Technische Universität München
Lehrstuhl für Kommunikationsnetze
Prof. Dr.-Ing. Wolfgang Kellerer

Master's Thesis

Implementation and Performance Evaluation of TCP Extensions in FreeBSD

Author:	Aris Angelogiannopoulos
Matriculation Number:	03626880
Email Address:	aris.angelog@hotmail.com
Supervisor:	Dr. Lars Eggert
Begin:	15. April 2013
End:	15. October 2013

Abstract

TCP is the most widely used transport layer protocol and is used in a big range of applications. Understanding its behavior, is critical in order to properly engineer, operate, and evaluate the performance of the Internet, as well as to properly design and implement future networks. It aims into providing fast and reliable communication over an unreliable network such as the modern Internet. In order to achieve that, it is equipped with a number of mechanisms that allow it to recover from losses etc.

How fast a transmission is done, meaning how fast all bytes available for transmission from a sender to a receiver are sent is very important. This thesis explores two extensions of standard TCP that aim into decreasing the time needed for data transmission. The first one is the Proportional Rate Reduction (PRR) [DMCG11], an algorithm proposed as an alternative to the standard Fast Recovery algorithm of TCP, used for loss recovery. PRR addresses all the weaknesses that the standard algorithm has. The second extension that is explored is New Congestion Window Validation [FSS13], a method that improves TCP performance for rate-limited traffic. This traffic is produced by applications, which do not use the entire bandwidth available to them by TCP.

An implementation for each extension was done in the TCP/IP stack of the FreeBSD kernel. Their performance was evaluated. The results show that the transmission time is lowered for all the scenarios that have been evaluated. The results for each extension are presented at the respective section of each chapter.

Acknowledgements

This thesis has been done at the NetApp GmbH Research Lab in Munich, as the final part of the M.Sc. in Communications Engineering of the Technical University of Munich.

I would like to express my sincere acknowledgement to my NetApp supervisor, Dr. Lars Eggert for his invaluable support and guidance through the study. He did not only give me the opportunity to work on such advanced technologies such as the TCP/IP stack of a real operating system, but also he provided me with very useful advice, technical guidance and recommendations for my work. I would also like to thank my colleague at NetApp, Dr. Alexander Zimmermann who provided me with his support and guidance during the evaluation, experimentation and writing part of my thesis.

Furthermore, I would like to thank Prof. Dr. Wolfgang Kellerer, the chairman of the Institute for Communication Networks at TU Munich, and Dipl.-Inf. Andreas Blenk, my TUM supervisor, for giving me this opportunity and providing very informative feedback and assistance.

Contents

Chapter 1

Introduction

The Transmission Control Protocol (TCP), allows computers and systems of all sizes, from many different vendors, running totally different operating systems, to communicate with each other. TCP started in the late 1960s as a US government research project into packet switching networks. The Internet Engineering Task Force (IETF) has published the first RFC, which standardized TCP in 1981 [Pos81]. The main focus of the latter was to present TCP which, back then, was intended as a protocol aiming to assist communications between military system. A lot of time has passed since then and TCP is now, by far, the most used communication protocol between computer systems. Recent studies show that TCP accounts for 85% to 90% of the traffic in the internet both in the Backbone [JT07] and the Access Networks [PCUKEN09]. It is a truly open system since the definition and many of the implementations are publicly available at little or no charge.

TCP aims to provide reliable, end-to-end communication between two hosts. It includes a number of mechanisms that guarantees that data is delivered in order to the application by the protocol even if losses are present. TCP is used by a large number of applications including, smartphone apps, browsers, video streaming services etc. There has been very extensive research that aims into improving the mechanisms of the protocol and improve its performance with all types of traffic. There exists a great number of proposal and extensions, this is the reason that IETF has created a document that provides a roadmap to the various extensions that IETF has accepted and standardized since the first RFC discussed earlier [DBEB06].

1.1 Motivation

As discussed before, TCP provides a number of mechanisms that enable reliable communication, Loss Recovery and Congestion Control to name a few. Extensive research has been and is still being done aiming to improve performance of these mechanisms.

The term performance is somewhat vague, it is better seen as what the user understands as better performance. For example, better performance in a file transfer, or in that context any bulk data transfer, means a lower transmission time. Better performance in a streaming video service such as Youtube means faster transfer time of the data blocks that consist the video, such that the user experiences no delay or idle times (less jitter) while watching the video. In other words, how fast a transmission is done, meaning how fast all bytes available for transmission from a sender to a receiver are sent is very important.

This thesis investigates two TCP extensions that aim into improving the performance seen by the user as described before. The first one is Proportional Rate Reduction [DMCG11] and aims into improving the Fast Recovery mechanism of TCP. Fast Recovery is the mechanism that enables TCP to recover from losses fast and accurate. The second extension is called New Congestion Window Validation [FSS13] and aims into providing a method that improves the standard use of congestion control algorithm methods in order to support rate-limited applications, e.g., applications that do not consume the whole of the bandwidth that TCP assigns to them. It will be shown that these extensions are able to improve performance when compared to standard TCP.

1.2 Structure of the Thesis

The work done in this thesis that is implementation of the extensions in the FreeBSD TCP/IP stack and the evaluation of their performance are presented in this thesis. In order to give a clear view, the thesis is divided into six chapters.

Following an overview given in Chapter 1, TCP is explained and presented in Chapter 2. A brief overview of the protocol is given by examining the most important mechanisms that consist it. Chapter 3 describes the environment used for performance evaluation of the implemented TCP extensions. Chapter 4 describes the Proportional Rate Reduction (PRR). Along with the description of the algorithm, the results of the performance evaluation are presented. The same is true for New Congestion Window Validation that is discussed in Chapter 5. Finally, Chapter 6 contains a brief summary of the study and some suggestions for possible future work.

Chapter 2

Transmission Control Protocol

TCP is the element of the original Internet protocol suite that provides reliable data transmission between two peers across an unreliable IP-based communication channel. There exists a large number of documents that update its behavior since the original specification which can be found in [Pos81]. A roadmap of these documents can be found in [DBEB06]. TCP and the mechanisms that it includes constitute now the foundation of Internet congestion control.

TCP, like UDP, is a transport layer protocol. Even though both of these protocols use the same network layer (IP), TCP provides a totally different service to the application layer than UDP does. It provides a connection-oriented, reliable, byte stream service.

The term connection-oriented means that the two parties participating in a connection must first establish a connection through a mechanism called three-way handshake before they can exchange data.

The term reliable means that TCP guarantees that the data the application sends will be transmitted to the receiver. Due to network congestion, traffic load balancing or other unpredictable network behavior, packets can be lost, duplicated or delivered out of order. TCP detects these problems and requests retransmissions, discards duplicates, rearranges out-of-order data and helps also minimize network congestion through its congestion control algorithm.

The term byte stream service means that there is a stream of bytes exchanged across a TCP connection. TCP does not interpret the contents of the bytes at all meaning that sending does not depend on whether the data are binary, ASCII etc. This task is left to the application participating in the communication stream.

This chapter presents the main functions and mechanisms that TCP consists of.

2.1 Fundamental functions of TCP

As discussed, the main difference of TCP against UDP is the reliable communication that it provides. It uses sequence numbers and acknowledgements in order to identify that sent segments have arrived to the receiver. If a segment is lost, TCP has to resend it before sending any new data.

TCP has mechanisms that help it identify that a sent segment is lost and thus retransmit it. The main such mechanism, is called Fast Recovery and it allows fast identification of a lost packet and retransmission. The second way acts as a fallback in the case that Fast Recovery does not work. TCP maintains a timer that defines the maximum amount of time that a sender can wait for a segment to be acknwoledged. If the acknowledgement is not received by the time that the timer expires, TCP retransmits then the segment, which is assumed lost. Note that a lost packet is not the only reason that an acknowledgement is not received. TCP maintains a checksum on its header and data in order to detect any modifications on the data transmitted. A packet may have arrived at the receiver with an invalid checksum causing the receiver to discard and not acknowledge it. Another reason would be for the ACK packet to be lost as well.

TCP also provides flow control, meaning that it controls the amount of data that are present in the network at any given time through a window based mechanism that will be discussed in the next section.

Another function that TCP offers is the correct reassembly of segments arriving out of order. TCP transmits segments as IP datagrams, and since IP datagrams can arrive out of order, the receiver side of the connections has to re-sequence the data and pass them in the correct order to the application.

A TCP segment consists from the header and the data part. The header maintains all the information that are needed in order for TCP to be able to provide all the functions discussed before. Its normal size is 20 bytes except if options are present. In that case, the length is determined by the data offset, options can have a variable length of 0-320 bits (should be divisible by 32, padding is used otherwise). The value of the data field depends on the maximum segment size (MSS) of the path and the length of the header.

Figure 2.1 shows the structure of a TCP segment. As seen, it consists of a number of fields:

- **16-bit source port**: This field identifies the port that the segment was sent from
- **16-bit destination port** : This field identifies the port that this segment is sent to
- **32-bit sequence number**: This field has a dual role. If the *SYN* flag is set, this is the initial sequence number. The sequence number of the actual first data byte and the acknowledged number in the corresponding ACK are then this sequence number plus 1. If the *SYN* flag is not set, then this is the cumulative sequence number of the first data byte of this segment for this connection

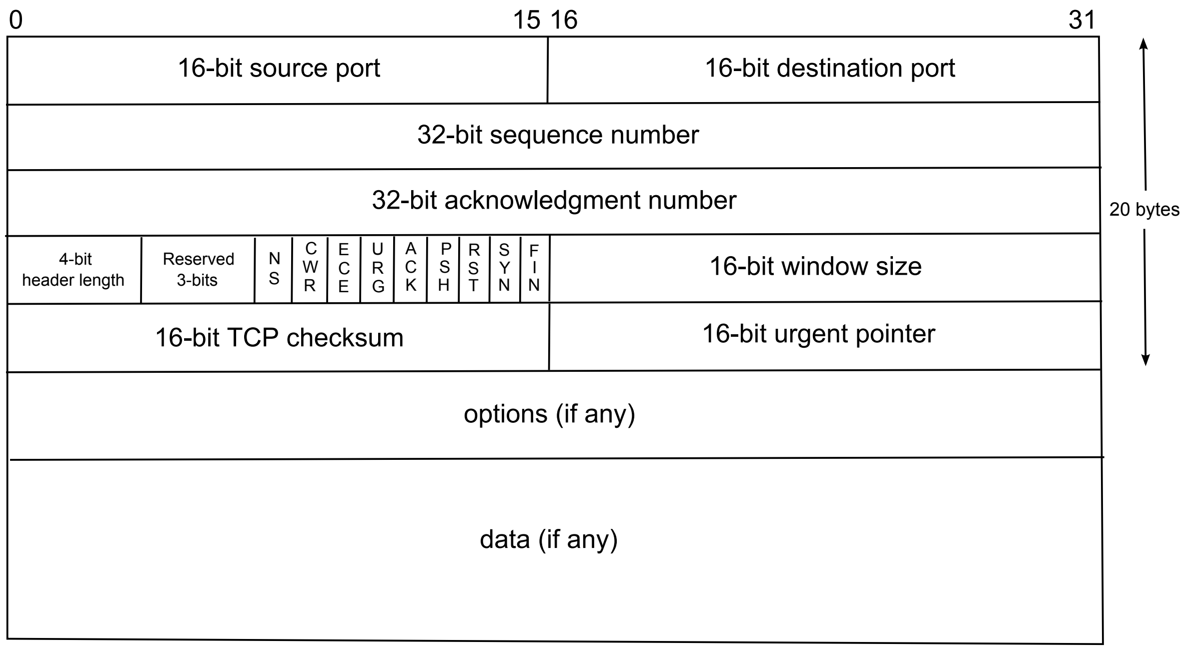

Figure 2.1: TCP header

- **32-bit acknowledgment number**: If the *ACK* flag is set, then this is the next sequence number that the receiver is expecting. This field acknowledges the sequence number up to this value.
- **4-bit header length**: This field specifies the size of the TCP header in 32-bit words. The minimum size of the header is 5 words and the maximum is 15 words thus giving the minimum size of 20 bytes and maximum of 60 bytes, allowing for up to 40 bytes of options in the header.
- **Reserved bits**: This field is reserved for future use and should be set to 0.
- **Flags**: There are nine control flags used in a TCP segment. *SYN*, *ACK* and *URG*, when set, indicate that the value in the corresponding field is valid. *NS*, *CWR* and *ECE* are used for Explicit Congestion Notification (ECN) [RFB01]. *FIN* is used to indicate the end of a connection. *PSH* is used to perform a push action. It requests to push the buffered data to the application. Lastly, *RST* is used to reset a connection
- **16-bit window size**: The size of the receive window in bytes that the sender of this segment is currently willing to receive. In most cases, it is the current size of the buffer of the receiving application
- **16-bit TCP checksum**: The checksum field that is used for error checking

- **16-bit urgent pointer**: If the *URG* flag is set, this field indicates the offset from the sequence number indicating the last urgent data byte
- **Options**: This field is optional and carries TCP options such as the MSS, the amount and sequence numbers of Selective Acknowledgment (SACK) blocks etc. MSS is used to indicate the maximum segment size allowed in the path and SACK is used by a receiver to notify a sender for delivered off-sequence segments
- **Data**: This optional field carries the actual data that an application wants to transmit.

