allowed to continue.

The duplication problem indicated that the solution was to block the original outgoing ACKs. This would make it easier to test if the forwarded ACKs worked, because if the host dropped the forwarded ACKs, the connection would die, and if it did work the connection would continue as normal. The next approach was to let the tool do the copying and forwarding of the ACKs, and drop the ACKs in the application layer, thus keeping the original ACKs from arriving at the host.

The problem with this approach was that the original outgoing ACKs got blocked in the application layer by IPtables, and the tool was designed to copy and forward ACKs going off the wire on the link layer (i.e not the application layer). This was because the tool was implemented with the pcap library, and since pcap only copies packets that either enter or leave the network card, we never got any ACKs to forward. So with the ACKs blocked in the application layer, the tool did not work, and eventually the connection died. We have illustrated this problem in figure 3.1(b) on the next page. We needed to find a way to block the original ACKs, but at the same time allow the ACKs to enter the network card.

To overcome this problem, we needed to make sure that that the ACK passed through the network card, but stopped before it arrived at the host. The solution was to allow IPtables to modify the TTL value in the IP header of the original ACK. The procedure was to let IPtables take the stored TTL value in the IP header and replace it. The replaced value in the IP header allowed the ACK to pass through the network card, but to be dropped before arriving at the host. The replaced value was a low number, so that we could make sure the ACKs would be



(a) No ACK blocking -> DupACK (b) ACK Blocking -> Connection Timeout

Figure 3.1: Blocking vs Non Blocking of ACKs in IPTables

o

dropped before arriving at the host, but still be allowed to leave the network card.

Since IPtables modifies the original ACKs, and gives them a lower TTL value, the tool has an easier task filtering out the correct ACKs. The tool can easily filter out which of the ACKs to copy, modify and forward, by allowing the tool to filter for outgoing TCP ACKs on a designated TCP port, and at the same time, filter for ACKs with the lowered TTL value.

As explained in subsection 3.3.4 on page 36, the tool is dependent on IPtables to mark the correct ACKs with a new TTL value, but at the same time, the tool manages the IPtables rule. By combining the filter expression in listing 3.1 on page 36 and the expression on line 6 in listing 3.3 on page 38. It can be seen that we can capture a TCP ACK with a TTL value with less than 1.

The combined use of the tool and IPTables is illustrated in figure 3.2 where we have eliminated any duplications or connection timeouts.

To achieve this functionality, the tool has to modify the forwarded ACK. Since the forwarded ACK is a true copy of the original ACK, it shares the same TTL value. So without increasing the TTL value in the captured ACK, the duplicated ACK would be dropped in the same way as the original ACK. Therefore the tool needs to modify the TTL value, and recalculate the IP header checksum in each captured ACK. The tool has to reverse what IPtables has done to the original ACKs, and restore the TTL value in the IP header.

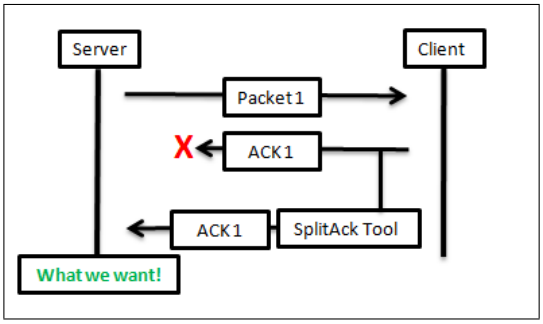


Figure 3.2: ACK with replaced TTL value

Listing 3.4: IP header modification

```
1  #include <netinet/ip.h>  /* IP header struct*/
2  #include <netinet/if_ether.h>
3  /* The packet_pointer */
4  const u_char *pcap_packet;
5
6  /* MAX_SPLITACK = 1024 copys of the original ACK is maximum */
7  struct ip *ip_header[MAX_SPLITACK];
8
9  /* The captured ACK */
10 pcap_packet = pcap_next(descr,&hdr);
11
12 /* The pointer to the IP header in the ACK is pcap_packet + Ethernet header */
13 ip_header[COPY_ORIGINAL_ACK] = (struct ip *) (pcap_packet + sizeof(struct ↩
       ether_header));
14
15 ip_header[COPY_ORIGINAL_ACK]->ip_sum = 0;  /* Set check sum to 0 */
16 ip_header[COPY_ORIGINAL_ACK]->ip_ttl = 64;  /* TTL value to 64 */
17
18 /* Recalculate check sum */
19 ip_header[COPY_ORIGINAL_ACK]->ip_sum = csum  ((unsigned short *)  packet[↩
       COPY_ORIGINAL_ACK]   , (sizeof(struct ip) >> 1);
```

This is done by accessing the headers in the ACKs, and changing each required field. In listing 3.4 `pcap_next()` returns a pointer to a captured ACK. The tool utilizes this pointer to access the different layers in the captured ACK. The process of extracting and altering the IP header from the captured ACK pointer can be divided into four steps:

- Since the pointer of the captured ACK points at the Ethernet header (i.e. the beginning of the ACK), the tool needs to skip the first layer that contains the Ethernet header. The `struct ip_header` as seen in listing 3.4 is an IP

header struct. The IP struct, in combination with the pointer to the captured ACK, is utilized to access the information in the IP header.

- The next step is to save the pointer in `ip_header struct`. The tool needs to move the current position of the pointer. The current position of the pointer is at the start of the Ethernet header, and has to be moved to the start of the IP header. This is done by adding the pcap ACK pointer and the size of the Ethernet struct, `pcap_packet + Ethernet header`. Since the size of the Ethernet header is known, we can use the function: `sizeof(Ethernet header)` to get the correct value.
- The next step is to convert the pointer from a `u_char pointer` to the IP pointer struct, `(struct ip *)`. This allows the tool to access the `ip_header` in the ACK, where the pointer points to the start of the `ip_header struct`.
- The final step is to alter the TTL value in the IP header struct. The tool utilizes the IP header struct to extract and change the TTL value in the ACK IP header. The tool alters the `ip_header[]->TTL` value, and recalculates the `ip_sum`. This allows the ACK to be forwarded by the tool without any complications or any risk of being dropped in transit.

The figure 3.3 illustrates how the tool can access the different headers and data fields in a captured TCP packet.

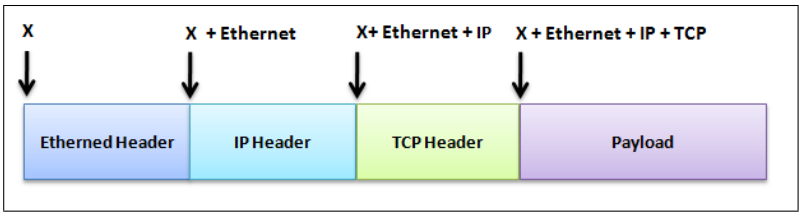


Figure 3.3: Access to the different headers in a captured TCP packet

Since the tool runs as root, the forwarded ACKs are not affected by the rules set by IPTables, because of the different UID of the download user and the root user. This allows the connection to continue without any problems. The tool now has full control over the TCP stream, and the host continues as normal with sending

more data packets.

The forwarding of the ACKs is done with `SOCK_RAW sockets`. With the raw sockets the tool does not need to establish a connection to be able to communicate. The tool only assumes that the there is an already active connection. The tool needs to alert the kernel that IP and TCP headers were already included in the packet. Without this option, the kernel would add extra headers and the host would not accept the ACKs. In listing 3.5 there is an example of how we initialized a raw socket and how we alerted the kernel with the `setsockopt`.

Listing 3.5: Alert the kernel with `setsockopt`

```
1 #include <sys/socket.h>
2
3 int s, optva l= 1;
4 socklen_t optlen = sizeof(optval);
5
6 s = socket (PF_INET, SOCK_RAW, IPPROTO_TCP);
7 if(setsockopt (s, IPPROTO_IP, IP_HDRINCL, &optval, optlen) < 0)
8       printf("Error: setsockopt() - Cannot set HDRINCL!\n");
```

The kernel adds the new Ethernet header, but leaves the IP and TCP header untouched. With the tool utilizing raw sockets to send the modified ACKs, we could add, remove or modify values in the different header layers, without the kernel interfering with the configuration. This became essential with the generation of Split ACKs.

3.4.3 Generation of Split ACKs

The tool blocks an original ACK and forwards a copy. The next step is to actually generate the split ACKs and forward them to the host. For every captured ACK, the tool makes X copies and generates Y new acknowledgment numbers from the original captured ACK. Table 3.1 on the next page illustrates how a single ACK is split up into multiple ACKs and how each split ACK gets a unique ACK number.

The number of generated split ACKs depends on the command line argument that is given to the tool. The tool can generate up to 1024 Split ACKs for every

	ACK Number	Action
Original ACK	2001	Copied and Dropped
Split ACK 1	1251	Sent as the first ACK
Split ACK 2	1501	Sent as the second ACK
Split ACK 3	1751	Sent as the third ACK
Split ACK 4	2001	Sent as the last ACK

Table 3.1: An example of a ACK, split up in 4 ACKs

captured ACK.

To be able to generate Split ACKs, the tool needs to make a copy of the captured ACK. A common way to make a copy of a memory block, is by utilizing the `memcpy` function in C. The tool uses `memcpy` to copy the IP header and everything after the IP header, including the TCP header and the payload. The Ethernet header is by default skipped in the tool, as explained in the last section. Note that the duplications of the original ACK are done after the IP header modifications are made. A simple example of using the memcpy function is shown in listing 3.6.

Listing 3.6: Copy a memory block

```
1  int i, number_of_splitacks;
2  int pckLength = sizeof(packet[ORIGINAL_PACKET]);
3
4  for(i=1; i < num_of_splitacks; i++){
5     packet[i] = malloc(pckLength); /* Memory allocate the packet buffer */
6     memset(packet[i],0, pckLength); /* Zero-out the buffer*/
7
8     memcpy(packet[i],packet[ORIGINAL_PACKET] , pckLength); /* Coppy the ACK */
9     ip_header[i] = (struct ip *) packet[i]; /* Set the IP header pointer*/
10    tcp_header[i] = (struct tcphdr *) (packet[i]+ sizeof(struct ip));/* Set the ↩
          TCP header pointer*/
11 }
```

With the split ACKs copied, the tool needs to modify the TCP header in each split ACK. Without any modification to the header, the ACKs would only be copies of the original ACK, and be detected as duplicates. So, to generate true split ACKs, each split ACK would require a unique acknowledgment number in the TCP header. This ensures that each Split ACK has a unique TCP header, but at the same time maintains the IP header from the original ACK. In each split ACK

header, the tool changes the acknowledgment number and checksum.

The generation of Split ACK acknowledgment numbers, is shown by the algorithm in listing 3.7 and in table 3.1 on the preceding page. The algorithm keeps track of the last acknowledgment number and the current acknowledgment number. To generate new acknowledgment numbers, the tool utilizes the differential value between the current `(current_ack)` and previous `(prev_ack)` acknowledgment number. The tool then divides the differential value by the number of Split ACKs `(number_of_split_acks)`. The divided value is multiplied with the current Split ACK index `( i )`, and the value is added to the last acknowledgment number `(prev_ack + calculated value)`, thus creating an unique acknowledgment number for the current Split ACK `(tcp_header(j)->ack_seq)`.