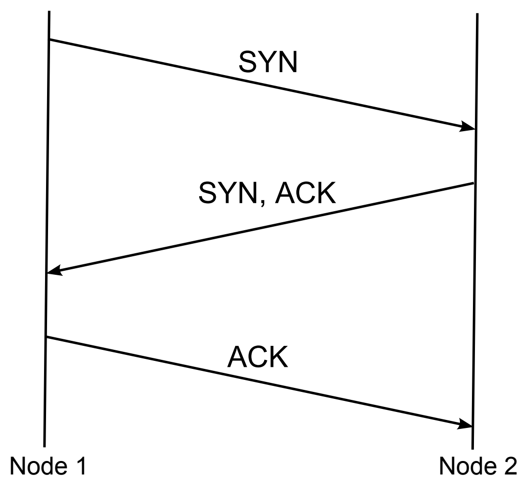

Figure 2.2: TCP initial handshake

As already mentioned, TCP is a connection oriented protocol. This means that in order to reliably transfer a data stream from one end of a network to another, it first needs to establish a logical connection using a three-way handshake. Figure 2.2 shows this procedure. Node 1 sends to Node 2 a segment with the *SYN* flag set, in order to initiate a connection. Node 2 replies with a *SYN-ACK*, meaning that both of these flags are set. Node 1 acknowledges then that a connection is now active with an *ACK*. The connection is now set up and the two nodes can exchange data. Note that these first segments can also carry information useful for the initialization of the connection such as the MSS, the initial sequence number etc. A recent extension, proposes that a connection can carry data while performing the three-way handshake, this is achieved by the use of a cryptographic cookie [RCC+11]. This approach can significantly speed up connections that require only a few round trips, something that is common in todays internet.

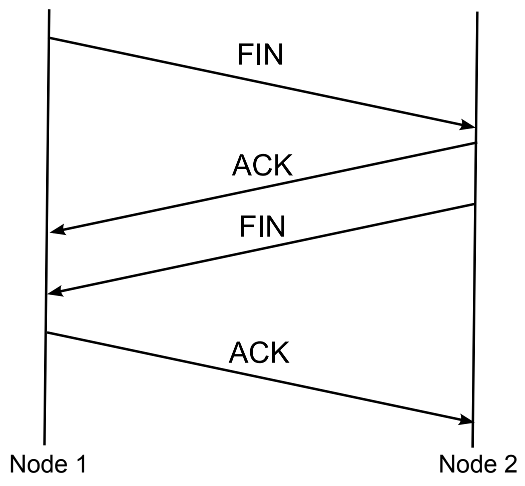

Figure 2.3: Termination of a TCP connection

While it takes three segments to establish a connection, it takes four to terminate one. This is caused by TCP ‘s half-close. Since each TCP connection is full duplex, meaning that data can flow to each direction independently, each part of the connection has to close independently. The rule is that both ends can send a *FIN* when it is done sending data. The receipt of the *FIN*, only means that there will not be any other data flowing in that direction. Figure 2.3 shows the termination of a TCP connection. Node 1 finishes sending data and thus sends a *FIN*. Node 2 acknowledges the *FIN* and sends its own *FIN* to Node 1. Lastly, Node 1 acknowledges the *FIN* from No2. The connection is now closed.

Note that for both the initiation and termination of a connection, the simplest scenarios are presented. There can be many issues, e.g., a *FIN* or *ACK* might get lost. Since these are normal segments the usual procedure described in the next section will be followed in order to recover the losses.

2.2 Flow Control

TCP uses window-based Flow Control. This means that the receiver carries out flow control by granting the sender a window, called receive window, of a certain amount of data that is willing to receive. The sender must not send more than what this receive window allows at any point, without receiving an acknowledgment for already sent packets.

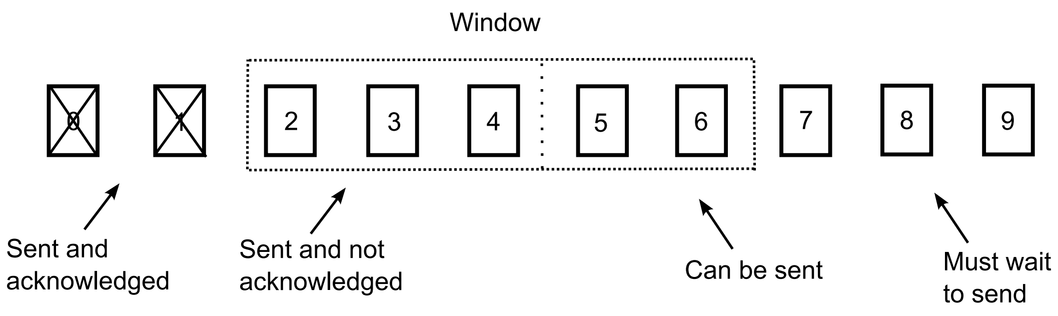

Figure 2.4: TCP sliding window

Figure 2.4 depicts the way that Flow Control works in TCP. In this example, the advertised window consists of 5 segments. Since segments 0 and 1 have already been acknowledged, the receive window covers segments 2 to 6. Segments 2 to 4 have been sent but not acknowledged. Segments 5 and 6 can be sent but are not, this can be a result of congestion control taking place. If an acknowledgement for segment 2 comes and the receive window remains the same the sender is then allowed to advance (slide) the window one position to the right. In the case that the received ACK indicates that the receive window has grown (opened), the sender is then allowed to transmit more packets. Similarly, if it closes less packets must be transmitted.

2.3 Congestion Control

Congestion Control mechanisms, control the amount of data entering the network and keep the rate below a value that could trigger collapse. Acknowledgements of data sent, or lack of them are used as an indication for TCP to understand network conditions. Modern implementations of TCP include four algorithms: Slow Start, Congestion Avoidance, Fast Retransmit and Fast Recovery [APB09].

2.3.1 Slow Start and Congestion Avoidance

In addition to the receive window discussed in the previous section, Congestion Control adds another window to the sender's TCP. This window is the congestion window. The sender can transmit up to the minimum between the congestion window and the receive window advertised by the receiver. When a new connection is established, the congestion window is initialized to three segments. Each time the sender receives an ACK, it increases the congestion window by one segment. The congestion window can be seen, as flow control imposed by the sender while the receive window as flow control imposed by the receiver.

At the beginning of the connection, the sender sends a segment and waits for its ACK. When it is received the congestion window is incremented by one and the sender sends now two segments. After the next *ACK* the sender can send 3 segments and so on. This is an exponential increase to the congestion window. After a while the limit of the intermediate network is reached and an intermediate router may start to discard packets. This is an unwanted behavior since lost packets mean a decrease in performance.

The assumption of the algorithm is that packet loss signals congestion somewhere in the network between the source and the destination. Congestion Avoidance defines the limit up to which a TCP sender may perform Slow Start. It defines a variable called slow start threshold (*ssthresh*). After the congestion window reaches *ssthresh* the sender then stops exponentially increasing it and rather performs an additive increase as follows:

```
cwnd = cwnd + MSS * MSS / cwnd
```

This means that the window will be increased by at most one segment per RTT. *ssthresh* is initialized to a rather small value depending on the implementation (but not less than 3**MSS*). When a loss event occurs, half of the value of the congestion window is saved as the value of *ssthresh*.

2.3.2 Fast Retransmit and Fast Recovery

Fast Recovery is the main algorithm that allows TCP to recover from a loss. Before it is described, it should be noted that TCP is required to send an immediate acknowledgment (duplicated ACK), when an out of order segment is received. The purpose of this duplicate ACK is to let the sender know that a segment was received out of order and thus indicating that a packet might be lost.

Fast Recovery algorithm states that after the receipt of three consecutive duplicate ACKs, the TCP sender must enter Fast Recovery and retransmit the next unacknowledged segment (Fast Retransmit). The value three is chosen because it is not possible to know if the duplicate ACK was generated because of a lost segment or it was the result of reordering. In the case of the latter, more ACKs are needed in order to assure that the packet was lost. It is assumed that if the duplicate ACK was the result of reordering, there will only be one or two duplicate ACKs before the reordered segment is processed.

After the Fast Retransmit, Fast Recovery is performed. Fast Recovery consists of a set of rules that dictate the way that the congestion window should increase during recovery. Note that there exist various congestion control algorithms that dictate the way that the congestion window should be increased during e.g. Congestion Avoidance, such as Vegas [BP95], CUBIC [HRX08] etc. Modern Fast Recovery algorithms use, in addition to normal ACKs, selective acknowledgments (SACK) [MMFR96]. SACK is implemented as a

TCP option, which is negotiated between the two parties participating in the connection. It allows the receiver to notify the sender in regards with the received out of order segments, i.e., the exact number of bytes that are successfully received. The use of SACK can significantly increase the performance of a TCP connection. Fast Recovery will be studied in detail in Chapter 4.

2.3.3 Conclusion

This chapter has provided a quick overview of TCPs main characteristics and mechanisms. TCP is a quite complicated protocol and there exist many books about it. For a detailed study, the reader can refer to [Ste93]. All the essential TCP details described in this chapter are specified in standards track RFCs, which essentially means that the IETF recommends their implementation. [DBEB06] provides a roadmap to the various TCP extensions that have been proposed and adopted over the years since the original specification.

Chapter 3

TCP Performance Evaluation

TCP, as presented in the previous chapter, leaves a lot of room for various improvements that were made over the years. For the needs of this thesis, two extensions were implemented. The first one is the Proportional Rate Reduction algorithm, a modern Fast Recovery scheme that aims to update the standard algorithm and overcome its weaknesses. The second one is the New Congestion Window Validation, a method aiming to provide better support to applications experiencing rate-limited and/or idle periods. For both extensions, a performance evaluation was also done.

This chapter will present the evaluation topology, as well as the tools that were used in order to evaluate the performance of the implemented extensions.

3.1 Topology

One of the topologies, most frequently used, in order to evaluate end-to-end protocols, is the one introduced in the Internet Draft "*Common TCP Evaluation Suite*" [AMF$^+$08] from the Transport Modeling Research Group (TMRG) of the Internet Research Task Force (IRTF). This topology is known as the Dumbbell Topology and is depicted in Figure 3.1. As it can be seen from the Figure, this topology consists of 3 different group of nodes. Two of the three groups have almost the same function, meaning that they serve as end points for the protocol. One of these two groups is the group of the senders and the other one is the group of the receivers. The third group that lies in the intermediate link between the senders and the receivers, is the group of the routers that introduces various functions, such as loss, delay etc.

A characteristic of the Dumbbell topology is that there is only one link that lies between the group of the senders and the group of the receivers so that there is only one way for the two parties to communicate. This means that they all have to share the bandwidth

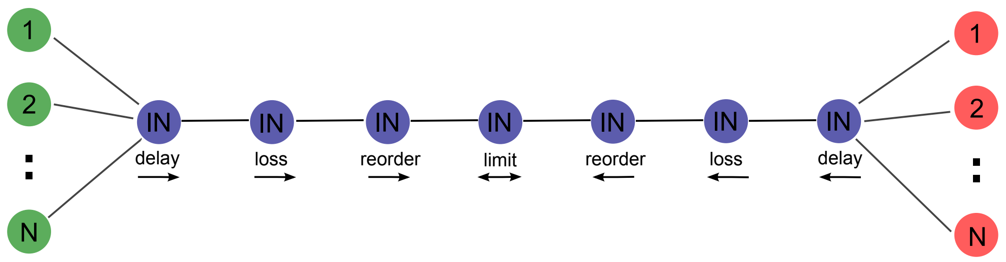

Figure 3.1: Dumbbell topology

and the queues of this link (router) so that a number of scenarios can be created such as an access link to an Internet Service Provider (ISP), a trans-oceanic link [AMF$^+$08] etc.

For the needs of this thesis, two TCP extensions were implemented in the FreeBSD kernel. For evaluating their performance, a Dumbbell topology was setup, that enabled the required experiments to be performed.

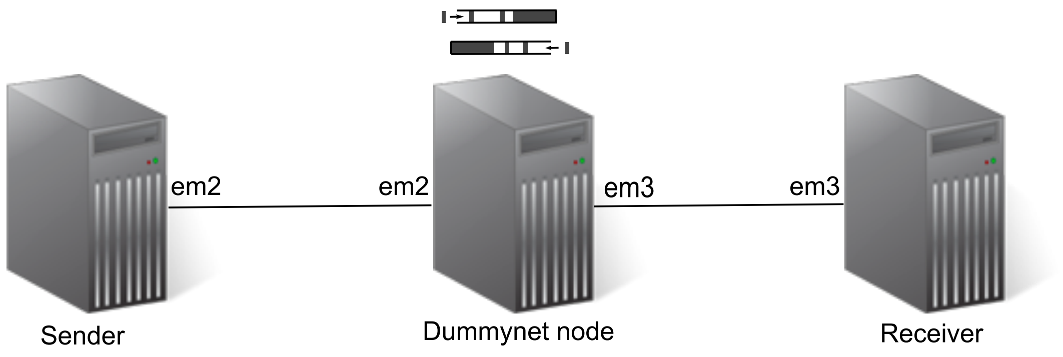

Figure 3.2: Experiment lab setup

Figure 3.2 shows the lab setup that was used for the evaluation. There are three nodes present in this setup. The sender node, initiates the TCP flows to the receiver. The TCP packets (both data and ACK packets) go through the intermediate node, which is marked as the dummynet node. Dummynet [CR09] is a live network emulation tool, originally designed for testing networking protocols, and since then used for a variety of applications including bandwidth management. In this setup, dummynet is used as a traffic shaper, able to perform all the functions that the Dumbbell topology that is presented before introduces (e.g. delay, loss).

Various evaluation scenarios were used, where different variables such as the bandwidth of the path and the round trip time were varied. A key note should be made here regarding the size of the queue of the intermediate node (defined also by dummynet like all the other parameters). Depending on the bandwidth and the delay of the path, the queue length should be calculated, such that, it is neither too small, nor too big. If the queue length is too small, it can happen that after a short amount of time the queue becomes full. If this happens, the extra packets that arrive after the queue is filled, will be dropped, causing the TCP sender to limit its sending rate and thus introduce greater unnecessary latency. If the queue length is too big, it is possible to fully utilize the link with a small amount of dropped packets. This will cause a big increase in the End-to-End Delay and the Round-Trip Time and thus create a very long path, which is undesirable. In order to avoid the queue length to be either too small or too big, it is configured as the Bandwidth-Delay product. This means that if B is the bandwidth of the path and R is the configured round-trip time, the length of the queue should be equal to their product B*R. For instance if B=2Mbit/s and R=40ms, the queue length is then equal to 7 packets with a packet size of 1500 bytes.

Multiple senders (flows) and receivers were simulated by flowgrind [ZHK10]. Flowgrind is a software application able to generate network traffic in order to measure throughput, transmission time and other TCP metrics in order to evaluate a TCP extension.

3.2 Tools

In order to evaluate the extensions that were implemented, a number of tools were used, that allow a closer inspection of a TCP connection. These tools include dummynet used for traffic shaping, flowgrind, used for generating traffic in the network, tcpdump for capturing packets of a TCP connection and examine their fields and tcptrace as well as xplot to visualize a TCP connection.

3.2.1 Dummynet

Dummynet is a widely used link emulator, developed to run experiments in user-configurable network environments. Since its original design, it is extended in various ways and has become very popular in the research community since it can emulate even moderately complex network setups on unmodified operating systems [CR09]. The original version was developed for FreeBSD and OSX. There currently also exist versions for Windows and Linux.

As discussed in the previous section, dummynet was used in the intermediate node in order to control the bandwidth of the path, the length of the queues and the round trip time between the sender and the receiver. Dummynet works by intercepting selected packets on their way through the protocol stack. It then passes the packets to objects called pipes, which implement a set of queues, a scheduler and a link, all with configurable parameters, that is bandwidth, delay, loss rate, queue size etc.

Traffic selection is done using the ipfw firewall, which is the main user interface for dummynet. ipfw lets the user select precisely the traffic and its direction that he wants to shape. It is very powerful and allows the specification of various scenarios, e.g. create multiple pipes, pipelines, specify different behavior depending on the protocol etc. The following scenario illustrates an example usage of dummynet. These commands limit the incoming TCP packets at the rate of 2 Mbit/s.

```
ipfw add pipe 2 in proto tcp
ipfw pipe 2 config bw 2Mbit/s
```

The first line creates a **pipe** with the id number 2. The keywords **in** and **tcp** specify that this rule applies to the incoming tcp packets only. After creating the pipe, the second line configures the bandwidth such that it is 2Mbit/s.

As discussed, dummynet was used for the evaluation of the TCP extensions, implemented in FreeBSD for the needs of this Thesis. For this reason, dummynet ran in the intermediate node of Figure 3.2 in order to shape the traffic and give it the characteristics that each measurement scenario required. As an example, the following commands are given:

```
ipfw add pipe 111 tcp from 240.0.3.2 to 240.0.4.3 out via em3
ipfw pipe 111 config queue 7 bw 2Mbit/s delay 20ms plr 0.01
```

The above commands, create a pipe for TCP packets sent from the sender (240.0.3.2) to the receiver (240.0.4.3) via the em3 interface. This pipe is then configured with a delay of 20ms, a queue with a size of 7 packets, a bandwidth of 2Mbit/s and a random loss of 1%. Note that the actual round trip time of the path will be 40ms and not 20ms since another pipe has to be configured for the opposite direction for ACK packets.

3.2.2 Flowgrind

Flowgrind [ZHK10] is a testing and benchmarking tool, used to measure throughput and other metrics in TCP connections. In contrast to other performance tools like iperf [hHK98] or netperf [Jon], it features a distributed architecture, meaning that it is split into two components: the daemon and the controller. Using the controller, flows between any two systems that run the daemon can be established. At regular intervals during the test, the controller collects and displays the measured results from the daemons. It can run multiple flows at once using the same or different interfaces. These flows can have the same or individualized options, like the traffic type (e.g. bulk, rate-limited, request-response etc). The following line states an example of its usage:

```
flowgrind -H s=host1,d=host2 -T s=0,d=10 -G s=q,C,800 -G s=g,N,0.008,0.01
```

The above command, will start a flow from host1 (source) to host2 (destination) for 10 seconds. The traffic type is set to be rate-limited by using normal distributed interpacket gap with mean 0.008 and a small variance (0.001). In addition with using request size of 800 bytes, a average bitrate of approximately 800 kbit/s is achieved.

Flowgrind can generate a varied type of traffics and provide a number of options that make it a better choice than other network performance tools. The reader can refer to [Sam11] for a comparison between network performance tools.

3.2.3 Tcpdump

Tcpdump [Tcp] is an open source packet analyzer that runs from the command line. It allows the user to capture TCP/IP or other packets being transmitted over a network. It is a powerful tool in analyzing network behavior and performance. It is possible to generate capture files that are used by tools such as tcptrace, described in the next section, in order to visualize a connection by using time-sequence graphs.

3.2.4 Tcptrace and xplot

The last two tools that were used for the evaluation of the TCP extensions that were implemented, are tcptrace [Ost] and xplot [She]. Tcptrace uses as input a packet capture file, generated by tcpdump and then creates output that are used by xplot in order to create time-sequence graphs. These graphs are an easy way of depicting a TCP connection and understand the way it works. Xplot provides an easy interface to handle those graphs and is thus a powerful and important tool.

3.3 TCP time-sequence graphs

TCP time-sequence graphs are an easy way to depict a TCP connection and understand the behavior of the system. Figure 3.3 shows an example time-sequence graph that depicts the way the Fast Recovery algorithm works. The advancing green line represents the forward movement of the advancing sequence numbers as they are acknowledged by the receiver. SACK blocks are depicted by using purple and black is the actually transmitted block of data.

As it can be seen in Figure 3.3, duplicate acknowledgements carrying SACK blocks are received, indicating a lost packet. After the duplicate packet threshold is reached, typically three, TCP enters Fast Recovery and transmits the lost packet. It then waits for multiple duplicate and partial ACKs up to the point where the lost packet is acknowledged. Partial

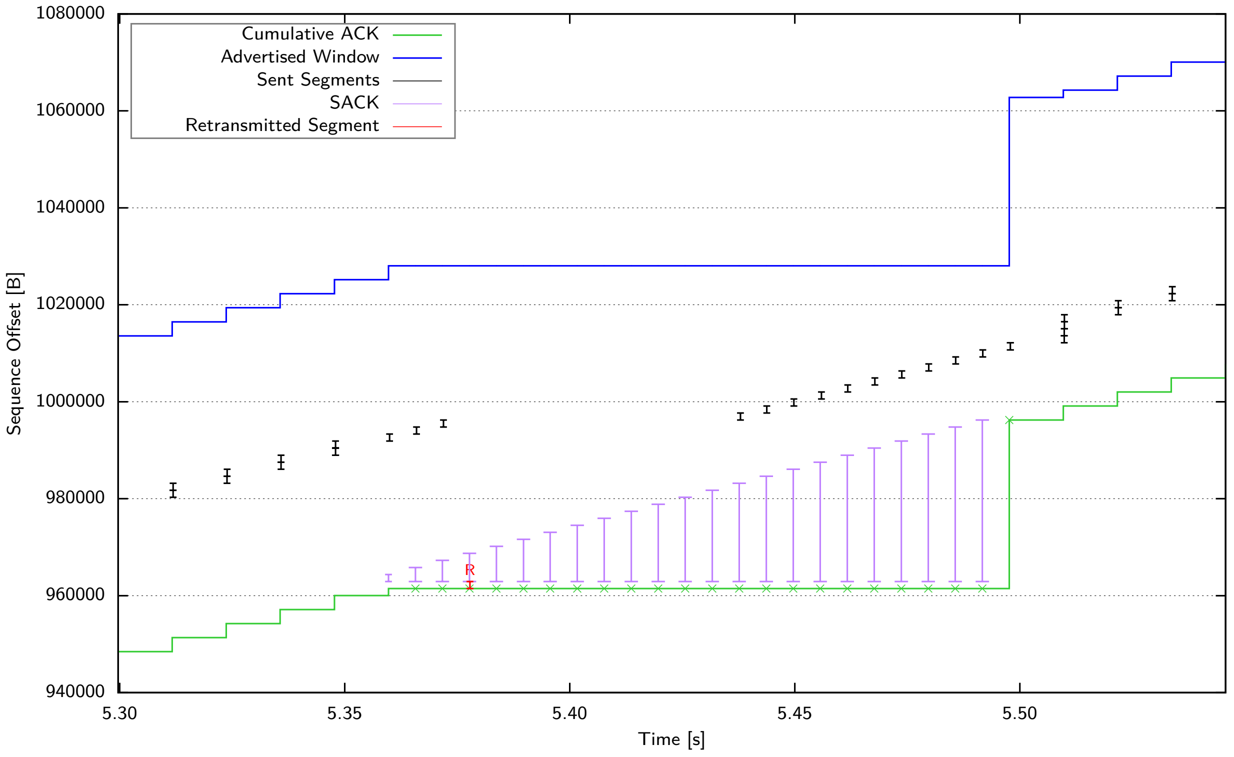

Figure 3.3: Time-Sequence Graph of a Fast Recovery event

ACKs advance but not cover the recovery point. TCP sender exits then Fast Recovery and continues to transmit data to the receiver.

Chapter 4

Proportional Rate Reduction

The first algorithm that this thesis addresses is Proportional Rate Reduction [DMCG11]. TCP, as already discussed, typically operates in lossy networks, e.g., networks that are unreliable, such as wireless or low bandwidth environments. This means that the sender can not trust the network to deliver a packet to the destination. A packet can be lost and this is the reason that TCP was designed with mechanisms that try to recover the losses as fast as possible. Packet losses, typically introduce high latency to the user space application and are therefore unwanted.

TCP recovers from a lost packet through two primary ways. The first is the Fast Retransmit/Recovery algorithm that retransmits a lost packet, after receiving a number of duplicate ACKs (typically three, called dupthresh). As a fall back in the cases where the Fast Recovery algorithm does not apply, e.g, not enough duplicate ACKs are received, TCP uses a second, slower but more robust mechanism to identify lost packets. It waits for an amount of time called the Retransmission Timeout (RTO) before assuming that a packet is lost and retransmitting it.

The SACK based Fast Recovery algorithm, which is described in [BAFW03], is the main mechanism, that is used in order to recover from losses. This thesis will explore some of the weaknesses that the standard algorithm has and describe the Proportional Rate Reduction [MDC13], a new experimental algorithm designed for (fast) loss recovery as an alternative to the currently widely deployed scheme. It will then evaluate the performance of PRR through an implementation in FreeBSD against the standard algorithm that is deployed there.

The evaluation will show, that the standard algorithm deviates from its intended behavior in the real world, due to the combined effect of short flows, application stalls (limited available amount of data to send), ACK loss etc. It will be shown that the Proportional Rate Reduction algorithm, performs better, meaning reduced latency, due to the fact that it recovers from losses smoothly, quickly and accurately by pacing out retransmissions or new data transmissions across received ACKs.

The Proportional Rate Reduction is inspired by the Rate Halving Algorithm [MSM99, DMCG11], which will also be presented. PRR has been approved to become the default Linux fast recovery algorithm for Linux 3.X [DMCG11].

4.1 Related Work

The Proportional Rate Reduction algorithm is based on the Rate Halving algorithm [DMCG11] and as such, a quick overview of the latter [MSM99] is needed.

As discussed, most TCP Reno implementations use the Fast Recovery algorithm, which, when a packet is lost, waits for a number of duplicate ACKs (typically three) before it assumes a packet is lost and retransmits it. After the retransmission, TCP has to wait for enough additional duplicate ACKs to arrive which indicate that half of the data that were in flight when the loss happened has left the network, i.e. they were delivered to the receiver. Only when this has happened will TCP inject new data into the network.

This behavior is known as Half RTT silence and wastes precious opportunities to send new data. It sometimes results in no new data being sent during recovery, which in turn increases the chances of a timeout and thus greater latency. Another consequence of this design is that, the entire new window of data is transmitted on one half of one Round Trip Time (RTT). This causes a bursty behavior which can lead to additional bursts in the successive RTTs following recovery and thus increases the likelihood of a new loss occurring.

The Rate Halving algorithm is using an idea initially by Hoe [Hoe95], who suggests that during TCP Fast Recovery, the data sender should space out transmissions (both retransmissions and new data) on alternate returning ACKs across the entire recovery RTT. Rate Halving mainly reduces the congestion window following the detection of congestion in the network. Rate-Halving reduces the congestion window over one RTT by transmitting one new packet for every exactly two packets acknowledged by the receiver.

During the adjustment interval, the window is reduced by sending one packet for each two segments, which are acknowledged. The choice of which data to send (new or retransmissions) is completely independent from the Rate Halving algorithm and is done by the TCP stack.

The Proportional Rate Reduction algorithm is also inspired by Hoe's suggestions that there should be no half RTT silence during the recovery period but the responses to incoming ACKs are calculated in dependence on the amount of data that the SACK information indicates that were delivered by the network. This is in contrast with the Rate Halving algorithm that sends one data segment every second incoming ACK.

4.2 Fast Recovery Proposals

The following subsections will discuss the SACK based Fast Recovery described by the RFC standards as well as the one implemented in FreeBSD. The main drawbacks of the algorithms will be presented.