Listing 3.7: Generation of Split ACKs

```
1  #define ORIGINAL_ACK 0
2  int prev_ack;  /*  The previous acknowledgment number */
3  int current_ack;  /*  Current acknowledgment number */
4  int number_of_split_acks; /* From 1 to N, where N =< 1024 and 1 is no Split ↩
       ACKs */
5  int i, j=0;
6
7  for( i = number_of_split_acks ;  i > ORIGINAL_ACK  ;    --i ) {
8     j++;
9    /* Index 0  holds the original ACK with the highest ACK. And index 1 to N, */
10   /* holds the second highest ACK number down to index N with the lowest ACK ↩
         number */
11    tcp_header[j]->ack_seq = prev_ack +(((current_ack- last_ack) / ↩
          number_of_split_acks) * i ));
12 }
13 prev_ack = current_ack;
```

3.4.4 Finalizing Split ACKs

The final step in the tool is to send the split ACKs over the raw socket. The split ACKs now have a unique TCP header (i.e. unique acknowledgment numbers), and therefore need a recalculated TCP checksum. The calculation of the new checksum is done in the tool. Each TCP checksum is calculated for all of the split ACKs and for the forwarded ACKs. Listing 3.8 on the next page follows

the RFC793 [12] for generating a checksum, and illustrates how we did the TCP checksum calculation in the tool.

Listing 3.8: TCP checksum calculation

```
1  /* The TCP sum calculation */
2  unsigned short tcp_csum(u16 len_tcp,u16 *psuedo_hdr,u16 *tcp_header)  {
3  u16 prot_tcp= IPPROTO_TCP, count,i,temp;
4  u32 sum =0;
5
6  /* We make 16 bit words out of every two adjacent 8 bit words and
7   * calculate the sum of all 16 bit words */
8  /* First we add the pseudo_header */
9  count = sizeof(struct pseudo_header);
10 while(count > 1){
11   sum   +=  *((u16 *)psuedo_hdr)++;
12   count -= 2;
13 }
14 if(count > 0) sum   += *(uint8_t*)psuedo_hdr;
15
16 /* Then we add the TCP header with data */
17 count = len_tcp;
18 while(count > 1){
19   sum   +=  *((u16 *)tcp_header)++;
20   count -= 2;}
21 if(count > 0) sum   += *(uint8_t*)psuedo_hdr;
22
23 /*We keep only the last 16 bits of the 32 bit calculated sum and add the ↩
       carries */
24 while (sum>>16) sum = (sum & 0xFFFF)+(sum >> 16);
25
26 /* Take the one's complement of sum */
27 sum = ~sum;
28
29 /* Return the checksum*/
30 return ((unsigned short) sum);
31 }
```

After the checksum calculation is done, the tool sends the ACK out on the raw socket. Because we uses raw sockets, the tool always assumes that there is an already active TCP connection. When the ACK arrives at the destination, the server is expecting it.

The sending mechanism is shown in listing 3.9 on the following page. The ACKs are sent off in the order of the smallest acknowledgment number to the current

acknowledgment number, so that the host/server should be triggered by the split ACK to increase the cwnd.

Listing 3.9: Sending mechanism for Split ACKs

```
1 /*Arg 1 our socket */
2 /*Arg 2 the buffer containing headers and data */
3 /*Arg 3 total length of our datagram */
4 /*Arg 4 routing flags, normally always 0 */
5 /*Arg 5 socket addr, just like in */
6
7 for(i= splitacks-1; i >= ORGINAL_PACKET; i--)
8   sendto( s, packet[i], pckLength, 0, (struct sockaddr *) &sin,sizeof (sin)) < ↩
        0)  ;
```

3.4.5 Structure

The tool is split up in three threads. Each thread is synchronized to each other to speed up the process of testing split ACKs. The main thread is the Split ACK thread, where we capture ACKs and generate split ACKs. This thread was explained in the previous sections.

The second thread is the counting thread. The counting thread counts each unique data packet, and all the duplicates. As we increase the number of split ACKs, the number of incoming packets increases as well (in case of host reacting as we expect). The counting of the incoming packets is done in the same way as the capturing of the ACKs, but with a different filter expression. The filter expression is shown in listing 3.10. The filter expression filters the packets from the correct host, and from the correct source. Without specifying each filter for each scenario, we could risk counting traffic that arrives from the wrong host. So we have to make sure that we only count the incoming data traffic from the correct host, and filter out the rest of data traffic. This also means that no other application could use the Internet on the same host while our tool is running.

Listing 3.10: Counting Filter

```
1 struct bpf_program fp; /* The compiled filter expression */
2 memset(buffer,0,sizeof(buffer));
```

```
3 sprintf(buffer,"tcp port %d and dst host %s and src host %s", port, ip, ip2);
4 char *filter_exp = buffer;
```

The third thread is the wget synchronisation thread. The tool utilizes wget as the default program (i.e. if nothing else is specified by the command line argument) to fetch files and html pages. The thread waits for the first and second thread to set up the filter expressions, and when the other threads are ready to capture / count ACKs / packets, the wget thread initializes the connection, and starts off the split ACK procedure.

3.4.6 The test script

After a successful run, the split ACK tool returns the number of data packets received, the number of retransmits and the total number of packets. This is returned to the standard out `(stdout)`. To be able to collect and save this data, we needed a script that wrapped around the split ACK tool. The SplitAckScript was written in python; it executes the split ACK tool and saves the stdout result to a data file, combined with a timestamp. The script allows the tool to be run automatically for X times, and save each result in the resulting data file.

The command line for using the tool is: sudo SplitAckScript.py X device Y Z URL, where X is the number of executions, Y is the number of split ACKs, and Z is the number of ACKs before the tool stops. The device is the network device (e.g. eth0, wlan0), and the URL is the address of the server we want to test.

3.5 Summary

The implementation was based on the design decision of making the split ACK implementation at the user level. That allows the split ACK implementation to be run as a separate tool for splitting ACKs on an already active TCP connection. With the help of IPtables and the libpcap library, the split ACK tool is able to copy, duplicate, modify, and forward the split ACKs, thus essentially hijacking

the ACKs of a TCP connection, and tricking the server to increase the cwnd. The key to capture ACKs and to stop the original ACK was to modify the TTL value in the IP header. Without this modification, an implementation of the split ACK tool would have been nearly impossible because of the original ACK arriving before the split ACKs. The TTL value allows the ACK to travel through the network card, but be dropped at the first router.

In this thesis we have limited the split ACK functionality to the Slow Start phase of TCP, but the tool is not limited to only work in the Slow Start phase, and can have a greater use than what we focused on in this thesis. It is not recommended to split ACKs with a large cwnd because of the risk of congestion.

The source code of the split ACK tool is listed in the Apendix A.3 on page v.

Chapter 4

Test Results

4.1 Introduction

The classical Slow Start starts with approximately 3 packets and doubles the cwnd for each RTT. The problem with this approach is that the initial start value, and how TCP increases the cwnd is too slow, given the connection speeds of today. We wanted to see if we could improve the Slow Start algorithm in TCP, so that the start-up phase was not a potential bottleneck.

As mentioned before in the background and related work chapter 2 and in section 2.5, there are multiple proposals and RFCs that aim to improve the start-up phase of TCP. Most of the proposals are modifications or improvements on the server side. What we proposed to do, was to improve the slow-start phase with splitting the outgoing acknowledgments on the client side, so that we could trigger a faster increase of the cwnd. This would allow the host to get more data per RTT (compared to the classical slow-start) and speed up the start-up phase of TCP.

We decided to do a real world approach by implementing a split ACK tool that splits the outgoing ACKs, so that we could see if we triggered a response at the host. The core functionality of the split ACK tool was described in chapter 3.

First, we tested different operating systems (i.e Linux, Windows, Solaris, FreeBSD)

to see how a real operating system (OS) would react to the split ACK approach. We also tested against the big Internet companies (i.e. Google, Microsoft, Apple, Ubuntu, Oracle) to get a sample from the real world, before we did the final extensive test against the top 600 most visited web sites in the world.

4.2 Setup

4.2.1 Hardware and Software

The tests were done on a desktop computer, with a wired network card, Marvell Yukon 88E8056 PCI-E Gigabit Ethernet Controller, running a 100 MBit local connection to the router / ADSL modem.

The modem was a Thomson TG585, running software version 8.2.3.10. The Internet connection was ADSL2+, also known as ITU G.992.5 annex B [32], provided by Powertech [33]. The speed of the connection was 14.4 MBit downstream and 1.45 MBit upstream, giving the connection a 10 to 1 ratio between download and upload rate.

Most of the software configuration was the same as the implementation software. The desktop computer was running Ubuntu Linux, with kernel version 2.6.31-18. The dependent pcap library was running version 4.1.1. We utilized wget 1.12 to communicate with the web servers, and all the tests were done against the HTTP protocol.

To be able to collect data and run multiple scenarios in a row, a small python script was implemented to ease the process of testing the different hosts. The python script was explained in detail in chapter 4 section 3.4.6 on page 49. The tool was not dependent on the script, but the script made it easier to run multiple test and to obtain the final results.

4.2.2 Test Parameters

The split ACK tests were performed within the second RTT of the connection (excluding connection establishment). The tool would stop all split ACKs after it has reached an expected number of ACKs. The expected number of ACKs were set to the expected size of IW of 3 ACKs [17]. We set the tool to only split the 3 first ACKs (i.e. the first RTT), but counted all the incoming data packets in the second RTT (or later). To make sure that the tool would notice retransmitted packets after a timeout, the tool waited for 3 seconds before it timed out and ended this process. Figure 4.1 illustrates how the tests were done on a TCP connection. To estimate the packet size (MSS) from a server, we divided the file size by the number of unique data packets.

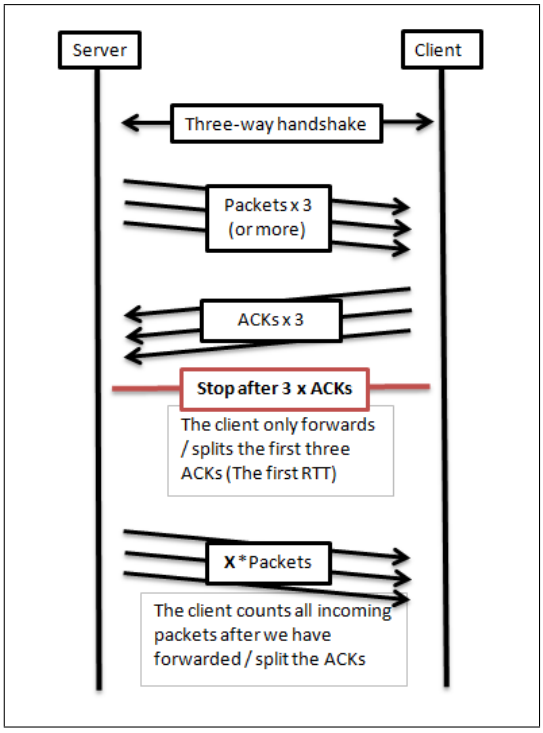


Figure 4.1: How the tests were done for the preliminary tests and the split ACK tests

If a host had a higher IW than 3 segments, the 4 to N ACKs were dropped and not forwarded, but the incoming data packets were always counted. We could then see if a host had a higher IW than the recommended IW of 3 [17], because of the

Tests	Type of test	Number of ACKs
A	Forward Original ACK	3
B	Split ACK in 2	6
C	Split ACK in 4	12
D	Split ACK in 8	24
E	Split ACK in 12	36

Table 4.1: The tests were split up in one baseline test and 4 split ACK tests

Test	Data Packets	Retransmits	Total	File Size
A	9	0	9	13 Kb
B	15	0	15	21 Kb
C	27	0-1	27-28	38 Kb
D	39	0-2	39-41	55 Kb
E	75	1-4	76-79	106 Kb

Table 4.2: Prediction of how the server would react to the tests

increased retransmissions. Note that the RFC5681 [4] does allow the IW to be higher than 3, as explained in chapter 2.