4.2.1 Fast Recovery in RFC standards

The standard TCP congestion control algorithms are described in [APB09]. The four algorithms that are described there are slow start, congestion avoidance, fast retransmit and fast recovery. Their use with TCP were standardized in [Bra89]. [BAFW03] describes a SACK-based loss recovery algorithm that is the current standard in most modern operating systems. The Fast Recovery algorithm used in FreeBSD, which is the one that this thesis examines and uses as a comparison to the Proportional Rate Reduction, is also based on RFC 3517.

```
RFC 3517 Fast Recovery algorithm
On entering recovery:
// cwnd used during and after recovery
cwnd = ssthresh = Flightsize/2

// Retransmit first missing segment
fast_retransmit()
// Transmit more if cwnd allows

For every ACK during recovery:
update_scoreboard() pipe=(RFC 3517 specification)
transmit ( MAX(0,cwnd-pipe) )
```

Table 4.1: RFC 3517 FR

Table 4.1 briefly presents the SACK-based Fast Recovery algorithm, described in the standard. A TCP sender enters fast recovery upon receiving a *dupthresh* amount of duplicate ACKs that indicate that a packet is lost. The typical value of *dupthresh* is three. The reason for it being three and not, e.g., one or two is because there might have been some reordering in the sequence of the packets and thus it is considered to be a better practice than immediately retransmitting a possibly not lost packet into the network.

After the *dupthresh* value is reached, the sender is entering in Fast Recovery and thus is required to reduce its slow start threshold (*ssthresh*) and its congestion window to half the amount of outstanding data in the network. The sender has afterwards to retransmit the first missing segment. In case the value of the congestion window allows to transmit more data, the sender transmits more data (either more missing data or a retransmission).

For every incoming ACK the sender should update the scoreboard, used for keeping track of the SACK information, compute *pipe* and transmit data in case the *pipe* variable allows it, i.e., if *pipe* falls behind the congestion window variable. *Pipe* holds the senders estimate of the number of packets currently in the network. The calculation is based on the information that the scoreboard holds and RFC 3517 describes the way to compute its value. Fast Recovery ends when all data that was outstanding in the network before entering recovery is acknowledged.

There are two main problems that can be seen with the standard algorithm.

Half RTT silence: As pointed out during the discussion of the Rate Halving algorithm, standard Fast Recovery waits for half of the received ACKs to pass before transmitting anything after the first fast retransmit. This happens, because initially, just after the loss has occurred, *pipe* is typically much higher that the congestion window, since a lot of data was in flight due to a possibly large congestion window. Each returning ACK, reduces the value of *pipe*. A certain number of ACKs is needed in order for *pipe* to fall behind the congestion window and make it possible to transmit data again. This design wastes precious opportunities for data to be sent and may sometimes lead to nothing being sent during recovery. The latter can increase the chances of a timeout occurring.

Aggressive and bursty retransmissions: In case of large variations in *pipe*, the current standard can lead to aggressive and bursty retransmissions. This can happen when burst losses take place, i.e., many segments get lost together. This will lead to a sudden decrease of *pipe*, which will lead to the sender sending a large burst. The more losses there are, the bigger the bursts will be. This means that the sender keeps injecting traffic to a congested network and thus further contributes to its congestion.

Having described the standard Fast Recovery algorithm and its drawbacks, the next section will present the way that this algorithm is implemented in FreeBSD.

4.2.2 Fast Recovery in FreeBSD

The FreeBSD TCP stack implements a version of the standard SACK-based Fast Recovery as described in [BAFW03]. There is one key issue that makes the implementation different than what is described in the standard. There is no implementation of *pipe*. In contrast to that, *pipe* is implemented in the Linux TCP stack. Instead of *pipe* the following workaround is used:

```
pipeBSD = (snd_nxt - snd_fack) + sack_bytes_rexmited
```

where *snd_fack* holds the sequence number of the highest SACKed block and *sack_bytes_rexmited* holds the amount of bytes retransmitted during recovery. This calculation might be good enough but can be inaccurate by underestimating *pipe* in the presence of reordering and thus allowing to inject more data in an already congested network. Table 4.2 describes the Fast Recovery algorithm that is implemented in FreeBSD.

FreeBSD Fast Recovery Algorithm (SACK)

```
On entering recovery:
// ssthresh used during and after recovery
ssthresh = CongContrAlgo()
cwnd = max_seg
// Retransmit first missing segment
fast_retransmit()

For every duplicate ACK during recovery:
update_scoreboard()
cwnd += max_seg
if (cwnd > ssthresh)
          cwnd = ssthresh
transmit ( MAX(0,cwnd-pipeBSD) )

For every partial ACK during recovery:
update_scoreboard()
if(more than 2 max_seg size bytes were acked)
          cwnd += 2*max_seg
else
          cwnd += max_seg
if(cwnd > ssthresh)
          cwnd = ssthresh
transmit ( MAX(0,cwnd-pipeBSD) )
```

Table 4.2: FreeBSD FR

As can be seen from Table 4.2, the FreeBSD implementation does not halve the congestion window when entering recovery but rather resets it to the size of one segment. It then builds it up to the value of *ssthresh*. This approach is similar to the standard and keeps its drawbacks, meaning we still have a half RTT silence and even for a slightly bigger amount of time since it might take some time for the congestion window to grow to the *ssthresh* and meet the downgrading *pipe* variable.

The advantage of this design is that it may avoid bursty retransmissions during the early stages of the recovery process and in the presence of excessive loss events. This problem remains after a certain point when the congestion window grows and converges to the slow start threshold, ssthresh.

4.3 Proportional Rate Reduction

The Proportional Rate Reduction algorithm is designed to overcome both of the problems mentioned in the previous section. Table 4.3 shows a pseudocode version of PRR.

PRR Fast Recovery Algorithm

```
On entering recovery:
ssthresh = CongContrAlgo() // Target cwnd after recovery
prr_delivered = 0 // Total bytes delivered during recovery
prr_out = 0 // Total bytes sent during recovery
RecoverFS = snd_nxt - snd_una // Flightsize at the start of recovery
// Retransmit first missing segment
fast_retransmit()

For every duplicate or partial ACK during recovery:
update_scoreboard()
DeliveredData = change_in(snd_una) + changed_in(SACKd)
prr_delivered += DeliveredData
pipe = (RFC 6675 pipe algorithm)
if (pipe > ssthresh) { // Proportional Rate Reduction
        sndcnt = $\lceil prr\_delivered * ssthresh/RecoverFS \rceil$ - prr_out
} else { // Two versions of the Reduction Bound
        if (conservative) // PRR-CRB
                limit = prr_delivered - prr_out
        else // PRR-SSRB
                limit = MAX(prr_delivered - prr_out, DeliveredData) + MSS
        // Attempt to catch up, as permitted by limit
        sndcnt = MIN(ssthresh - pipe, limit)
}

On every data transmission or retransmission:
prr_out += (data sent) // strictly less than or equal to sndcnt
```

Table 4.3: PRR FR

The PRR algorithm determines the number of segments to be sent during recovery to achieve two goals. First, a speedy and smooth recovery from losses and second ending recovery at a congestion window near the slow start threshold.

PRR avoids excess window adjustments and is thus able to avoid bursts that can cause large fluctuation of the congestion window. The foundation of the algorithm is the Van Jacobsons packet conservation principle, which states that the number of packets (bytes) that are delivered to the receiver, are used as a clock to inject new data into the network [Jac88].

The principle also included the assumption that the lost packets have indeed left the network.

The Proportional Rate Reduction Algorithm has two main parts. The first part, which is called the proportional part, is active when *pipe* is greater than *ssthresh*, which is typically true in the early stages of recovery and under light losses. When this part of the algorithm is active, it gradually reduces the congestion window clocked by the incoming packets, i.e., the congestion window is reduced with every incoming ACK. The second part, named Reduction Bound, of the algorithm is active when *pipe* becomes smaller than *ssthresh*. In that case, the Reduction Bound part attempts to inhibit any further reduction of the congestion window. Instead, it performs slow start and tries to build *pipe* up to the value of *ssthresh*.

PRR is independent of the congestion control algorithm, which is used to determine the new value of *ssthresh*. It takes into account possible modern congestion control algorithms that the stack might be using by allowing them to specify the new value of the congestion window such as Cubic and Vegas [HRX08, BP95].

As can be seen from Table 4.3, PRR introduces several new variables.

- **prr_delivered**: TCP control block variable, that indicates the number of bytes delivered to the receiver during recovery
- **prr_out**: TCP control block variable, that indicates the number of bytes sent to the receiver during recovery, counting both retransmissions and transmissions of new data
- **RecoverFS**: TCP control block variable that indicates the *Flightsize* at the beginning of recovery, defined as *snd_nxt-snd_una* in RFC 3517 [BAFW03]
- **DeliveredData**: Local variable that holds the number of bytes that the current acknowledgment indicates that were delivered to the other side. It is computed as the change in snd.una in case of a partial ACK plus the signed change in the scoreboard information that concerns the bytes that were SACKed by the receiver.
- **sndcnt**: Local variable that indicates how many bytes to send in response to the current acknowledgment

Sndcnt is the variable that holds the outcome of all the calculations performed by the algorithm. It holds the amount of bytes that should be sent in response to each incoming ACK, after having taken into account all the other data. The output function of the TCP stack performs all the necessary checks in order to send *sndcnt* bytes into the network.

Table 4.3 shows how PRR updates *sndcnt* after every ACK received during recovery. When *pipe* is larger than *ssthresh*, PRR spreads the reduction of the congestion window over a full RTT, such that at the end of recovery, *RecoverFS* approaches *prr_delivered*, *prr_out* approaches *ssthresh* and thus attempting to minimize the time spent in recovery. If there exist multiple losses that cause *pipe* to fall below *ssthresh*, the second part of the algorithm is activated, which tries to build *pipe* back to the value of *ssthresh*. This part of the algorithm is based on the *prr_delivered-prr_out* difference. It performs just like TCP is behaving during slow start, by additively increasing the congestion window based on the aforementioned difference, which will usually be one segment. The increase to the number of packets in flight during recovery is done more smoothly than RFC 3517, which as discussed earlier would send out *ssthresh-pipe* segments in one burst.

A key difference of PRR in comparison to the standard algorithm is its reliance on the *DeliveredData* variable that holds the value of the number of bytes that are actually delivered to the other side. Standard algorithms rely on incoming ACKs in order to adjust the value of the congestion window rather than the amount of data that was actually delivered. The approach that PRR is using is making the algorithm actually more precise to its calculations of how many data to inject into the network. This can be explained with the following example. If there is an ACK loss, the standard algorithm will not know that this ACK ever existed and thus, when the next ACK comes in, will adjust the congestion window using only the latter one, leading to a conservative behavior that is not justified. In contrast to that, PRR will know exactly how many bytes were delivered and will inject back to the network the appropriate amount of bytes through *sndcnt*.

The Proportional Rate Reduction algorithm overcomes the two main drawbacks of the standard algorithm. It addresses the Half RTT silence by sending data every other incoming ACK during that period (and not exactly every second ACK like the rate halving algorithm is doing), thanks to the Proportional part of the algorithm and the way sndcnt is calculated during that period. PRR also addresses the second issue by avoiding excess window adjustments and behaving like TCP does when in slow start mode and thus avoiding bursts of multiple segments like RFC 3517 can send (up to *ssthresh-pipe* in case of multiple loss events.

4.3.1 PRR Algorithm Behavior

In order to better understand the behavior of the algorithm the following time sequence graphs are provided. The example illustrates that PRR avoids the Half RTT silence by injecting packets into the network even when the data on flight are more than the *ssthresh*, using the proportional part of the algorithm.

Figure 4.1 shows the behavior of the standard SACK based Fast Recovery algorithm of FreeBSD. Purple lines indicate the SACK blocks, whereas black lines indicate new data sent. The green line indicates the sequence number line, which is advanced over time, indicating that more data is delivered to the receiver and acknowledged.

A retransmission (red line) is done after receiving the third duplicate ACK, which leads to the Fast Recovery code as described in Section 4.2 being executed. Note that the first ACK carrying a SACK block is not a duplicate and as such is not considered as so by the implementation.

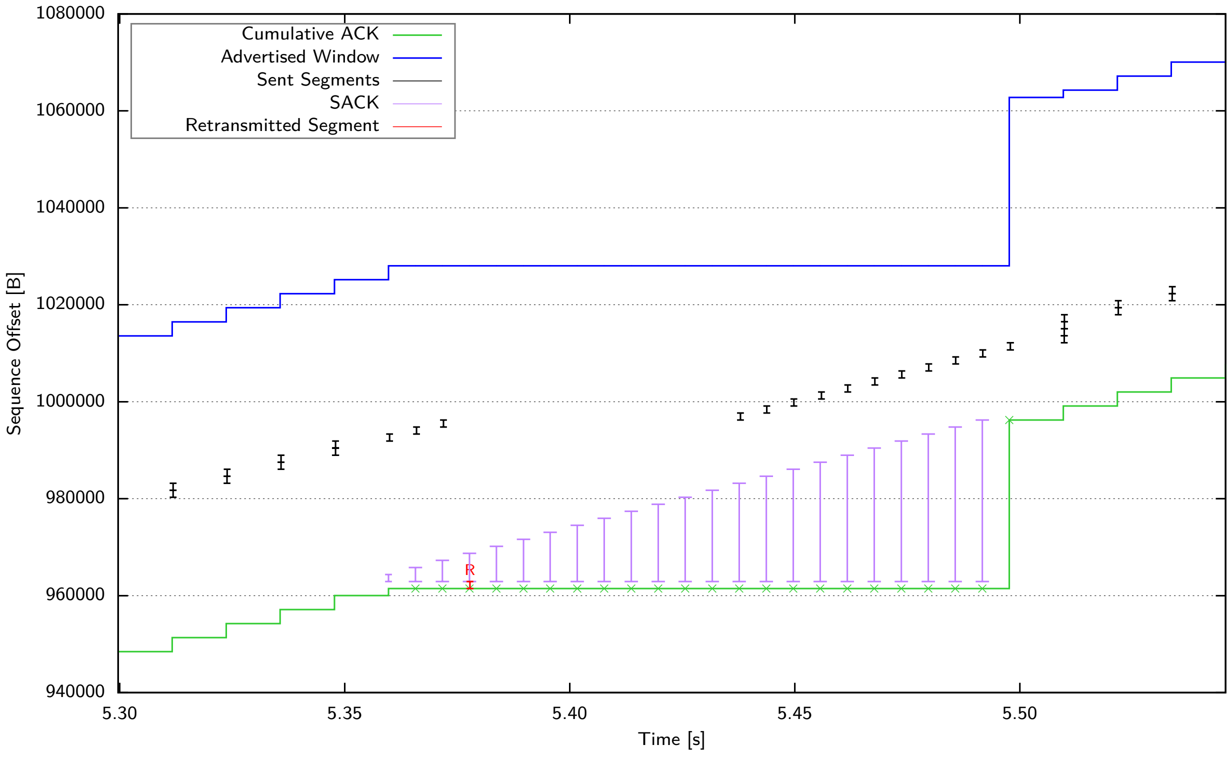

Figure 4.1: FreeBSD Fast Recovery algorithm behavior

It can be observed that the half RTT silence is seen, that is the time that the algorithm is waiting for enough data to be delivered to the other side in order for it to be able to clock more data into the network. As discussed previously, the returning ACKs act as a clock indicating how many packets (data) are delivered.