The tests were split up in two parts, with a total number of five different tests, with a 3 second pause between the tests. Table 4.1 shows how we split the different tests, and table 4.2 shows the predicted reaction to a successful split ACK approach.

The first test (cf. test A) in the table was the preliminary test. The preliminary test gave a baseline of the server behavior. If the preliminary results showed us that the connection had a high number of retransmits and low number of data packets, it could be an indication that the server was under (or the path to the host) heavy load (i.e. congestion) or – perhaps more likely that a larger IW was used and our 3 ACKs were not enough. The preliminary test gave us a baseline of the different behavior between the different servers.

The different split ACK tests were done to see if the increased number of ACKs could affect the connection, or to verify that the split ACK approach did not change the state of the connection. This could be in a scenario where a server did not respond to the split ACK, but generated more retransmits because of the

increased number of ACKs.

The highest number of split ACKs per RTT was set to 36 ACKs as a precaution to the ssthresh (i.e. 64Kb), and for a potential MSS of 1460 bytes. To make sure that the tests were done in the slow-start phase, we kept the maximum number of split ACKs to 12 ACKs per ACK, even though it would take around 46 packets to reach the ssthresh within the first RTT. Note that the tool has a potential to generate up to 1024 ACKs per ACK, as explained in chapter 4.

The determination of the OSs on the different servers was done with the tool NMap [34]. It uses HTTP signatures to determine the OS signature and gives an estimate of what kind of OS the server is running. In the scenarios where we knew what OS the server was running (e.g. local servers at University of Oslo), the NMap tool did predict the correct OS, but the prediction result was not always with 100% confidence for every server (but always >80%).

Section 4.5 on page 73 describes an extensive test of the top 600 sites in the world [35]. We did not run the full split ACK test on all the servers, but we did run test A and test C on each site. Each site was tested 10 times, 5 for each type of test, with a total amount of 6000 tests. The results were saved as the maximum number of data packets / retransmits, the minimum number of data packets / retransmits, and the average number of data packets / retransmits, with a total amount of 1800 results.

4.3 Preliminary Tests

To be able to determine any differences between the servers, we first needed preliminary baseline tests. The preliminary tests were done to test a normal connection, and see how the connection acted. We would then count the number of incoming packets and number of retransmits, and compare the results against the different servers.

The split ACK baseline test was done with the split ACK tool, so that we could control the flow of data. The tool only sent a copy of the original ACK (i.e. no

Test Nr	Data Packets	Retransmits	Total	File Size
1-5	8	1	9	8,5 Kb

Table 4.3: Preliminary test versus folk.uio.no - Solaris 10

split ACK) and counted the incoming data packets of the connection. The original ACK was still dropped, so that we maintained the control over the flow of ACKs going to the host.

The test was done with the same test parameters as the split ACK tests, where we only measured the first and second RTT of the connection. Each test was repeated five times, where we measured the number of data packets, retransmits, and the file size. The traceroute to the different servers are shown in the Appendix A.1 on page i.

4.3.1 Results - Operating Systems

In this section we present the preliminary results of the 9 different hosts that we have tested. We have tested five different operating systems (i.e Windows 2003 SP1 Server, Linux, Solaris, FreeBSD and Windows 7), and four different companies (i.e Google, Microsoft, Apple and Oracle).

Solaris 10

The first server we tested, was a web server at the University of Oslo (UiO), folk.uio.no. This particular web server was running Solaris 10 as an OS. Table 4.3 shows the preliminary results for the web server, folk.uio.no.

As we can see from table 4.3, the results did not change from test one through test five. The baseline results for Solaris 10 were very stable, and a good starting point before the split ACKs. With a solid baseline results, it is easier to see how the split ACK approach would affect the server.

Test Nr	Data Packets	Retransmits	Total	File Size
1-5	7	4	11	8,1 Kb

Table 4.4: Preliminary test versus heim.ifi.uio.no - Linux 2.6.9

Test Nr	Data Packets	Retransmits	Total	File Size
1-3	7	1	8	7,7 Kb
4-5	7	0	7	7,7 Kb
Average	7	0,6	7,6	7,7 Kb

Table 4.5: Preliminary test versus connexion.at - Windows 2003 SP1

Linux 2.6.9

The next server was another server at UiO, heim.ifi.uio.no. This server is the main web server for the Department of Informatics (IFI), and at the time of the test, the server was running Linux with a kernel version 2.6.9. The preliminary results are shown in table 4.4.

The results from the Linux server showed that it had more retransmits than the Solaris server, with a smaller file size. The results were consistent, i.e. they were equal in every test. When we compare the results form the Solaris server and Linux server, we can deduce that the two servers have a different IW.

Windows Server 2003 SP1

The next server was a web server running Windows Server 2003 SP1. This web server was not located in Norway (i.e. compared to the Linux and Solaris), but in Austria. Therefore, there was a higer number of hops from the client to the server. This could potentially give the connection a greater chance of congestion. Table 4.5 shows the results, and the average result from the Windows Server 2003 SP1.

The Windows server consistently sent 7 data packets, leading to a consistent file size of 7,7 Kb. If we compare the Windows server to the Solaris server, the Solaris server had a 10% higher transfer rate, considering the file sizes and the number

Test Nr	Data Packets	Retransmits	Total	File Size
1-4	7	0	8	5,7 Kb
5	7	1	8	5,7 Kb
Average	7	0,2	7,2	5,7 Kb

Table 4.6: Preliminary test versus freebsd.org - FreeBSD 7.0

of packets received. But because the Windows server was not located in close proximity, the longer path to the server could affect the preliminary result.

FreeBSD 7.0

The test was against the FreeBSD web server (freebsd.org). The traceroute was somewhat unclear about the location, but we estimated that the server was located somewhere in the United States of America (USA), thus giving the FreeBSD server the longest path of the OS tests. Table 4.6 shows the result, the OS was FreeBSD 7.0. At the first glance of the preliminary results the FreeBSD server was looking similar to the Windows 2003 SP1 server, where the number of data packets was the same, but the file size dose not match. This is because the average packet size was actually 22% smaller than with the Windows 2003 SP1 server. This would indicate that the FreeBSD server, or a server in the path, had a smaller MSS compared to the path of the Windows 2003 SP1 Server.

Windows 7 SP1

The Windows 7 preliminary result was obtained with a client side OS, with a web server application. The Windows 7 computer was on a similar ADSL2+ from the same ISP [33] as the client. The connection had limited bandwidth (i.e 1 MBit), compared to the other servers we tested. The limited bandwidth should not cause any negative results, because all the tests were done in the slow-start phase of TCP. The Windows 7 tests had the shortest path of all the tests. The preliminary results are shown in table 4.7 on the facing page.

When we compare the results of the Windows versions, we can see that the Server

Test Nr	Data Packets	Retransmits	Total	File Size
1-5	6	2	8	5,7 Kb

Table 4.7: Preliminary test versus s1010-0002.dsl.start.no - Windows 7 SP1

2003 has fewer retransmits and a larger number of data packets. Another difference was in the file size, where the Server 2003 had an average file size of 7,7 Kb, and where Windows 7 had a average file size of 5,7 Kb, almost 35% less in size.

The file size of Windows 7 is equal to the file size of the FreeBSD server, but Windows 7 did not share the same number of data packets. Our results indicate that Windows 7 has the smallest IW of the tested OSs.

OS results

The OSs had some differences in amongst them, and it was hard to tell if it was the configuration or the path to the server that affected the results. The Linux server and Solaris server were on the same path, and shared the same host at UiO, yet there was a clear difference between the Linux server and the Solaris server. To get a clear understanding of the differences between the OSs, we can take a look at figure 4.2 and figure 4.3 on the following page. Even if the difference was only one or two packets more or less, the servers started off from different baselines in the slow-start phase.

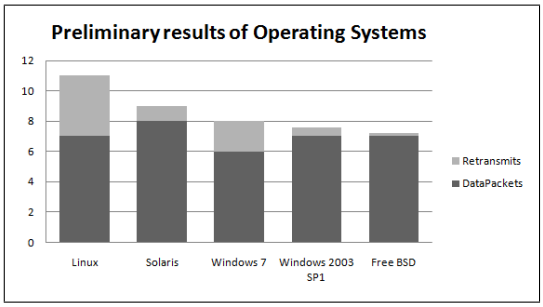


Figure 4.2: Difference between the Operating Systems

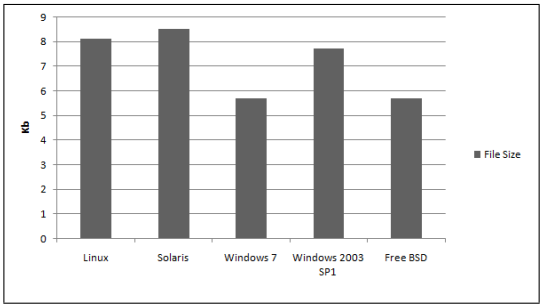


Figure 4.3: Difference between the Operating Systems - File Size

4.3.2 Results - Commercial Companies

The idea behind testing different Commercial Companies (CC) was that they partly control how the different standards are used today. As mention in chapter 2, Google proposes in an article to increase the IW to 10 packets, as opposed to the ordinary 3 packets. It would be expected that Google has already increased the IW on their servers, and that Microsoft have possibly followed in the same footsteps as Google [36].

For the different CC, it is all about the technology that is best for their business. That is why certain businesses wish to change the standards (e.g higher IW) of the Internet. The Internet standards are not the law (i.e. there is no Internet police), they are guidelines, so in theory a CC could do what ever they want on their own systems. What we wanted to investigate, was how the different CC reacted to Split ACKs, compared to the ordinary OSs. Is the Linux version running at Apple reacting in the same manner as the Linux version running at Oracle, and are Microsoft running the same configuration as any ordinary Windows 2003 SP1 Server?

One difference between testing OSs and CCs was in network traffic. Higher network traffic could potentially affect the preliminary and the split ACK results. The CCs had much higher traffic, compared to the OS servers, and might have been more vulnerable to congestion. This could potentially obscure the the test results.

Test Nr	Data Packets	Retransmits	Total	File Size
1-5	9	4	13	11 Kb

Table 4.8: Preliminary test versus Apple.com - Linux 2.6.9

Apple - Linux 2.6.9

Apple.com was running Linux with a kernel version 2.6.9 on their web server and the server was located in the USA. The preliminary results are shown in table 4.8. The results were similar to the Linux server at UiO, with the same stable result, but had more incoming data packets and the same number of retransmits. Because of the increased number of data packets, we measured a higher file size compared to the UiO Linux server. The results were equal in every tests, thus indicating a stable server and a stable path.