In contrast to Figure 4.1, Figure 4.2 shows the behavior of the Proportional Rate Reduction algorithm in the same situation. As described in the previous section, PRR does not wait for half an RTT to clock data into the network but rather, when in the Proportional part, injects data on every other ACK.

This means that no opportunities to send data are wasted and thus PRR is able to recover smoother and faster from a loss. During the Reduction Bound part, the algorithm keeps injecting data into the network with the goal to bring *pipe* up to the value of *ssthresh*. This is one of the properties of PRR that will be discussed in the next section.

4.3.2 PRR Algorithm Properties

This section focuses on discussing some key properties that the Proportional Rate Reduction algorithm exhibits.

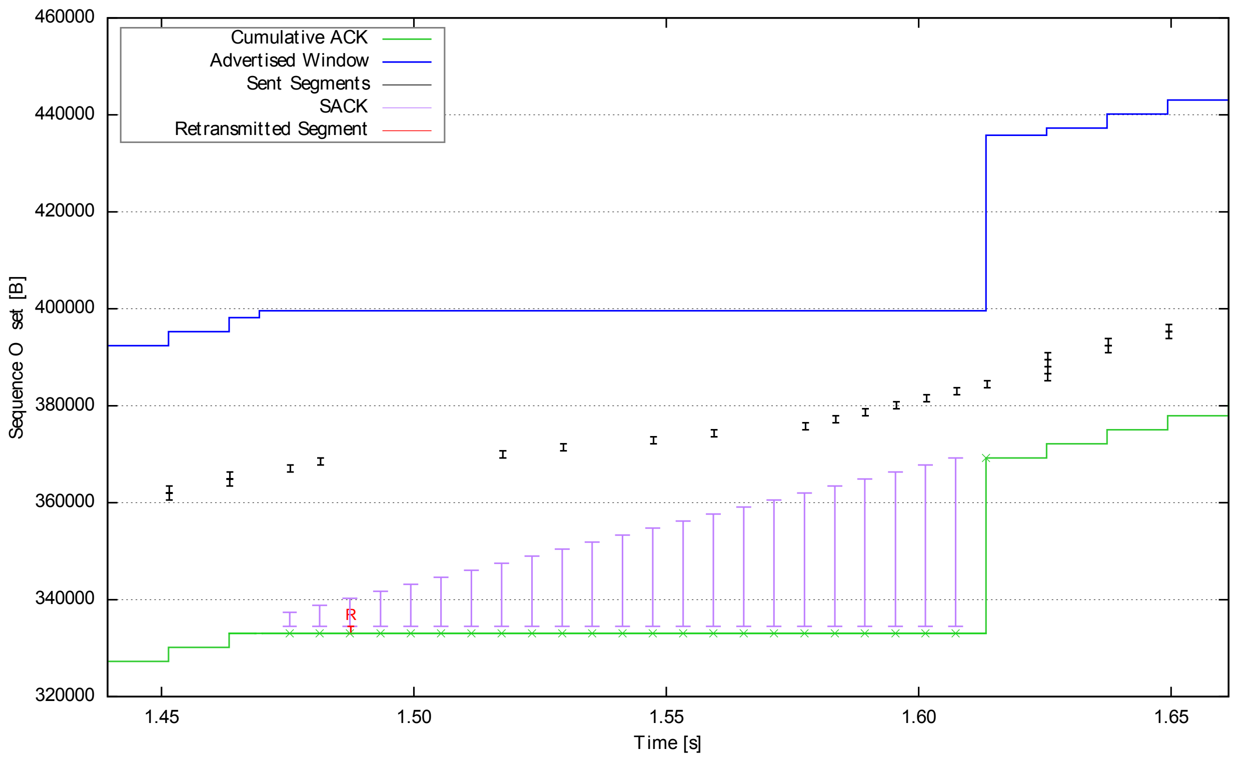

Figure 4.2: PRR Fast Recovery algorithm behavior

- **Maintains ACK clocking**: RFC 3517, as well as the FreeBSD standard implementation can send, under heavy losses, a large burst of data back into the network. This can happen since in the case of a large amount of data being lost, *pipe* will go well below *ssthresh* causing their difference to grow and as a result inject (*ssthresh*-*pipe*) data into the network. This behavior does not occur in PRR primarily due to the slow start part of the algorithm. If a large amount of data is lost, the reduction of *pipe* will cause the slow start part to execute and thus causing less data to be injected to an already congested network. Normally, PRR will spread window reductions out evenly across a full RTT. This has the effect of potentially reducing the burstiness of the internet if PRR is widely used and can be considered to be a type of soft pacing. Hypothetically, any pacing increases the probability that different flows are interleaved, reducing the opportunity for ACK compression and other phenomena that increase traffic burstiness. However, these effects have not been quantified [MDC13].
- **Convergence to *ssthresh***: If there are minimal losses, PRR will cause pipe to converge to exactly the target window chosen by the congestion control algorithm. In such a case, the algorithm will typically operate in the proportional mode, where the congestion window will gradually decrement up to the point that *pipe* reaches the value of *ssthresh* at which point the second part of the algorithm maintains the pipe

at *ssthresh* value. Note that as TCP approaches the end of recovery, *prr_delivered* will typically approach *RecoverFS* and *sndcnt* will be computed such that *prr_out* approaches *ssthresh*. In the case of burst losses, this property may not always hold true since there may not be enough ACKs to raise *pipe* up to *ssthresh*. Note that earlier voluntary window reductions can be undone by sending extra segments in response to ACKs arriving later during recovery.

- **Banks sending opportunities during application stalls**: If an application stalls during recovery, i.e., the sending application is not able to write data fast enough to the socket. PRR stores this missed sending opportunities and sends them at a later point. During application stalls, no or little data is sent, causing *prr_out* to fall behind *prr_delivered* and thus results to their difference *prr_delivered-prr_out* becoming larger leading to sending more data at a latter point. This burst is bounded by *prr_delivered-prr_out+max_seg*. RFC 3517 and the FreeBSD implementation also bank sending opportunities through the difference *ssthresh-pipe*. However, these banked opportunities are subject to the inaccuracies of pipe. *Pipe* is only an estimate of the data in the network and can therefore be inaccurate, e.g., in the presence of reordering. FreeBSDs workaround on *pipe* is also subject to inaccuracies as discussed above.
- **Robustness of DeliveredData**: One of the most important parts of the PRR algorithm is its reliance on the newly introduced DeliveredData variable that holds the amount of data that have been acknowledged from the receiver at each incoming ACK. Use of the SACK information allows the sender to compute the exact amount of data that was delivered to the receiver at each duplicate or partial ACK and thus allowing to send the appropriate amount of data depending on the actual congestion seen at the network. In contrast to that, RFC 3517 and the FreeBSD implementation, rely on the incoming ACKs and *pipe* in order to adjust the congestion window during recovery. This can lead to false estimates of the network congestion due to lost ACKs, reordering in the ACK sequence etc.

4.3.3 Implementation Choices

For the needs of this Thesis, the Proportional Rate Reduction Algorithm was implemented in the TCP stack of a real operating system, which was FreeBSD.

In order to implement the algorithm as part of the TCP/IP stack of the FreeBSD operating system, a number of implementation choices were made.

Firstly, FreeBSD does not implement *pipe*, which is a key part of the PRR algorithm. Instead of *pipe*, a workaround is used in order to calculate the amount of data that is on flight. As already discussed, the following line of code shows the workaround that it is used:

```
pipeBSD = (snd_nxt - snd_fack) + sack_bytes_rexmited
```

where *snd_fack* holds the highest (rightmost) byte that has been SACKed and *sack_bytes_rexmited*, holds the number of bytes that were retransmitted during this recovery episode. This calculation is not near the RFC 3517 standard calculation of *pipe*. The main problem with it is that it can underestimate *pipe* in the presence of reordering and thus introduce more traffic than it should to the network. Nevertheless, since there is no alternative to that in FreeBSD, and the scope of the thesis did not included implementing *pipe*, the above calculation was used in order to compute it and find out which part of the PRR algorithm should be used each time.

The second choice that was made refers to the *sndcnt* variable. RFC 6937 specifies that sndcnt is exactly the number of bytes that should be sent in response to each incoming ACK. The fact that FreeBSD is a byte based TCP stack, unlike Linux that is packet based, leads to a small confusion. Sending packets that contain less bytes than the maximum segment, does in fact increase the latency against standard TCP since the same processing time is needed for extra packets that carry less bytes and thus leading to a smaller Goodput. PRR was implemented, keeping in mind, that it is a better practice to send full size than smaller packets. The implementation, only injects data into the network when *sndcnt* holds a larger value than the maximum segment size that was discovered for the path that TCP operates on.

4.4 Measurements

Various experiments were performed in order to evaluate the behavior of PRR against the standard SACK-based Fast Recovery algorithm of FreeBSD. For this reason, an implementation of PRR was done in the FreeBSD kernel. In order to correctly implement the algorithm in FreeBSD, some implementation choices have been made, that have already been discussed in Section 4.4. The experiment environment and setup are presented in Chapter 3. During the experiments, the IW (Initial Window) of TCP was set to ten (10) segments as proposed by [CDCM13].

There are two metrics that were evaluated during the experiments. The first and most important one, since it reflects what the user experiences, is the application level latency during the transfers meaning the time that is needed for a transmission to finish. In order to quantify the improvement, transmission time was used as a metric. PRR was found to perform better (0%-9%) in all the cases where loss was present. This comes as a result of the fact that PRR reduces (2%) the second metric that was used, the average time that a flow spends in recovery.

Figure 4.3 shows a comparison of the CDFs of the transmission time needed for a 1MB file to be transmitted from the sender to the receiver in the environment described in Chapter 3. A CDF shows the probability that a value x (x-axis) will be found at a value less than

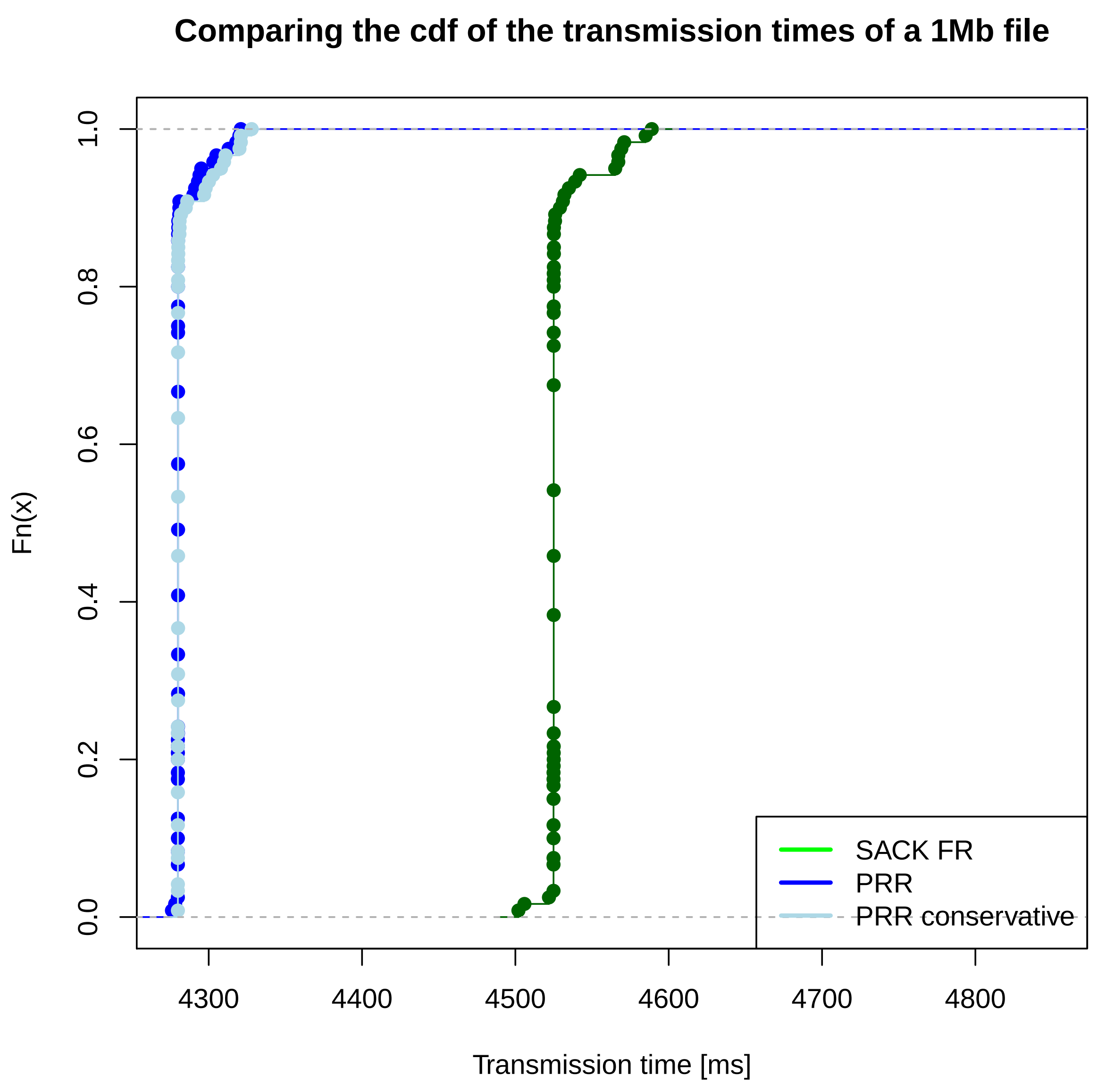

Figure 4.3: Comparing the CDF of the transmission times of a 1MB file

or equal to y (y-axis). We can see that PRR performs always better (4.6%-7.2%) than the standard SACK-based Fast Recovery algorithm of the FreeBSD TCP/IP stack.

Table 4.4 summarizes the scenarios that were used in order to evaluate the PRR algorithm.

The metric evaluated during all of the above scenarios was the transmission time, that was

Scenario	Bottleneck [Mbit/s]	RTT [ms]	Loss Rate [%]	Transmission size [Bytes]
1	2	40	0	1 kB - 2 MB
2	2	40	0 - 10	1 MB
3	2	10 - 120	0	3 MB
4	1 - 35	40	0	3 MB

Table 4.4: Summary of the evaluation scenarios PRR

experienced by the application during the transmissions.

Scenario 1 varies the size of the transmission while keeping constant the RTT, the bottleneck at the intermediate node and introduces no random loss. Figure 4.4 shows the result of scenario 1. The standard deviation of the results is drawn as well. For each measurement point, 20 iterations were run.

As Figure 4.4 shows, both algorithms behave the same as long as the transfer sizes remain small (less than 50kB), because the transfer sizes are not yet big enough in order for the network to introduce many (or any) loss events. As the transfer sizes become bigger, more loss events are present, causing the Fast Recovery algorithms to triger. Both versions of the Reduction Bound of the PRR algorithm perform better than the standard SACK-based Fast Recovery algorithm (0%-3%). It is also observed that there is a very small (0.1%) difference in the performance of the two different reduction bounds of PRR.

The spike observed at the lower end (x-axis value of 15kB) of Figure 4.4 (15kB transfer size) can be explained using Figure 4.5.

As discussed in Chapter 3, when having a 2 Mbit/s bottleneck bandwidth and an RTT of 40ms, the queue size should be equal to 7 segments. A 15 kB transmission amount, will roughly be more than 10 segments (with a 1448 byte MSS). Since we are using an IW of 10 segments, we can see from figure 4.5, that only the first seven packets fit in the queue during the initial burst. The last three are dropped. Since there are no other packets after them, it is not possible for the sender to trigger Fast Recovery by receiving duplicate ACKs and thus a timeout has to occur. The timeout will cause a greater latency introduced and thus the spike is observed. A possible solution to this is the Tail Loss Probe algorithm, proposed by Google, which aims to a faster recovery from tail losses [DCCM13].

Scenario 2 tries to simulate a lossy environment (i.e. a wireless link) where the probability of a random, non-congestion packet loss is high. It varies the loss rate introduced by the intermediate node (1%-10%), while keeping the bandwidth, RTT and transmission size constant. A transmission size of 1 MB was chosen in order to increase the number of possible loss events and thus the number of times that the Fast Recovery algorithms are triggered.

Figure 4.6 shows the results, including the standard deviation. PRR behaves on average better (2% - 9%) than the standard algorithm, although a high deviation is observed in the results. This is expected, since the loss is random and thus the results will vary. 40

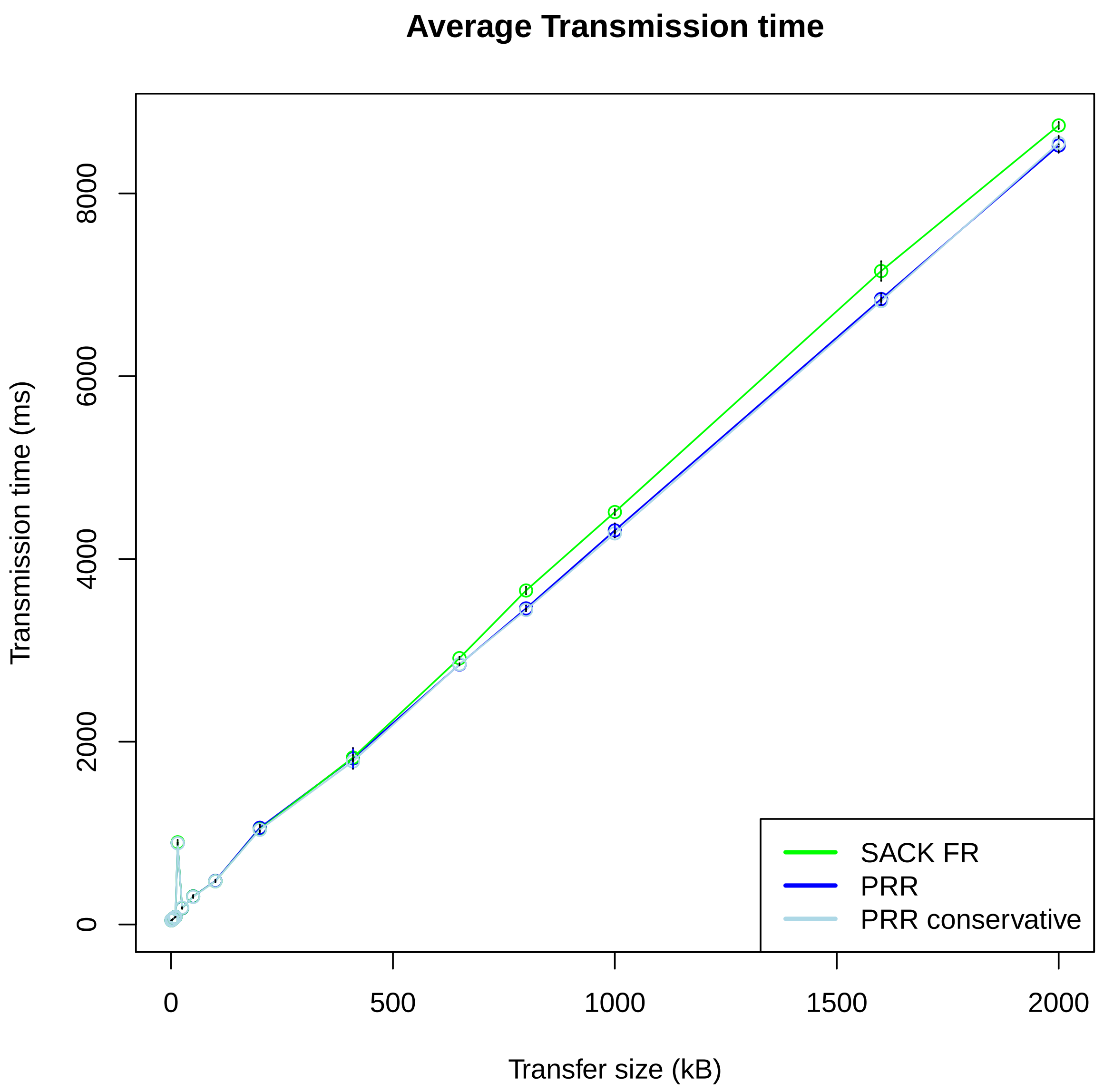

Figure 4.4: Scenario 1 - Average transmission time for varied file sizes

repetitions were run for each measurement point.

Scenario 3 varies the value of the RTT, while keeping the bandwidth and the transmission size constant. A transmission size of 3 MB was used in order to increase the likelihood of loss events occurring and thus the number of times that the Fast Recovery was triggered. This scenario simulates the varied RTT values that a server may encounter in a real scenario.

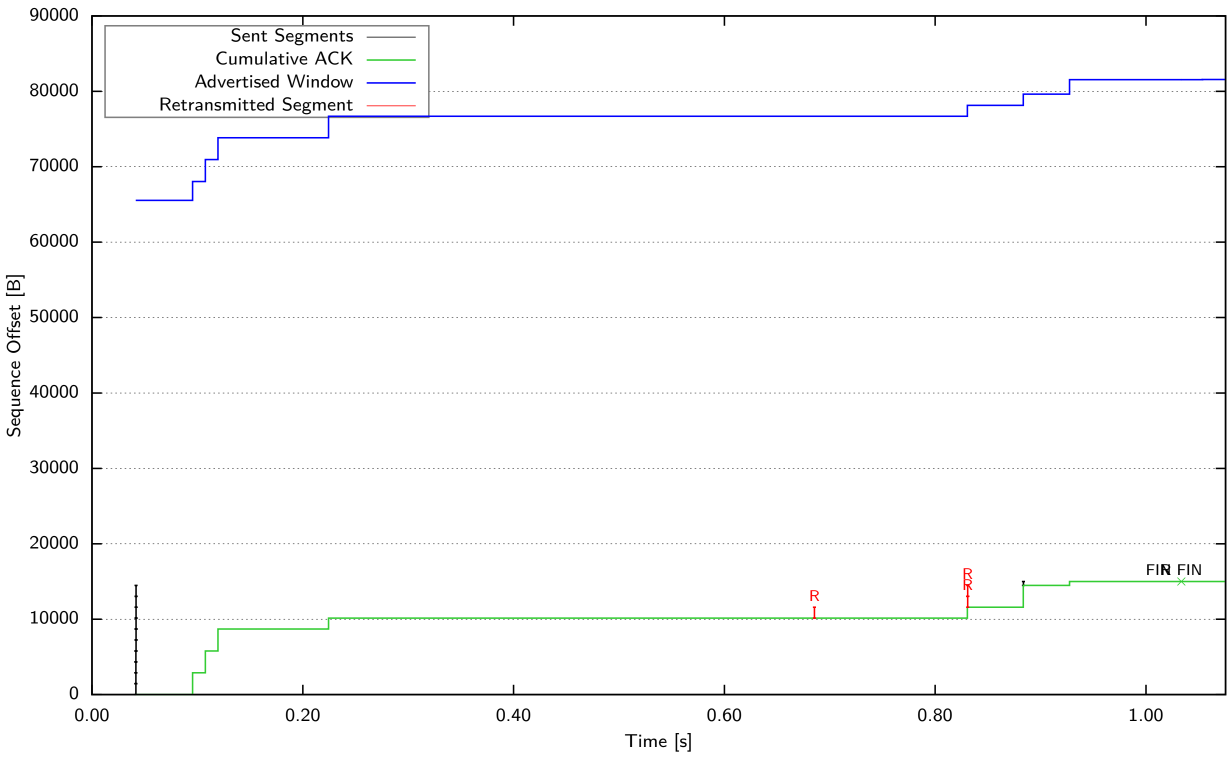

Figure 4.5: Timeout occuring

Figure 4.7 shows the average of 20 runs for each RTT value together with the standard deviation. It is clearly seen that the standard deviation is very small, meaning that the algorithm is always stable, with no large variations in its performance. It can also be seen that both variants of PRR behave better (0.1% - 2.8%) than the standard SACK-based Fast Recovery algorithm of FreeBSD.

Transmission time, for a very small RTT value, here 10ms, is higher than the one that is observed for the rest of the values. This is because the bandwidth-delay product for this case results in a 2 segment queue. This is a quite small value causing many packets to get dropped and thus increasing the transmission time.

The conservative variant of PRR exhibits the same performance as the non-conservative one. It can perform better in certain cases, when not sending an additional packet during the reduction bound part of the algorithm can inhibit another loss event and thus avoiding additional latency. This experiment shows us that the RTT value plays no significant value in evaluating the algorithm.

Scenario 4 varies the bandwidth of the path, while keeping the RTT and the transmission size constant. Twenty runs were performed for each point.