Microsoft - Windows 2003 SP3

Microsoft.com was running a Windows Server 2003 SP3 at the time of the tests. We expected the server to run Windows Server 2008, but the NMap tool determined that the server was running Windows Server 2003 SP3. The traceroute indicated that this server was also located in the USA. Table 4.9 on the following page shows the results of the preliminary test versus the Microsoft server. As seen in the table, the results were very unstable. To make sure that this was not a random situation, we tested against the Microsoft server several more times to verify the results, and in every test (on separate days) we got the same unstable behavior.

The Microsoft server seemed to have an extremely large IW (cf. test 1), as it yielded a very large file size. The average result showed that the server had about double the data rate than the other server tests, but in every test the retransmission rate was the same. It was not possible for us to determine if it was the server that was under heavy load (i.e congestion), or if it was the path that caused the unstable data flow. What we could see, was that the Microsoft server had the most aggressive approach in slow-start of all the tested servers.

Test Nr	Data Packets	Retransmits	Total	File Size
1	36	2	38	50 Kb
2	29	2	31	40 Kb
3	22	2	24	30 Kb
4	9	2	11	11 Kb
5	27	2	29	37 Kb
Average	24,6	2	26,2	33,6 Kb

Table 4.9: Preliminary test versus www.Microsoft.com - Windows 2003 SP3

Test Nr	Data Packets	Retransmits	Total	File Size
1-2	16	3	19	17 Kb
3	15	4	19	16 Kb
4	12	5	17	13 Kb
5	14	2	16	13 Kb
Average	14,6	3,4	18	15,2 Kb

Table 4.10: Preliminary test versus Google.com - Open BSD 4.3

Google - OpenBSD 4.3

Google.com was running OpenBSD with kernel version 4.3. The traceroute also indicated that the Google server was located in the USA. Table 4.10 shows the results for the Google server. The server had some similarities to the Microsoft server, in that it had different results from test to test. The first two tests had a high data throughput and a relatively low number of retransmissions, indicating that the server was using a larger IW than three packets. To determine the exact size of the IW was not feasible because of the divergent results.

If we compare the Google server to the Apple server, we can see that Apple had a more restricted approach in the slow-start phase, but still got about the same number of retransmits as the Google server. The average file size was about 38% higher with the Google server, compared to the Apple server, but the Apple server had a larger packet size.

Test Nr	Data Packets	Retransmits	Total	File Size
1-4	11	4	15	14 Kb
5	11	5	16	14 Kb
Average	11	4,2	15,5	14 Kb

Table 4.11: Preliminary test versus Oracle.com - Linux 2.6.9

Oracle - Linux 2.6.9

The last CC was Oracle.com. The Oracle server was running Linux with kernel version 2.6.9, same as the UiO Linux server and Apple server. We expected the results to be similar to the Apple server, since this server was also located in the USA. Table 4.11 shows the preliminary results for the Oracle server. The average number of data packets, and the average file size was the largest on the Oracle server, compared to the other Linux servers. And, as with all the other Linux servers, the results were very stable, but with a large number of retransmits. The three different Linux servers actually had the largest number of retransmits, when we compare it to the other OS and CC servers.

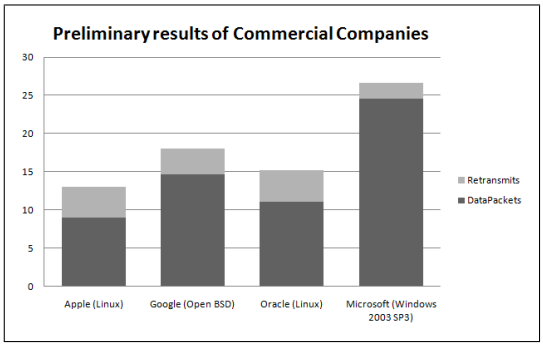


Figure 4.4: Difference between the Commercial Companies

CC results

The test of the CC was to see if there is any difference between the servers. We knew that there would be certain differences, e.g. due to traffic, the different paths, and other unknown sources, but the preliminary test results showed that the

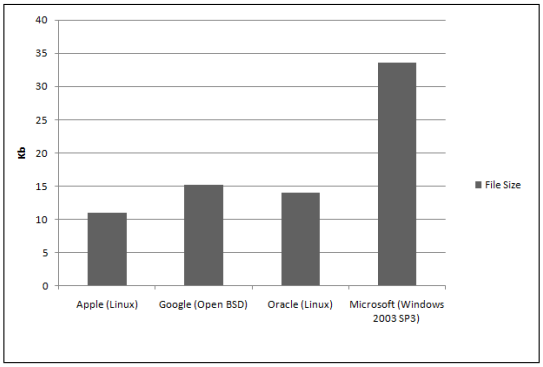


Figure 4.5: Difference between the Commercial Companies - File Size

differences were quite substantial. Figure 4.4 and figure 4.5 show the average differences between the servers. Compared to the OS tests, the differences between the CC were much higher than anticipated. The Microsoft server had three times the average file size of the Apple server, but half the number of retransmissions. The three Linux versions did all behave differently, but they all shared the same number of retransmissions.

4.4 Split ACK Tests

With a baseline for all the servers, we now have the supporting data for doing split ACK tests against the servers. This would hopefully make it easy to see how split ACKs would affect the different servers.

The tests were done with the split ACK tool, and we split the first 3 ACKs in to 2, 4, 8 and 12 ACKs per ACK, thus increasing the cwnd for the server, and increasing the data flow, with a potential risk of increasing the number of retransmits.

The results are presented in graphs (listed as figures), where the first dot in the graphs are the preliminary results, and the next four dots are the split ACK tests.

The following section is divided in the same manner as the previous section. We will present the split ACK results for the different OSs first, and then the results for the different CCs.

4.4.1 Results - Operating Systems

In this section we will present the split ACK results of the 5 different OS hosts that we have tested. These OSs as mentioned before, Windows 2003 SP1, Linux 2.6.9, Solaris 10, FreeBSD 7.0, and Windows 7 SP1.

Solaris 10

The Solaris server had a very stable baseline result, with only a few retransmissions. In figure 4.6 we have the results from the Solaris server. As seen in the result, split ACKs did increase the flow of data, without increasing the retransmission rate. The baseline results had on average one retransmission per test, and the split ACK tests did not increase this value.

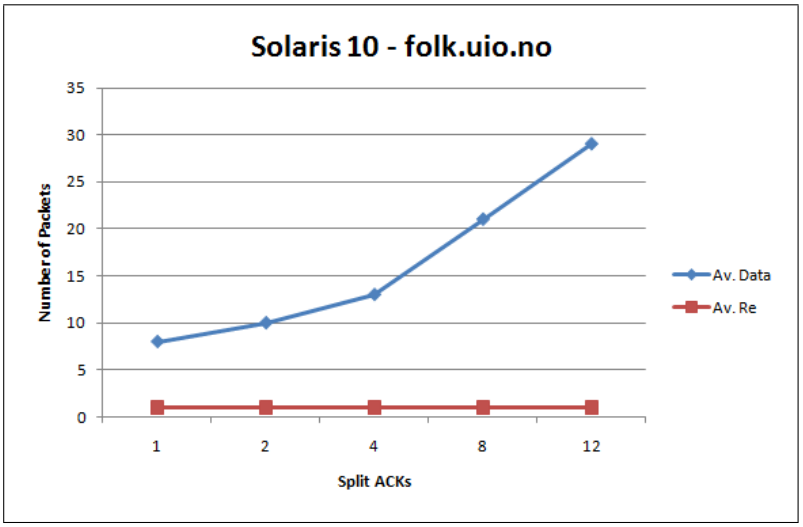


Figure 4.6: Split ACKs - Solaris 10 - folk.uio.no

As we can see from the results, when we split the ACKs in two (i.e. two ACKs per ACK), the server did not double the sending rate. In fact we had to generate 4 ACKs per ACK to achieve a doubling in the sending rate, compared to the baseline result.

One thing to note, is that Solaris 10 is not a very common OS, and the test was done on a UiO server, so the path was fairly short from the private ADSL2+ con-

nection (compared to the servers in the USA).

Linux 2.6.9

The Linux server had a stable baseline result, but with a high number of retransmissions. In figure 4.7 we have illustrated the results from the Linux server. The results clearly showed that the split ACK approach had no effect. All the results stayed the same as the baseline test. One positive effect was that the retransmission rate did not increase when we deployed the split ACK tool.

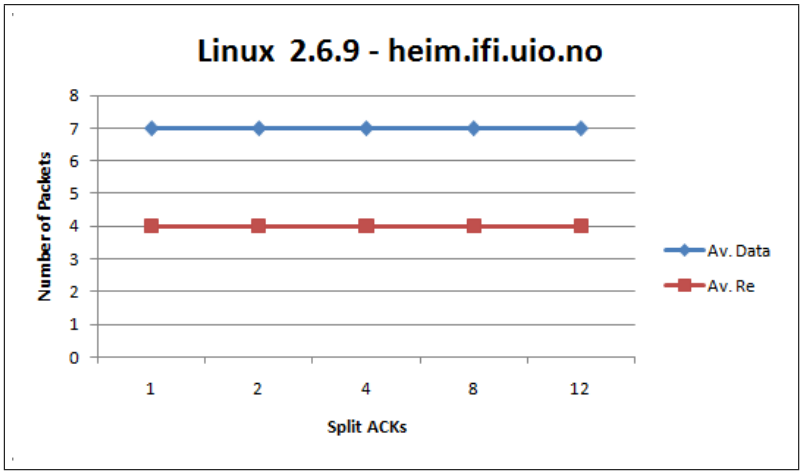


Figure 4.7: Split ACKs - Linux 2.6.9 - heim.ifi.uio.no

To see if we could trigger a different response from the server, we tried to split the ACKs into more than 12 ACKs per ACK, but when we reached 200 ACKs per ACK, the server started to send a large amount of retransmissions and the file size dropped to a few bytes. We therefore concluded that there was no beneficial use of the split ACK approach against the Linux server.

Windows 2003 SP1

The Windows 2003 server had good baseline behavior with a few variances in the results, but with a fairly low retransmission rate. The server had a longer path than

the UiO servers, but shorter than the servers in the USA. Figure 4.8 illustrates the results from the Windows 2003 server.

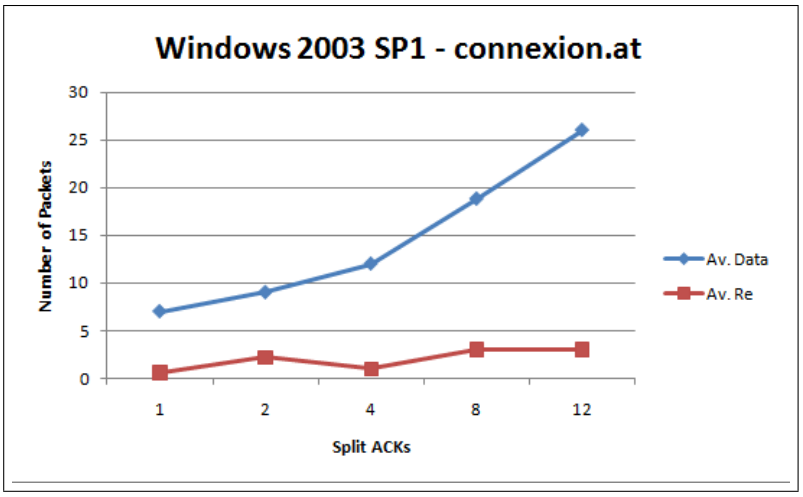


Figure 4.8: Split ACKs - Windows2003 SP1 - connexion.at

The results show that split ACK did work very well on the Windows 2003 server. The retransmission rate increased by a few packets, but dropped down to the the baseline on the second split ACK test (test C). If we compare the data packets to the number of retransmissions, the baseline has an 8.5% retransmission rate per data packet, where the last split ACK test has an 11% retransmission rate per data packet. Based on these results, we could see that there was a possible chance that the the retransmission rate increased when we split the ACKs.

In one RTT we got around 3 times more data than with the traditional slow-start approach, and only a 2.5% increase in the retransmission rate. This makes the split ACK approach against the Windows 2003 Server a very beneficial approach for a client that aims to improve the slow-start phase in TCP.

FreeBSD

The FreeBSD server had the smallest MSS of the servers. In the preliminary results the file size was smaller than the number of data packets for the FreeBSD server. In other words, the MSS was smaller than 1024 bytes. The exact average

MSS was around 880 bytes per packet in the preliminary results.

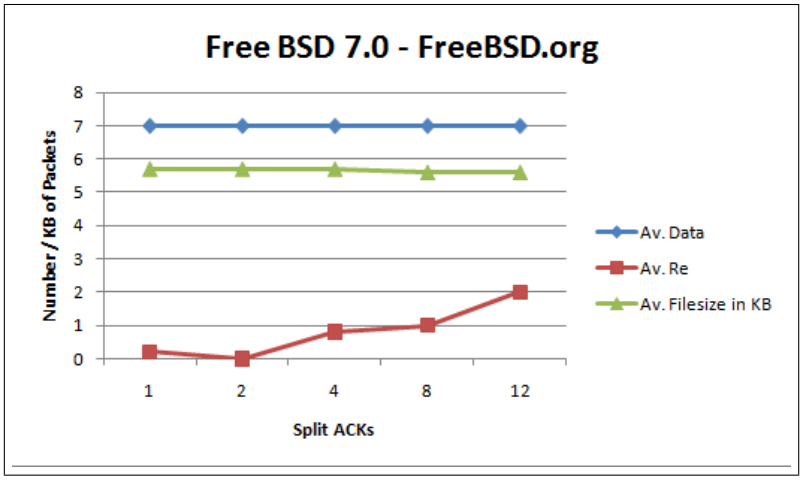


Figure 4.9: Split ACKs - FreeBSD 7.0 - FreeBSD.org

The results of the split ACK tests are shown in figure 4.9, where we have added the file size. The split ACK approach did not increase the data flow for the server. The server reacted negatively to the approach, by sending more retransmits. The number of retransmits increased in the same manner as we increased the number of split ACKs to the server, and the average file size also went down, but the number of data packets received did not go down. This would indicate that the MSS declined, either in the path to the server, or from the FreeBSD server.

Windows 7 SP1 Client

The Windows 7 SP1 client computer was the newest version of the client OS from Microsoft, so it may be expected that the split ACK approach would not necessarily work, but we did not know this for sure. As shown in figure 4.10 on the facing page the response was equal to the Linux server, where the results stayed the same. The Windows 7 SP1 client had the shortest path of all the test servers, so there was a lower probability that the path would interfere with the results, and as we can see from the results, the number of data packets and retransmits was equal in all the tests. The split ACK approach triggered no response from the Windows 7 SP1 client computer. An alternative would be to test an ordinary Windows 7,

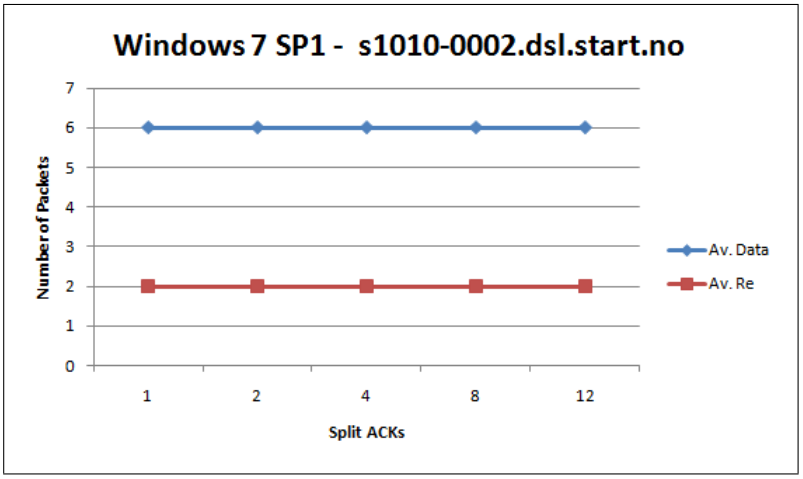


Figure 4.10: Split ACKs - Windows 7 SP1 - s1010-0002.dsl.start.no

without the service pack installed.

OS results

The split ACK approach did work in, some cases, giving the client a performance boost with increased data flow, resulting in a larger file size. The Solaris and Windows 2003 SP1 servers did both react to the split ACK approach, but not as we expected. The increased performance was lower than we first anticipated, where we thought that two split ACKs per ACK would double the cwnd and double the data flow. The results showed that we would need four ACKs per ACK to achieve the expected performance. The retransmission rate increased by a small amount on the Windows 2003 SP1 server and did not increase at all on the Solaris server, but when we compared the performance gain to the increased retransmission rate, the increased retransmission rate was negligible.

The Linux server and Windows 7 SP1 client computer had no change in behavior in the split ACK approach, where the results stayed the same in all the tests.

The FreeBSD server had the worst outcome from the split ACK approach. The packet size went down, and the retransmission rate went up. The split ACK approach would actually make the connection slower, and cause higher congestion.

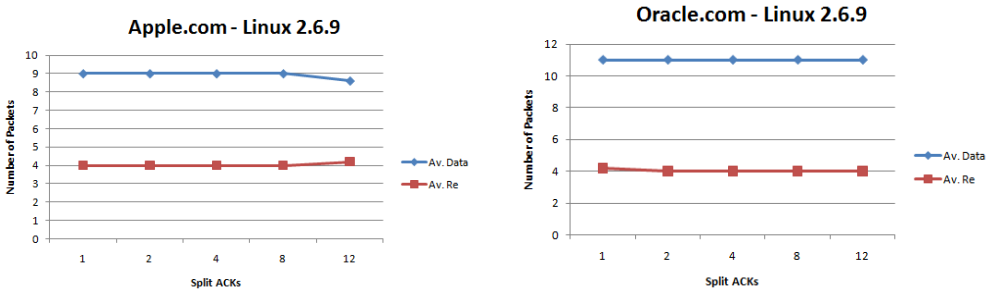
It would therefore not be recommend to use the split ACK tool against a FreeBSD server.

4.4.2 Results - Commercial Companies

With the OS split ACK result, we knew how the different OS could potentially react to the split ACK approach. What we did not know was how the CC had configured their servers. We tested the same CC as we did in the preliminary results. We tested four different CC, with two Linux 2.6.9 server, one Windows 2003 SP3 Server, and one Open BSD 4.3 server.

Apple and Orcale - Linux 2.6.9

The Apple and Oracle servers where both running Linux 2.6.9. Since in the Linux OS test, we got no reaction from the split ACK approach, we expected similar results in the CC tests with the Linux servers. In figure 4.11(a) on the next page we have the results from the Apple server, and in figure 4.11(b) on the facing page we have the results from the Oracle server. As we can see from the figures, the effect was the same as towards the UiO Linux server, where the baseline and split ACK test were equal. The Apple server did get a small decrease in the average number of data packets. The retransmission rate did also increase slightly, which could have been triggered by the split ACK approach.



(a) Split ACKs - Apple.com

(b) Split ACKs - Oracle.com

Figure 4.11: Apple and Oracle - Linux 2.6.9

The three Linux servers had different baseline results, but the reaction to the split ACK approach was very similar. This would suggest that the reaction to split ACKs was a default behavior in the Linux kernel, and not necessarily a configuration feature.

Microsoft - Windows 2003 SP3

The Microsoft server had very unstable baseline results, but had a stable retransmission rate. The results for the Microsoft server are shown in figure 4.12 on the next page. The Microsoft server was running an upgraded (latest service pack) version of the Windows 2003 server. The first split ACK test showed that the data flow decreased by an average of four packets, and increased again on the second split ACK test. This result could be due to high traffic on the server, but the last two tests showed that the data rate dropped with 10 packets below the baseline.

With the increased number of split ACKs the connection actually became more stable, but with a lower number of data packets and a smaller file size. As we showed in the last section, the Microsoft preliminary baseline had very divergent results. The split ACK approach did not increase the performance, as it did with the Solaris and Windows 2003 SP1 servers, but we measured a more stable connection, and with less divergent results compared to the baseline.

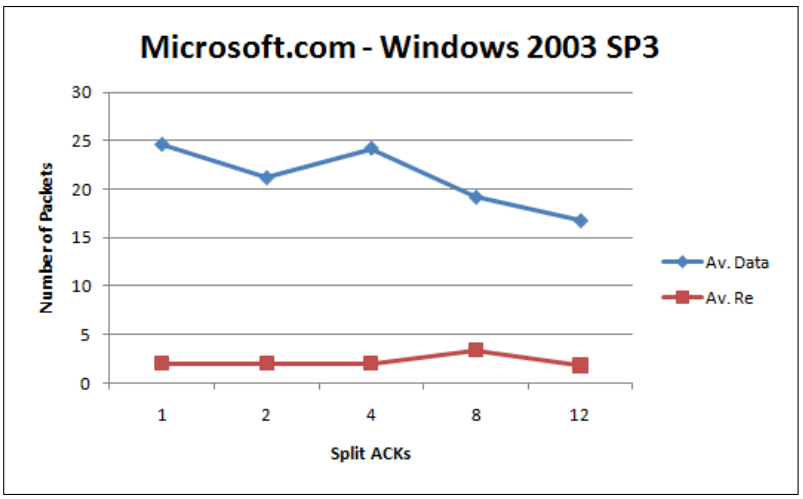


Figure 4.12: Split ACKs - Windows 2003 SP3 - Microsoft.com

Google - Open BSD 4.3

The Google server was the last of the CC servers. The baseline result was less divergent than with the Microsoft server, but not as stable as with the FreeBSD server. The Google server results are shown in figure 4.13 on the following page. As we can see, the Google server did not respond with a performance gain to the split ACK approach.

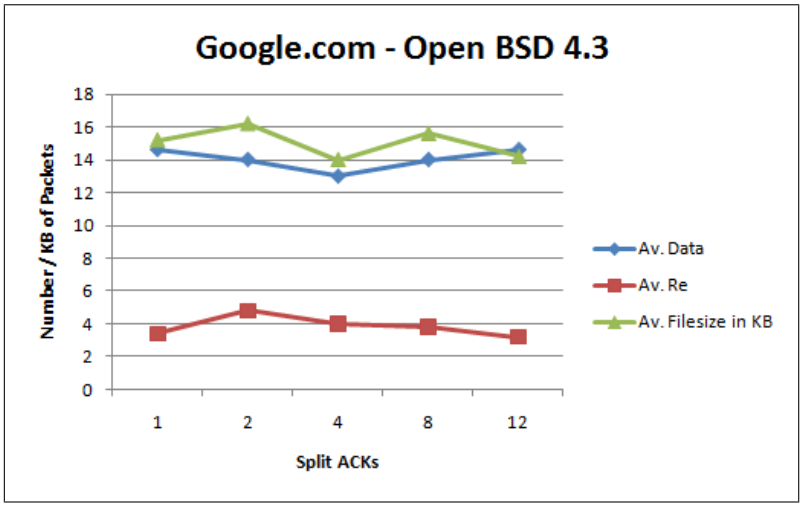


Figure 4.13: Split ACKs - Open BSD 4.3 - Google.com

The variance in the split ACK results were too inconclusive to tell whether the split ACK approach did affect the server in any way. The baseline test, and the last split ACK tests had about the same number of data packets, file size and retransmission rate, but the results were very unstable and hard to predict.