Figure 4.8 shows the results of Scenario 4 along with the standard deviation. As expected,

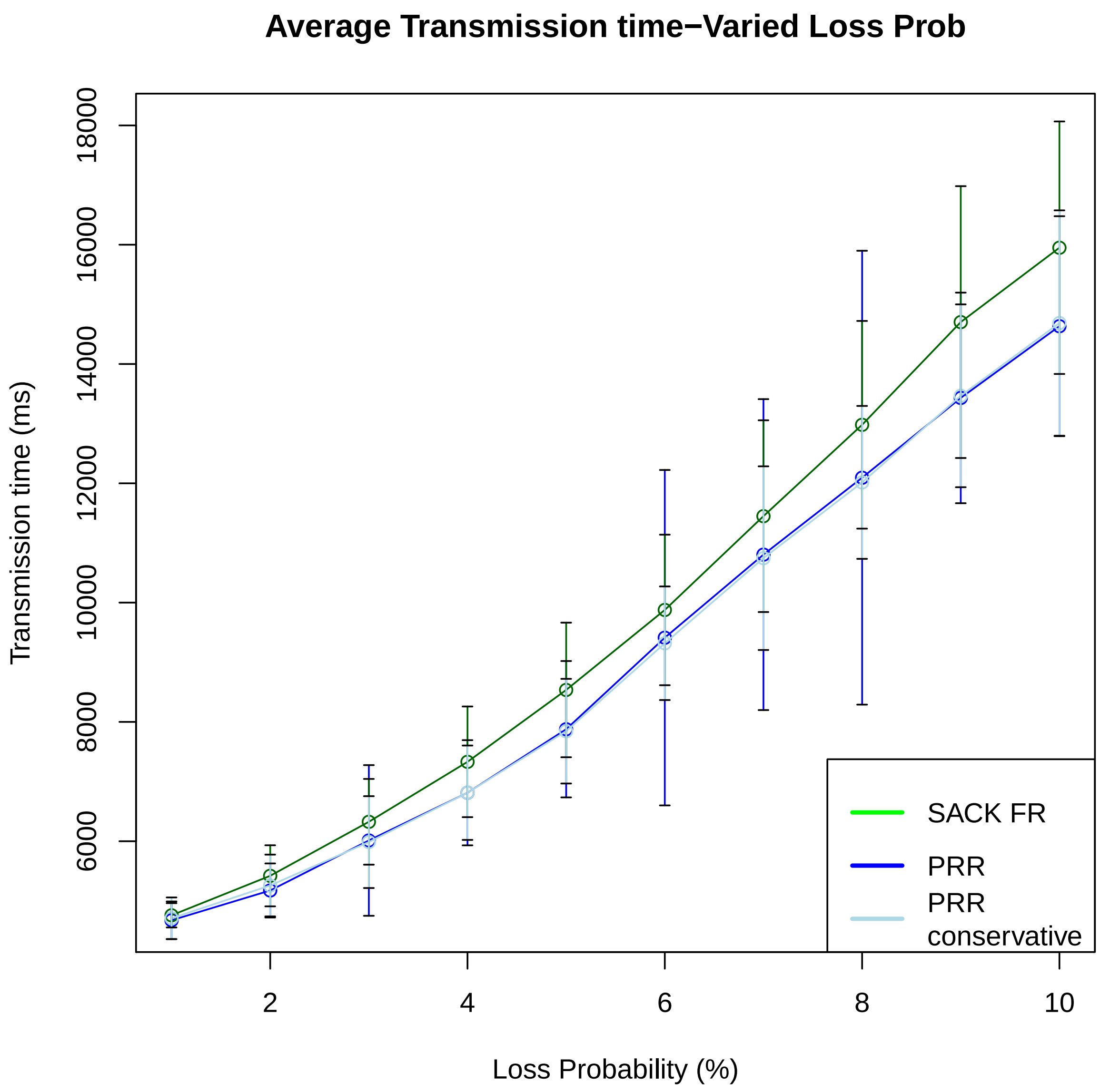

Figure 4.6: Scenario 2 - Varied Probability Loss

PRR lowers the application level latency (0.1%-1.9%) for the low bandwidth topologies, where more loss events occur. It is observed that in the case of 16Mbit/s, the conservative reduction bound of PRR, performs 1% worse than the standard algorithm. This happens because of the conservative nature of this reduction bound. The latter means that choosing not to transmit an extra packet causes a small performance degradation in this particular scenario due to the particular traffic pattern, e.g., less packets transmitted during recovery

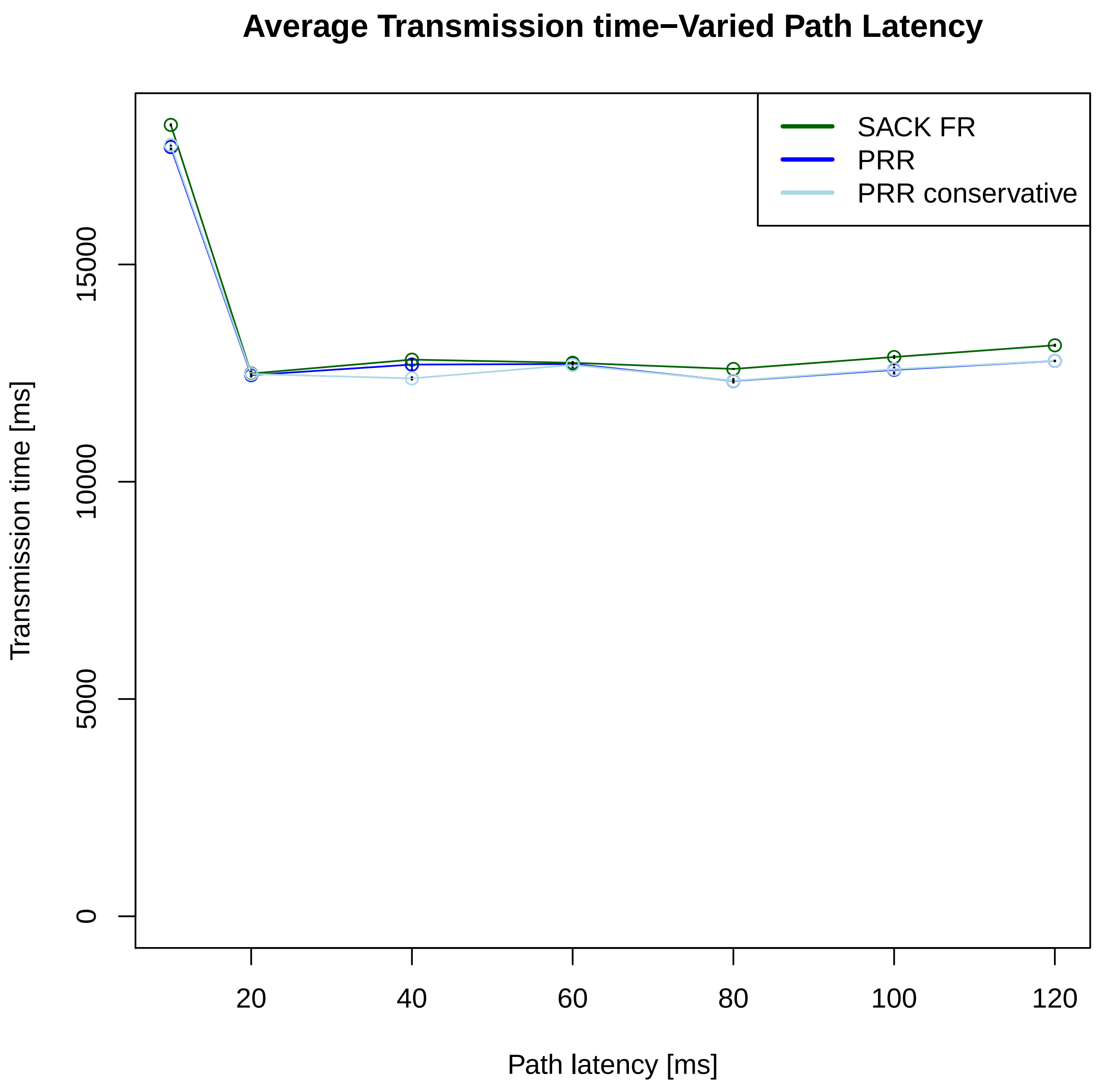

Figure 4.7: Scenario 3 - Varied RTT - Constant Transmission Size

can lengthen the transmission time. The difference in performance for high bandwidths is low due to the fact that higher bandwidth-delay products imply larger queues and thus less lost packets that would trigger the Fast Recovery algorithms.

Having reviewed all 4 scenarios the second metric can now be discussed. The time spent in Recovery metric explains the results observed in this section. This is a kernel observed metric that helps in understanding why the application level latency is on average reduced,

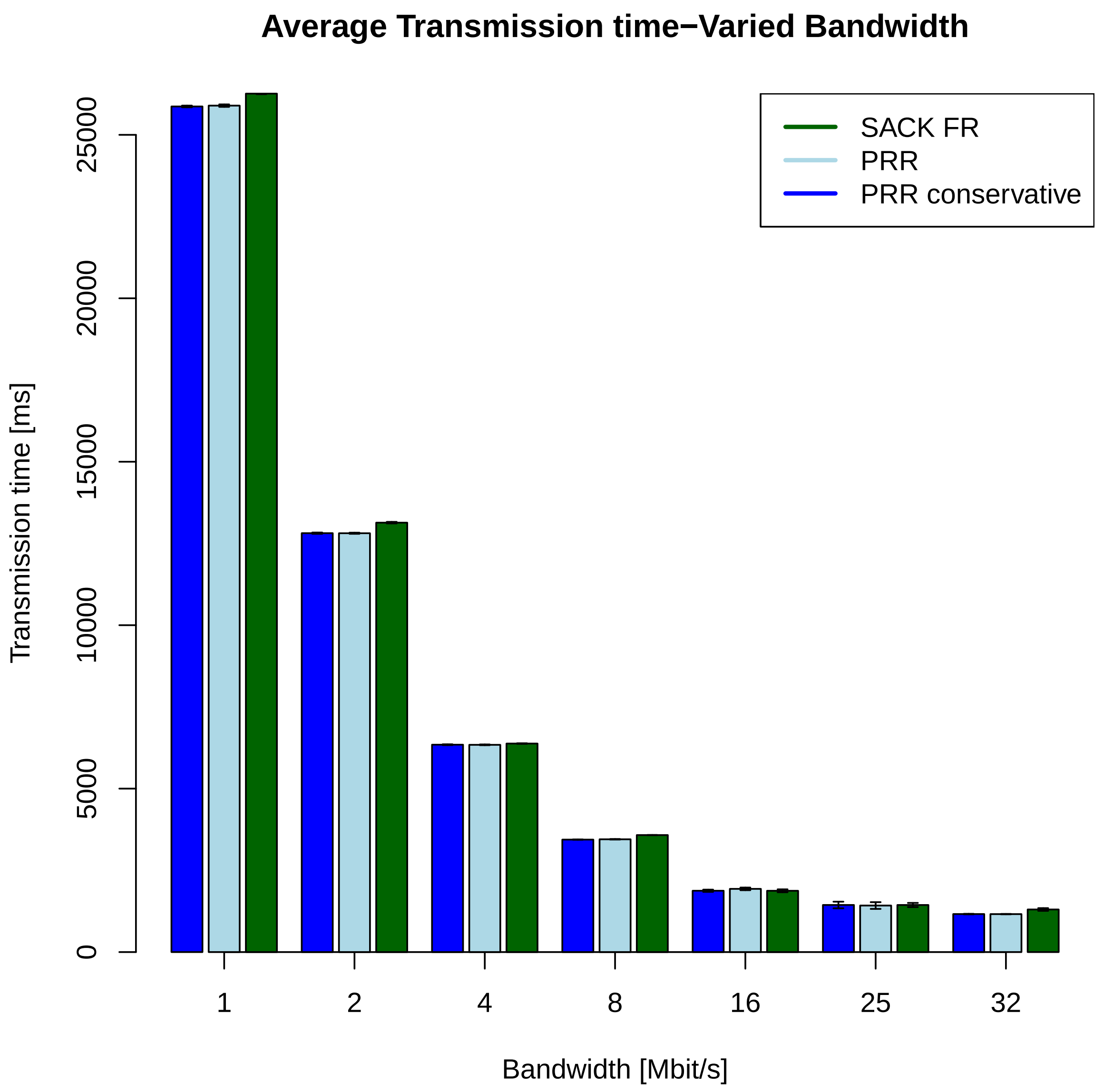

Figure 4.8: Scenario 4 - Varied BW - Constant Transmission Size

when using PRR. The statistics gathered during various experiments, show us that the average time spent in recovery for various runs (data was collected during most of the experiments presented in this section), is reduced by 3% on average. This means that for longer runs, where more loss events are present, having on average a faster recovery, implies a lower transmission time due to the additive effect of PRR on multiple loss recovery events.

4.5 Conclusion

Proportional rate reduction improves fast recovery under practical network conditions. PRR operates smoothly even when losses are heavy, is quick in recovery, and accurate even when acknowledgments are stretched, lost or reordered. In live experiments, PRR was able to reduce the transmission time in various scenarios by 0% to 9% as compared to the FreeBSD standard SACK based Fast Recovery.

Chapter 5

New CWV

New Congestion Window Validation is the second extension that this thesis presents. This method tries to address the issue of enhancing TCP in order to support Rate-Limited Applications efficiently.

Many current Internet applications, such as video streaming, can be characterized as Rate Limited. This means that the sender does not transmit at a rate controlled by the transport layer protocol (such as TCP), but rather at a rate dictated by the application. In bulk transfers, the application will always have data available to transmit and thus the rate at which it transmits to the network, is controlled by the transport layer protocol. In contrast, a rate-limited application might not have always data available to send and thus the writes to the socket may be characterized by short or long pauses. Also even when there are data available to send, the amount might be less than the congestion window, which means that the sending rate will be dictated by the application.

In other words, a rate-limited application may send at a Constant Bit Rate (CBR), less than limited by the transport protocol, or may send with periods of higher (but limited) rate or periods where no data is sent (idle periods). TCP [DBEB06] has been designed and is able to support a range of application behaviors, but TCPs congestion control [APB09] has been mainly optimized with various variants such as Cubic [HRX08], Vegas [BP95] etc. for bulk traffic.

The recent growth of TCP based multimedia applications has reopened the debate on TCP usage for rate-limited applications [BBM+06]. Other rate-limited TCP behaviors include HTTP Adaptive Streaming (HAS), Google SPDY (which uses persistent connections to retrieve multiple objects) and HTTP 1.1 persistent connections.

The New Congestion Window Validation method [FSS13] focuses on such traffic where standard TCP is generally not well adapted. It proposes a set of modifications to the standard algorithm which will enable more effective use of standards-based methods. Standard TCP does not limit the growth of the congestion window during a rate-limited period, which potentially leads to it having a big value that does not reflect the current state of the network path. This can lead to congestion in case data is suddenly available to the application buffer. Standard TCP does also reset the congestion window to a value called restart window, after an idle period, which can lead to decreased performance. New-CWV proposes a freezing of the congestion window during rate limited or idle periods that will stop the additional unrestricted growth of the congestion window and can improve performance.

5.1 Related Work

A similar approach of freezing the congestion window was also proposed by Freeze TCP [GMPG00], however, it was specifically proposed in order to mitigate mobility related disconnections and is not a suitable solution for variable rate congestion control.