CC results

The CCs were protected against the split ACK approach, and none of the CCs showed a beneficial reaction to split ACKs. The Linux server did behave exactly as the Linux server at UiO, without any reaction. The Microsoft server did experience a performance decrease, but because of the very divergent results, it was hard to tell if it actually was the split ACK approach that caused it. The lower

performance may also have been caused by an unknown source (i.e. traffic, the path) that affected the outcome of the results on the Microsoft server.

4.5 Top 600 Web Sites in the World

Based on the results from the previous sections, we wanted to do a final extensive test against the top 600 most visited web sites in the world (list downloaded from Alexa.com on 13.07.2011). With this test, we observed how the real world reacts to the split ACK approach in TCP Slow Start. As mentioned before, we did one baseline test and one split ACK test (i.e test A and C). The results are presented in two data sets; the first data set is the number of data packets (relevant to answer: can a speed-up be attained?), and the second data set is the total number of data packets and retransmits (relevant to answer: can we provoke host to send more packets in an RTT for measurement purposes?).

4.5.1 Data Packets

The baseline results are presented in figure 4.14 on the next page. The average data rate of all the sites was around 8 - 9 packets, and the range 6 - 13 data packets covered about 83% of all the tested sites. With an IW of 3 - 4 packets, we expected to receive around 9 - 12 data packets within the first two RTTs, and the expectation correlated with the results.

The goal of the split ACK approach was to increase the number of data packets and hence the average data rate for all the sites. The results of the split ACK approach are presented in figure 4.15. As we can see from the results, the split ACK approach did not have a significant impact on Slow Start. The split ACK approach did work, because the results show a small increase of sites in the range of 10 - 16 data packets – but if the expected split ACK approach had worked on the majority of the sites, the results would be around double of the baseline results, and the peak in the figure would have been around 16 - 17 data packets. Table 4.12 on the next page shows a result with an expected reaction to split ACKs.

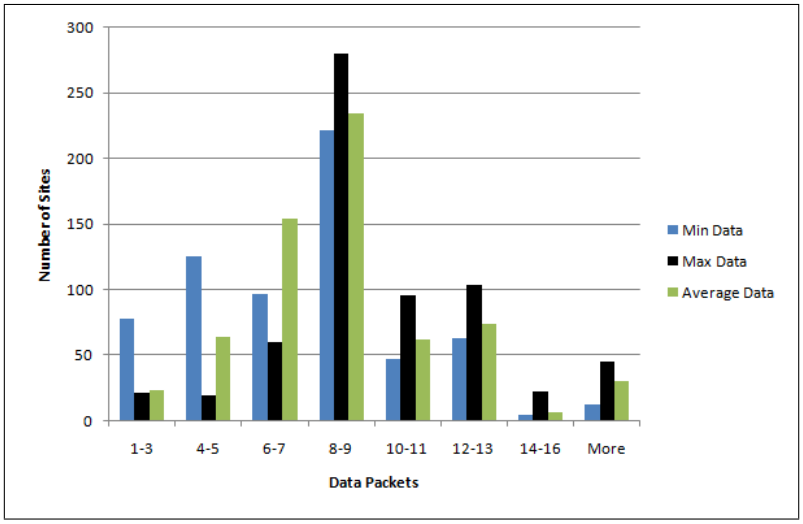


Figure 4.14: Data Packets - Baseline results

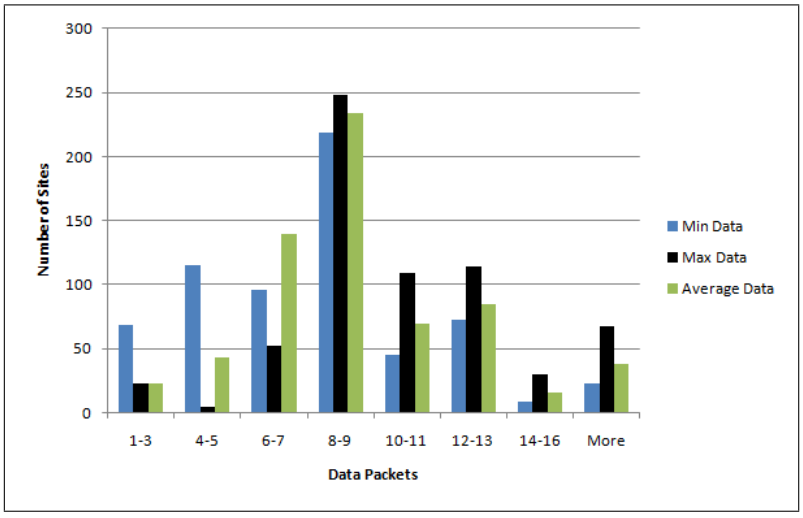


Figure 4.15: Data Packets - Split ACK results

The successful split ACK result was nearly twice as large as the baseline result. But the overall results showed that only a small number of sites reacted to the split ACK approach.

Test Type	Data Packets	Retransmits	Total
Baseline	11	2	13
Split ACK	23	4	27

Table 4.12: Average results from AOL.com

4.5.2 Total Number of Packets

In the previous section we only observed the incoming new data packets, and ignored the total number of packets (i.e data packets + retransmissions). As we saw, the split ACK approach did not improve the data rate in Slow Start for the majority of the sites, and only handful of sites did improve the data rate in TCP Slow Start. Her, we examine the total number of packets arriving at the client.

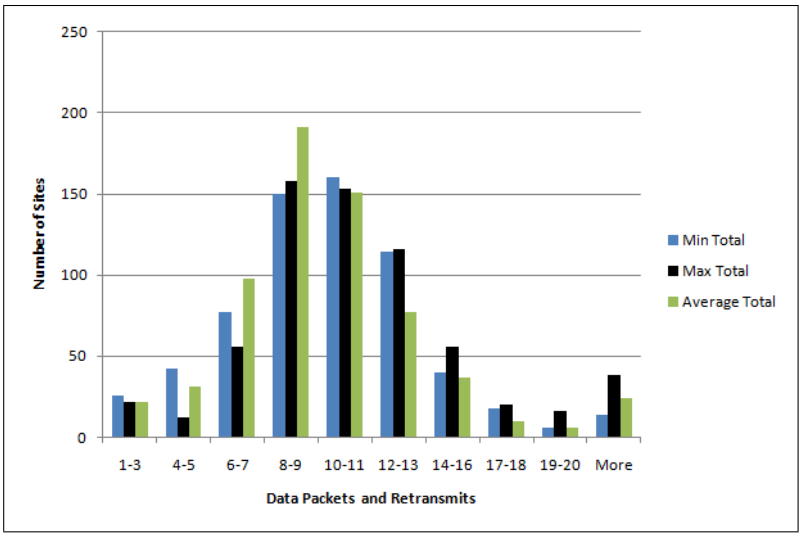


Figure 4.16: Total Number Packets - Baseline results

The baseline results are presented in figure 4.16. The results were consistent with the baseline data results, because most of the sites were in the range of 6-13 packets. The results were more spread out compared to the data results because of retransmissions.

The results of the split ACK tests are presented in figure 4.17. The graph shows very clearly that the total number of packets did increase. Based on the results from the last section, we can say that the increased traffic was caused by an higher

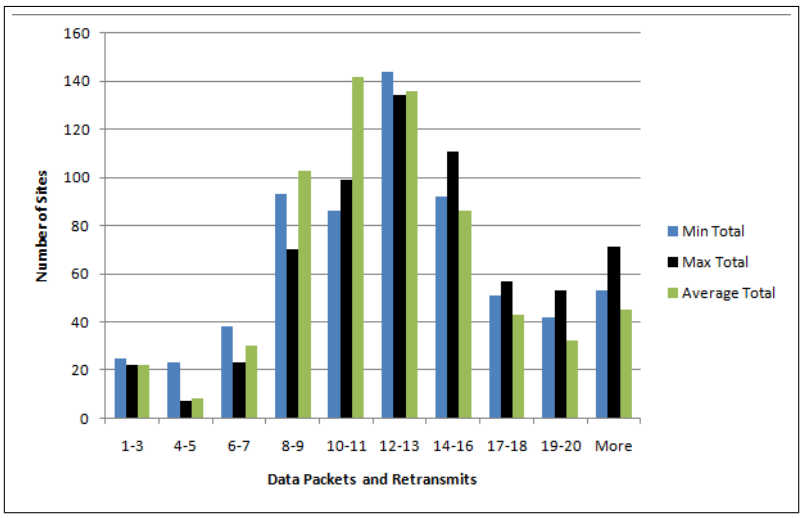


Figure 4.17: Total Number Packets - Split ACK results

retransmission rate. From the four final tests, it is reasonable to say that the split ACK approach was not very beneficial to the Slow Start phase in a real world scenario.

4.6 Summary

In this chapter we set out to test our tool on different host, to observe the reaction when we split the ACKs. We divided the results in OS tests and CC tests, where we first did a preliminary test to determine the baseline and to observe the differences between the OSs and the CCs.

The preliminary tests showed that none of the servers shared the same result. The Linux servers did share the same behavior, where they had a high retransmission rate, compared to the number of incoming data packets. This could be linked to the fact that Linux was potentially using a larger IW [27] [28]. The FreeBSD server had, as expected, the most conservative approach of all the tests. The FreeBSD server had the smallest MSS and a low number of incoming data packets. The Windows 7 client shared the same behavior as the FreeBSD server, with a low number of incoming data packets. But the difference was that the Windows 7

client had a higher MSS per packet, and would therefore yield a larger file size.

In the preliminary CC tests, the Microsoft server had an extremely high data rate. The Microsoft server peaked at 38 data packets, with only 2 retransmissions. The tests were very unstable, with divergent results. The average result in the baseline test was around 24,6 incoming data packets, even with a very unstable data rate, but the retransmission rate was very stable with only 2 retransmitted packets on average. The Google server had a higher data rate than the other servers, except for the Microsoft server, and it peaked at 16 data packets. The Google server had an unstable retransmission rate, thus lowering the number of incoming data packets in the other tests. Microsoft and Google had the highest IW of all the servers. Microsoft had an up to 5 times higher data rate than other hosts (e.g FreeBSD server) in some tests.

The successful split ACK tests were done to the Solaris server and the Windows 2003 SP1 server. The results were mainly divided in three reactions, where the first reaction was a successful split ACK, the second reaction was no reaction at all, and the third reaction was a negative reaction.

As mentioned before, the servers we tested with success with split ACKs were the Solaris server and the Windows 2003 SP1 server. The Solaris server did get the best reaction, with the largest increase of data packets, and with very low retransmissions rate. The performance gain between the baseline result and the largest split ACK test was about a 300% increase of incoming data packets. The Windows 2003 SP1 server had a similar behavior, but also had an increase in the retransmission rate. The retransmission rate only increased with 2.5%, and the data rate increased with 250% when compared to the average baseline results.

The Linux servers and the Windows 7 SP1 client shared the same behavior, with no reaction to the split ACK approach. The Windows 7 SP1 client showed no increase in performance, and the Linux server had no increase in performance regarding incoming data packets, and also had no increase in the retransmission rate.

FreeBSD reacted poorly to the split ACK approach. The server got no improvement from the split ACKs, but had a decrease in performance. The FreeBSD

server increased the retransmission rate, consistent with the increase of the number of split ACKs. The Google server and the Microsoft server had very unstable, and inconclusive results. The overall average performance went down on the Microsoft server, but the results were more stable (i.e less divergent) in the later split ACK tests. Google had less divergent results than Microsoft, but the baseline test and the last split ACK test had equal results (test E). The other split ACK tests (test B,C,D) either had a negative effect or no effect at all.

The overall results showed what we were expecting, that the different OSs and CCs, had very different baseline results, and that the reaction to the split ACK approach were unpredictable, with positive and negative effects. The newer OSs did not react to the split ACK approach, and this could indicate that the servers were following the ABC or RFC5681 method for increasing the cwnd.

The final test was to observe how the «real world» reacted to split ACKs. The tests determined if the top 600 most visited sites in the world supported split ACK in the slow-start phase. We ignored what type of OS, and where the location of the server was (i.e path), and focused on testing a large number of sites. The results were clear; split ACKs did increase the number of packets (traffic), but did not increase the number of new data packets. Instead, the retransmision rate increased for a large portion of the sites. There were cases where split ACK had extreme effects, but in most cases the split ACK approach did not increase the data rate, and was therefore inefficient for increasing the cwnd in Slow Start. We knew that split ACK could potentially improve Slow Start in TCP, but only a few OSs did support the split ACK approach. The split ACK tool could still be used as a measurement tool, to provoke host to quickly generate a large amount of traffic in Slow Start.

Chapter 5

Conclusion & Future Directions

5.1 Conclusion

The Slow Start phase in TCP is lagging behind the speed of the connections today. The performance can in some cases be limited by the Slow Start algorithm, and not necessarily by the bandwidth. Our goal in this thesis was to see if a client could improve the Slow Start phase in TCP by using split ACKs. To achieve this, we implemented a tool to split the outgoing ACKs on a host. Our goal was not to simulate a split ACK scenario in a controlled network, but rather do a real world test, because in a real world test, we have no other control of the severs, or the path to the server.

The concept of split ACK did not violate the defined algorithms of the TCP congestion control. As an example, a delayed ACK in TCP can acknowledge more than one segment per ACK, and in the same manner, split ACKs can use multiple ACKs to acknowledge a single segment. The difference between the two approaches is how the sender increases the cwnd in accordance with the received ACKs.

The latest RFC for TCP Congestion Control, RFC5681, is recommending a sender to increase the cwnd with the actual number of bytes acknowledged in the ACK (Quote 2.4.3 on page 20 [4]). This was inspired by the experimental ABC algo-

rithm that we described in chapter 2. Unfortunately some of the OSs do not follow this standard (good for split ACK), and they increase the cwnd by the number of incoming ACKs as described in RFC2581. This allows the client to send multiple ACKs per segment to increase the cwnd faster than intended by the original Slow Start algorithm. A study that we described in 2.4.2 on page 19, showed that it was possible to improve the Slow Start phase in a simulated environment with split ACKs.

The split ACK approach is one approach to improve the Slow Start phase in TCP. In last part of chapter 2 we presented alternatives that aimed to improve Slow Start. The use of persistent connections works on the application layer. The idea is to open a single TCP connection that could transfer multiple files, so that a client could reduce the number new TCP connections. Most of the TCP stack improvements were achieved on the server side, and only a handful of the algorithms are utilized today. Google proposes to increase the IW in Slow Start to 10 segments, so that they could kick-start the Slow Start phase (i.e. exponential growth with a higher start value). The other TCP stack improvements introduce new algorithms, but Google aims to improve the classical Slow Start algorithm.

With the same goal as Google, to improve the deployed Slow Start phase, the split ACK approach aims to also improve the classical Slow Start algorithm, but from a client point of view. We implemented a portable tool to test split ACKs on real world scenario. The tool was not implemented as a kernel modification. This allows the implementation to run on any system that support IPtables and the libpcap library.

The tool was written in C, in multiple threads, to maximise the performance and efficiency of splitting ACKs. The tool controls everything from connection establishment, to counting the incoming results, and generating split ACKs from the original ACK. The test of split ACKs was divided into two parts, different operating systems and commercial companies. In each scenario we had one baseline test and four split ACK tests. The OS baseline tests were done to observe differences between the servers in the start up phase of TCP. The OS split ACK tests were done to identify the different reactions to the split ACK approach. The CC baseline tests were done to see how the CCs had implemented the standards, and

if they were cheating on the default behavior in TCP. We also observed how the CCs reacted to split ACKs, and if there were any differences between the CC tests and the OS tests.

The split ACK results were divided into three reactions, where the first reaction was a successful split ACK, where the split ACK tool increased the performance. The second reaction was no reaction at all, resulting in an identical results between the baseline test and the split ACK tests. The third reaction was a negative reaction, where the split ACK tool decreased the performance of Slow Start.

The successful split ACK tests were done against a Solaris server and a Windows 2003 SP1 server, where the data rate doubled at around 4 split ACKs per ACK, and with only a small increase in the retransmission rate. The Linux servers and the Windows 7 SP1 client computer had no reaction to the split ACK approach. We speculated that the OSs had implemented the new congestion control [4]. With the new congestion control, the increased number of ACKs would not affect the server. This was because the server would only increase the cwnd with the number of bytes the split ACKs acknowledged (e.g 1 ACK with 10 Bytes or 10 ACKs with 1 Byte). The FreeBSD server experienced a decrease in performance, where the packet size dropped and the retransmission rate increased.

With the results from the OS tests and CC tests, we set out to determine the true effect of split ACK. We tested the top 600 most visited web sites in the world. The tests were simplified versions of the OS and CC tests, where we only tested the baseline and a single split ACK test (i.e Test A and C). The results clearly showed that the split ACK approach did not improve the data rate of the Slow Start phase. Most of the servers did not support split ACK, and most of the sites that reacted to the split ACK approach reacted poorly. The results showed that the traffic did increase with the split ACK approach, but the data rate of new packets did not. The increased traffic was therefore caused by retransmissions of old data packets. There were sites that reacted successfully to the split ACK approach, but compared to the number of sites that did not react, and to the number of sites that reacted poorly, we can say that the split ACK approach is not an efficient method for increasing the performance in TCP Slow Start. In some scenarios, and against some servers, it was possible to achieve a performance boost, but in the real world,

the split ACK approach was not an efficient method for improving the Slow Start phase in TCP. The tool was successful in splitting ACKs and measuring a server reaction. The tool could therefore be utilized for measurements, instead of being a tool to improve the slow start phase.

5.2 Future Directions

In this thesis we utilized the split ACK tool on web servers for testing split ACKs in TCP Slow Start. Based on the behavior in the split ACK tests, we saw that some features and behaviors could be improved. The split ACK script should be able to determine what port a web server is communicating on. This is because we experienced problems when we tested HTTPS sites, where the default port was not port 80 (TCP Port 443). The tool was no able to adapt to a port change from one site to the next. We could set the port manually, but if the script could determine the default port for communication, the testing procedure would not need interaction from the user, and require less supervision.

One of the more interesting results, was the successful split ACK against the Windows 2003 SP1 server. It might be interesting to test more windows versions to observe what the default behavior is. It also seems relevant how the client OS, Windows XP react to split ACKs, since this is still one of the most common OSs today. It would also be interesting to observe with which kernel version of Linux split ACKs did actually work, how they affected the behavior of the OS, and if it was possible to configure an OS to support split ACKs without alliterating the source code or the kernel, but by configuring the parameters of the OS (i.e. regedit and systel).

In this thesis we focused on split ACKs against web servers, but how does it work against other clients? If we deployed the split ACK tool on a peer-to-peer network (i.e. Bittorent) how could split ACKs affect the peer-to-peer network, and could we improve the speed of the transfer? Are the client OSs more receptive for the split ACK approach, or do they share the same behavior as the server OSs? These questions deserve further investigation.

Appendix A

Appendix

A.1 Traceroutes

All the tests were carried out from 14.07.2011 to 15.07.2011.

Solaris

The first server we tested was the main web server for all students and employees at the University of Oslo (UiO). The web server is running Solaris 10. The traceroute to the server is shown in the list below.

Listing A.1: Traceroute Solaris

```
1 s22-00001.dsl.no.powertech.net (77.40.134.1)          32.513ms
2 s02b02.no.powertech.net (195.159.88.82)               29.096ms
3 oslo-gw.uninett.no (193.156.90.1)                     29.377ms
4 uio-gw8.uio.no (128.39.65.18)                         29.053ms
5 mrom-gw2.uio.no (129.240.25.18)                       29.766ms
6 folk.uio.no (129.240.4.230)                           29.379ms reached
```

Linux 2.6.9

The second server was the main web server for the Informatics Department (IFI) at UIO. The traceroute to the server is shown in the list below.

Listing A.2: Traceroute Linux 2.6.9

```
1 s22-00001.dsl.no.powertech.net (77.40.134.1)          32.656ms
2 s02b02.no.powertech.net (195.159.88.82)               28.967ms
3 oslo-gw.uninett.no (193.156.90.1)                     28.819ms asymm  5
4 uio-gw8.uio.no (128.39.65.18)                         28.844ms
5 uio-gw21.uio.no (129.240.24.254)                      29.134ms
6 ifi-gw21.uio.no (129.240.24.250)                      28.343ms
7 webserver.ifi.uio.no (129.240.64.5)                   29.378ms reached
```

Windwos 2003 Sp1

The third server was a company hosting web sites, using a windows web server located in Austria. The traceroute to the server is shown in the list below.

Listing A.3: Traceroute Windows 2003 SP1

```
1 s22-00001.dsl.no.powertech.net (77.40.134.1)          32.017ms
2 s01b01.no.powertech.net (195.159.88.81)               28.887ms
3 oso-b4-geth3-0-13.telia.net (213.248.97.33)           28.695ms asymm  5
4 kbn-bb2-link.telia.net (213.155.131.132)              40.698ms asymm  6
5 hbg-bb2-link.telia.net (213.155.133.24)               47.501ms asymm  7
6 kol-b2-link.telia.net (80.91.247.249)                 55.239ms
7 xe-3-1-0.cr-nashira.cgn4.hosteurope.de (213.155.141.46)  54.334ms asymm  9
```

FreeBSD

The fourth server was the main web site for FreeBSD, running FreeBSD. The traceroute to the server is shown in the list below.

Listing A.4: Traceroute FreeBSD

```
1 s22-00001.dsl.no.powertech.net (77.40.134.1)          31.194ms
2 s02b02.no.powertech.net (195.159.88.82)               28.373ms
```

```
3  te0-1-0-5.ccr21.ham01.atlas.cogentco.com (130.117.2.177)  45.254ms
4  te0-0-0-3.mpd21.fra03.atlas.cogentco.com (130.117.49.237)  53.880ms
5  te0-4-0-0.mpd21.dca01.atlas.cogentco.com (154.54.26.101) 152.925ms
6  te0-2-0-5.mpd21.iad02.atlas.cogentco.com (154.54.41.122) 152.577ms
7  yahoo.iad01.atlas.cogentco.com (154.54.10.2)          152.617ms asymm 11
8  ae-6.pat2.dce.yahoo.com (216.115.102.176)             152.465ms asymm 12
9  ae-7.pat2.che.yahoo.com (216.115.100.137)             174.372ms
10 as-0.pat1.sjc.yahoo.com (216.115.100.66)              222.413ms asymm 16
11 ae-7.pat1.pao.yahoo.com (216.115.101.128)             216.010ms
12 gi-1-36.bas-b2.sp1.yahoo.com (98.136.16.39)           230.615ms
13 gi-1-44.bas-b2.sp1.yahoo.com (209.131.32.39)          217.424ms asymm 15
```

Windows 7 SP1

The Windows 7 client computer is owned by a personal friend, who ran a simple web server. The traceroute to the server is shown in the list below.

Listing A.5: Traceroute Winows 7 SP1

```
1  s22-00001.dsl.no.powertech.net (77.40.134.1)
2  s1010-0002.dsl.no.powertech.net (77.40.196.2)
```

Microsoft.com

The first commercial server was located in the USA, and it was the Microsoft server. The traceroute to the server is shown in the list below.

Listing A.6: Traceroute Microsoft.com

```
1  s02b02.no.powertech.net (195.159.88.82)               28.352ms
2  te0-0-0-5.ccr22.ham01.atlas.cogentco.com (130.117.2.181)  45.027ms
3  te0-0-0-3.ccr21.ams03.atlas.cogentco.com (130.117.50.41)  54.205ms
4  te0-1-0-2.ccr22.lpl01.atlas.cogentco.com (154.54.37.110)  63.486ms
5  te0-2-0-4.ccr22.bos01.atlas.cogentco.com (154.54.43.57) 137.888ms asymm 11
6  te0-6-0-6.mpd21.jfk02.atlas.cogentco.com (154.54.46.34) 138.442ms
7  te0-0-0-0.ccr21.jfk07.atlas.cogentco.com (154.54.27.210) 139.452ms
8  microsoft.jfk07.atlas.cogentco.com (154.54.11.150)   138.202ms
9  209.240.199.227 (209.240.199.227)                     137.915ms asymm 13
10 ge-0-0-0-0.nyc-64cb-1a.ntwk.msn.net (207.46.47.18)    139.298ms asymm 12
11 ge-7-0-0-0.co1-64c-1b.ntwk.msn.net (207.46.40.90)     215.174ms asymm 17
12 ge-0-1-0-0.wst-64cb-1b.ntwk.msn.net (207.46.43.185)   218.235ms asymm 18
```

```
13 ge-6-1-0-0.tuk-64cb-1b.ntwk.msn.net (207.46.47.72)   218.402ms asymm 17
14 ge-7-0-0-0.tuk-64cb-1a.ntwk.msn.net (207.46.47.68)   214.734ms asymm 16
15 ten1-2.tuk-76c-1b.ntwk.msn.net (207.46.46.15)        215.878ms asymm 17
16 po17.tuk-65ns-mcs-1b.ntwk.msn.net (207.46.35.146)    219.233ms
```

Apple.com

The second commercial server was Apple.com. The traceroute to the server is shown in the list below.

Listing A.7: Traceroute Apple.com

```
1  s22-00001.dsl.no.powertech.net (77.40.134.1)          29.815ms
2  s02b02.no.powertech.net (195.159.88.82)               30.245ms
3  te0-0-0-5.ccr22.ham01.atlas.cogentco.com (130.117.2.181)  44.959ms
4  te0-1-0-3.mpd22.fra03.atlas.cogentco.com (130.117.49.225)  54.927ms
5  te0-2-0-6.mpd22.dca01.atlas.cogentco.com (130.117.51.230) 150.398ms
6  te0-2-0-5.mpd22.iad02.atlas.cogentco.com (154.54.41.126) 152.027ms
7  192.205.35.105 (192.205.35.105)                       153.392ms
8  cr1.wswdc.ip.att.net (12.122.135.158)                 169.910ms asymm 17
9  cr2.rlgnc.ip.att.net (12.122.3.169)                   169.189ms asymm 16
10 cr1.rlgnc.ip.att.net (12.122.30.89)                   169.914ms asymm 15
11 cr82.chlnc.ip.att.net (12.123.138.22)                 175.654ms asymm 14
12 12.123.138.153 (12.123.138.153)                       168.936ms
13 12.248.48.10 (12.248.48.10)                           170.256ms asymm 14
```

Oracle.com

The third commercial web server was Oracle. The traceroute to the server is shown in the list below.

Listing A.8: Traceroute Oracle.com

```
1  s22-00001.dsl.no.powertech.net (77.40.134.1)          31.590ms
2  s02b02.no.powertech.net (195.159.88.82)               29.076ms
3  te0-0-0-5.ccr22.ham01.atlas.cogentco.com (130.117.2.181)  45.388ms
4  te0-1-0-3.mpd22.ams03.atlas.cogentco.com (130.117.50.37)  54.116ms
5  te0-0-0-2.ccr22.lpl01.atlas.cogentco.com (154.54.37.102)  63.259ms
6  te0-0-0-4.ccr22.ymq02.atlas.cogentco.com (130.117.0.253) 133.236ms
7  te0-0-0-7.ccr22.yyz02.atlas.cogentco.com (154.54.25.33) 141.132ms
8  te0-4-0-2.ccr22.ord01.atlas.cogentco.com (154.54.27.253) 154.183ms asymm  9
```

```
9  te0-0-0-3.ccr22.mci01.atlas.cogentco.com (154.54.30.101) 166.070ms asymm 10
10 192.205.36.181 (192.205.36.181)                          176.398ms asymm 13
11 cr2.dlstx.ip.att.net (12.122.138.110)                    178.813ms asymm 16
12 gar7.dlstx.ip.att.net (12.122.100.113)                   291.615ms
13 12.94.97.22 (12.94.97.22)                                176.787ms
14 adcq7-swi-8-xe-0-1-0.oracle.com (141.146.0.137)          183.021ms
```

Google.com

The fourth commercial web server was Google.com. The traceroute to the server is shown in the list below.

Listing A.9: Traceroute Google.com

```
1 s22-00001.dsl.no.powertech.net (77.40.134.1)          32.728ms
2 s02b02.no.powertech.net (195.159.88.82)               29.705ms
3 s01b01.no.powertech.net (195.159.50.41)               29.142ms asymm  3
4 oso-b4-geth3-0-13.telia.net (213.248.97.33)           28.703ms
5 s-bb1-link.telia.net (80.91.251.68)                   35.183ms
6 s-b3-link.telia.net (80.91.253.226)                   37.734ms
```

A.2 Source Code - Split ACK Tool

The source code for the Split ACK tool can be found at:
`http://folk.uio.no/renorman/SplitACK`

A.3 Data Results

The data results from all the tests can be found at:
`http://folk.uio.no/renorman/SplitACK/Results/`

Bibliography

[1] L. Eggert, J. Heidemann, and J. Touch. Effects of ensemble-tcp. *ACM SIGCOMM Computer Communication Review*, 30(1):15–29, 2000.

[2] W. Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. RFC 2001 (Proposed Standard), January 1997. Obsoleted by RFC 2581.

[3] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM Computer Communication Review*, volume 18, pages 314–329. ACM, 1988.

[4] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681 (Draft Standard), September 2009.

[5] Definisjon av bredbånd - norge. `http://img.nrk.no/nyheter/norge/1.7380901`, 2011.

[6] Homepage | ubuntu. `http://www.ubuntu.com`, April 2011.

[7] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960 (Proposed Standard), October 2000. Obsoleted by RFC 4960, updated by RFC 3309.

[8] Lksctp. `http://lksctp.sourceforge.net/`, June 2011.

[9] Andrés Arcia. *Modifications du mécanisme d'acquittement du protocole TCP: évaluation et application aux résaux filaires et sans fils*. PhD thesis, Telecom Bretagne, Rennes, France, December 2009.

[10] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. Updated by RFC 1349.

[11] Checksum - wikipedia, the free encyclopedia. https://secure.wikimedia.org/wikipedia/en/wiki/Checksum, 2011.

[12] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168, 6093.

[13] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122 (Standard), October 1989. Updated by RFCs 1349, 4379, 5884, 6093.

[14] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168 (Proposed Standard), September 2001. Updated by RFCs 4301, 6040.

[15] T.J. Socolofsky and C.J. Kale. TCP/IP tutorial. RFC 1180 (Informational), January 1991.

[16] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), October 1996.

[17] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581 (Proposed Standard), April 1999. Obsoleted by RFC 5681, updated by RFC 3390.

[18] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP congestion control with a misbehaving receiver. *ACM SIGCOMM Computer Communication Review*, 29(5):71–78, 1999.

[19] M. Allman. TCP Congestion Control with Appropriate Byte Counting (ABC). RFC 3465 (Experimental), February 2003.

[20] Ajax - connectivity enhancements in internet explorer 8. http://msdn.microsoft.com/en-us/library/cc304129(VS.85), April 2011.

[21] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin. An argument for increasing TCP's initial congestion window. *ACM SIGCOMM Computer Communication Review*, 40(3):26–33, 2010.

[22] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785.

[23] D. Liu, M. Allman, S. Jin, and L. Wang. Congestion control without a startup phase. In *Proc. PFLDnet*. Citeseer, 2007.

[24] S. Floyd, M. Allman, A. Jain, and P. Sarolahti. Quick-Start for TCP and IP. RFC 4782 (Experimental), January 2007.

[25] S. Floyd. Limited Slow-Start for TCP with Large Congestion Windows. RFC 3742 (Experimental), March 2004.

[26] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window. RFC 3390 (Proposed Standard), October 2002.

[27] Tcp: Increase the initial congestion window to 10 packets. `http://kernelnewbies.org/Linux_2_6_39#head-c2acd2f0463943210471a42bf6f5b469a6999e7b`, June 2011.

[28] Increasing the tcp initial congestion window. `http://lwn.net/Articles/427104/`, June 2011.

[29] Tcpdump/libpcap public repository. `http://www.tcpdump.org/`, August 2010.

[30] Netfilter/iptables project homepage. `http://www.netfilter.org/`, June 2010.

[31] Gnu wget. `http://www.gnu.org/software/wget/`, November 2010.

[32] G.992.5 : Asymmetric digital subscriber line (adsl) transceivers extended bandwidth adsl2 (adsl2plus). http://www.itu.int/rec/T-REC-G.992.5-200901-I/en, 2011.

[33] Powertech information systems as. http://www.powertech.no/, 2011.

[34] Nmap - free security scanner for network exploration & security audits. http://nmap.org/, April 2011.

[35] Alexa the web information company. http://s3.amazonaws.com/alexa-static/top-1m.csv.zip, June 2011.

[36] Google and microsoft cheat on slow-start. should you? http://blog.benstrong.com/2010/11/google-and-microsoft-cheat-on-slow.html, 2011.