TCP-Congestion Window Validation [HPF00] was proposed by the IETF as an experimental standard in RFC 2861. It takes the approach of exponentially decreasing the congestion window during an idle period. To be more specific, according to TCP-cwv, the congestion window is reduced by half each time the connection has been idle for an RTO period. During an application limited period, the recommended proposal is to reduce the congestion window by (cwnd + win_used)/2 for each transmission that does not utilize the full congestion window, where win_used is the estimated proportion of the congestion window that was used. This avoids the value of the congestion window becoming larger than what was previously used.

The problem of TCP-cwv is that in the presence of a long idle period (i.e. several RTO timeouts), this method will reduce the congestion window to the restart window (RW). The application would have to slow start again like in standard TCP. During an application limited period, the performance of TCP-cwv can be lower than that of standard TCP since the latter one is able to send data more aggressively in the case that they might become available at some point during the transfer. TCP-cwv can be therefore viewed as having a conservative behavior.

The New-CWV proposal tries to address the aforementioned issues and propose a design that will truly benefit applications with rate-limited behavior.

5.2 Congestion Control In Modern Operating Systems

Standard TCP uses the congestion window (cwnd) variable, in order to limit the number of bytes or packets that a TCP flow may have in the network at any time. The cwnd starts at a value known as the Initial Window (IW) and is updated by the congestion control algorithm as TCP continuously probes for additional capacity. TCP should also maintain a variable called pipe and described in [BAFW03], that holds an estimate from the senders side, regarding the amount of data that are outstanding in the network. The TCP/IP stack of Linux implements and uses pipe whereas FreeBSD does not and uses a workaround instead. The last important variable that is used is the slow start threshold (ssthresh) which reflects the available path capacity at the time that the last congestion event occurred.

The way that the congestion window is increased or decreased after received ACKs or Loss Events, is dictated by the congestion control algorithm. There are various congestion control algorithms addressed over the years such as NewReno [FHG04], Cubic [HRX08], Vegas [BP95] etc. Each one of them, tries to address a different issue i.e. Cubic is optimized for high bandwidth and high latency networks whereas Vegas uses the increase of the RTT measurements as a sign that a network is becoming more congested. It then tries to avoid loss events from happening by reducing the congestion window appropriately.

FreeBSD as well as Linux have various congestion control algorithms implemented, which can be used depending on the situation (i.e. topology, latency, bandwidth etc) to maximize throughput and minimize the latency experienced by the user.

All of these algorithms are mostly targeted to bulk applications, such as FTP, where continuous data is available at the sender. Experience shows that they perform well by limiting transmission to an order that reflects the fair share of the capacity of the path that is used. However, they fail to address applications that don't use the entire congestion window and may cause problems. Such an application may not have data available to send and can also experience periods of idle times, meaning that there is no data available to send.

Standard TCP does not impose any limitation to the growth of the congestion window when an application is rate limited. A rate limited application may then as a result, grow a congestion window far beyond the corresponding transmit rate, resulting into a value that does not reflect current information about the state of the network path that the flow is using. This is because the cwnd will be increased due to the received ACKs resulting into an arbitrarily large value. However, when the packets are sent along the path with a rate lower than what the cwnd allows, the reception of an ACK does not provide evidence that the network path was able to sustain the transmission rate reflected by cwnd. This leads to the cwnd becoming a poor estimate of the available path capacity. Severe congestion would occur if a rate limited application was to increase its transmit rate while having such an invalid cwnd. The latter may result in reduced application performance and could significantly contribute to network congestion.

The problem is highlighted with the following example. If an application transmits with a certain rate, less than the congestion window, the latter can then grow significantly due to the received ACKs, as discussed before, and thus reach a large value, not reflecting the current state of the path. In the case that data becomes available (i.e. transmit rate is suddenly increased) such that they can fill the whole of the congestion window, a big burst will be sent possibly leading to a congestion of the network and/or additional losses.

Another issue that can be observed in the currently implemented congestion control algorithms, is their behavior after an idle period, i.e. a period that there is no data available to send. Standard TCP dictates that when an application is idle for a period greater than the current Retransmission Timeout (RTO), the congestion window is reset to no more than the Restart Window (RW) [APB09]. While this behavior may be desired for bulk applications, it is not suitable for multimedia rate limited traffic, such as youtube, which has a bursty behavior followed by pauses.

The New-CWV method tries to address the aforementioned issues with a way that is described in the next section.

5.3 New-CWV

This section presents the New Congestion Window Validation (New-CWV) [FSS13], a set of proposed modifications that aim into overcoming the limitations that the Standard Algorithm and the TCP-CWV exhibit. The new method allows the connection to preserve the congestion window every time that a rate limited or idle period is experienced. The connection preserves the congestion window for a limited amount of time called the Non-validated period (NVP). The period where actual usage is less than allowed by the congestion window, is called Non-validated phase. In contrast, the period that usage reflects the estimate of the path, is called Validated phase. The technique of freezing the window, allows a sender to resume transmission at a preserved rate, without incurring the delay of slow start. If the sender experiences congestion, it is immediately required to reduce the congestion window to a value specified by the method. In case that the sender does not make use of the available window for a period more than the NVP, the sender is required to reduce the congestion window to an appropriate value, specified by the method. The value of the NVP is set to be 5 minutes.

The method introduces the following new terminology:

- **pipeACK sample**: A measure of the volume of data that were acknowledged by the network per RTT.
- **pipeACK**: A variable that records the volume of data that were acknowledged by the network within an RTT using the pipeACK samples.
- **pipeACK sampling period**: The maximum period that a pipeACK sample can influence the measurement of *pipeACK*.
- **Non-validated phase**: A phase that the sender enters when the usage reflects a

previous measurement of the available path capacity. A connection is in the Validated phase when *pipeACK* is less than half the congestion window, e.g., less than half of the amount of data that the congestion window permits is not used.

- **Validated phase**: A phase that the sender enters when the congestion window, reflects a current estimate of the actual path capacity. In other words a sender is in the Validated phase when pipeACK becomes more than half the current congestion window.
- **Non validated period**: The maximum amount of time that the sender is allowed to stay in the Non validated phase.
- **Rate limited**: In the context of New-CWV, a rate-limited flow is one that does not consume more than half of the congestion window and thus operates in the Non-validated phase.

New-CWV does not differentiate between periods where the application is rate limited and periods that it is idle, meaning that both behaviors are treated as being rate limited in the context that was defined previously.

New-CWV fulfills the requirements of rate-limited applications in contrast to CWV whose drawbacks are discussed in the previous section. It can also keep applications from sending data simply for preserving the congestion control state, e.g., not resuming with a restart window (RW). This behavior is known as padding.

5.3.1 Calculation of pipeACK

Pipe and *FlighSize* are two variables defined by the standards [APB09, BAFW03], that are used to indicate the amount of data present in the network. If there is no loss present, *Pipe* and *FlighSize* are assumed to be equal. In modern Fast Recovery algorithms, like PRR, described in the previous section as well as the standard SACK-based algorithm, *Pipe* is used during loss events leading to a faster recovery.

In New-CWV, *pipeACK* is used in order to measure the number of bytes acknowledged by the network per RTT. It is important to note that New-CWV does not use *pipeACK* during loss events but rather the default algorithms are used. The new variable is used in order to determine if the sender makes appropriate use of the congestion window, e.g., to determine if the application is rate limited or not.

A sender determines a *pipeACK* sample by measuring the volume of data acknowledged by the network per RTT. A possible way of doing that is caching the highest sequence number of a packet sent and assign the difference between the cached value and the current one to *pipeACK* when the cached value is acknowledged by the receiver. A sender should perform a *pipeACK* measurement at least once per RTT.

When no measurements are available, *pipeACK* is set to the maximum value, e.g., a value that will inhibit the connection from entering the Non-validated phase and thus restraining the growth of the congestion window.

5.3.2 Method description

A connection is initialized as being in the Validated phase. That means that *pipeACK* is set to the maximum, depending on the implementation, value in order not to inhibit any congestion window increase during the initial part.

When using New-CWV, a TCP connection may be in one of the two phases discussed in the previous section. This is introduced in order to be able to determine if a sender is using the whole of the congestion window, e.g., if it reflects the application rate or if the congestion window is growing to a value that does not reflect the current rate, leading to potential bursts that may cause congestion. The Validated and Non-validated phases are described as follows:

- **Validated phase**: A sender is in the Validated phase when *pipeACK* is greater or equal than $1/2 \times cwnd$. This is the normal phase for a connection, meaning that most of the congestion window is used in order to transmit data and thus the congestion window reflects the rate that the application is injecting data into the network. During this phase, the standard congestion control algorithm is used for increasing the congestion window.
- **Non-validated phase**: A sender is in the Non-validated phase when *pipeACK* is less than $1/2 \times cwnd$. This is the phase where the congestion window has a value that is not being used by the sender, e.g., the amount of data that the application has available for sending in the socket is far less than what is permitted by the congestion window. During this phase, the congestion window is not allowed to increase, e.g, it freezes. This is done in order to inhibit further increase to its value, leading potentially to congestion.

As discussed, when in the Validated phase, a connection operates normally, meaning that the behavior is the same with a connection that does not use New-CWV. The behavior during the Non-validated phase is describes as follows:

- The congestion window is not increased when ACK packets are received.
- When the sender receives an indication of congestion while in the Non-validated phase, such as detects loss or an ECN mark, it then must exit the Non-validated phase reducing the congestion window to a value discussed in the next section.
- In case of the Retransmission Time Out (RTO) expiring, the sender must then exit the Non-validated phase and resume by using the standard mechanism described in [APB09]. In the case of an RTO expiring, the reducing of the congestion window

that will occur by the standard method is considered appropriate since any path history that may have accumulated is considered unreliable.

- When a sender is in the Non-validated phase and measures a *pipeACK* value greater than half of the current value of the congestion window, it should then enter the Validated phase. Note that a rate-limited sender will typically not be affected by whether the sender is in the Validated or Non-validated phase since it will not use the whole of the congestion window. The transition will however release the sender from restricting the growth of the congestion window and restore the use of standard congestion control methods.

Reception of congestion indication (i.e. Loss, ECN) while in the Non-validated phase means that it was inappropriate to use the preserved value of the congestion window after a rate-limited or idle period. The sender should therefore select a new value for the congestion window based on the utilized value. In order to do that, the sender must record the current *FlightSize* [APB09] in the variable *LossFlightSize* and use it as follows:

```
cwnd = Min(cwnd/2,Max(pipeACK,LossFlightSize)
```

The method chooses this value because the nonvalidated congestion window may be much larger than the actual *FlightSize* or the one recently used, reflected by the value of *pipeACK*. The updated congestion window therefore prevents overshoot by a sender significantly increasing its transmission rate during the recovery period. Note that New-CWV does not specify a method to be used during Fast Recovery, therefore the standard algorithm is used.

After the recovery finishes, New-CWV dictates that the TCP sender must reset the congestion window to the following value:

```
cwnd = ((LossFlightSize - R)/2)
```

R holds the amount of data that were retransmitted during recovery. The inclusion of the term R, makes the adjustment more conservative than standard TCP [APB09]. This is needed since the sender may have sent more segments than what Standard TCP would have sent. If ECN is used, the congestion window should further be reduced by the number of ECN marked packets that have been received. The sender should also reinitialize *pipeACK* to the maximum value since any accumulated path history reflected in the value of *pipeACK* is considered as unreliable, after a loss event.

An application that remains in the Non-validated phase for a time period greater than the NVP, is required to adjust its congestion control state, by adjusting the value of *ssthresh* as follows:

```
ssthresh = max(ssthresh, 3*cwnd/4)
```

This adjustment ensures that the sender has safely sustained the present rate for an NVP period. Resetting *ssthresh* at the conclusion of NVP, allows a more rapid (exponential) growth towards the previous congestion window should the application start sending at a higher rate again. When exiting the Non-validated phase, the sender should also adjust the congestion window to:

```
cwnd = max(1/2*cwnd, IW)
```

This adjustment ensures that the sender appropriately reduces the congestion window value to better reflect the rate that was used most recently used.

Limiting the amount of time that an application is allowed to be in the Non-validated phase avoids undesirable side effects that could occur in case of having the congestion window unnecessarily high for a big amount of time, such as sudden bursts due to data becoming available. The value of the NVP is five minutes. This value was chosen because it is larger than the idle intervals of most common applications, but not sufficiently larger than the period for which the capacity of an Internet path may commonly be regarded as stable [FSS13]. Another reason for using a five-minute value for the NVP is that, also other TCP sender mechanisms have used such an interval, e.g., the default user timeout of 5 minutes [Pos81] which controls the amount of time that transmitted data can remain unacknowledged in the network before a connection is forcefully closed.

5.3.3 New-CWV behavior

As described, New-CWV freezes the congestion window of a sender that does not use the whole of the rate that is implied by the congestion window. It also defines two phases that the sender might be into. This allows a connection to faster restart after an idle time while the sending of big bursts is controlled. This section shows, through time-sequence graphs, the way that New-CWV functions.

Figure 5.1 depicts a typical bursty youtube flow [GCJM12], which is sent by using the standard TCP congestion control methods. Since there is not always data available to send, transmission is bursty with idle times. Standard TCP congestion control restarts at a RW after an idle period that is typically less than the one before the idle period. This can cause great latency for applications like youtube that have a bursty behavior with idle times. The connection has to get through slow start every time that there is new data available to send even if the path conditions have not changed at all, e.g., no congestion is introduced.

This problem is not present when using the new method. As seen in Figure 5.2, New-CWV preserves, as described, the congestion window during the flows.

The first burst has the same behavior as the standard TCP. This is to be expected since initially there is typicially no difference between the two methods since *pipeACK* is initialized to the maximum possible value in an effort for the sender to begin transmitting in the Validated phase. The burst following the first burst will continue transmitting with a preserved congestion window that is definitely greater than the RW that the flow with standard TCP uses. The effect can more clearly be seen in the third burst where data is

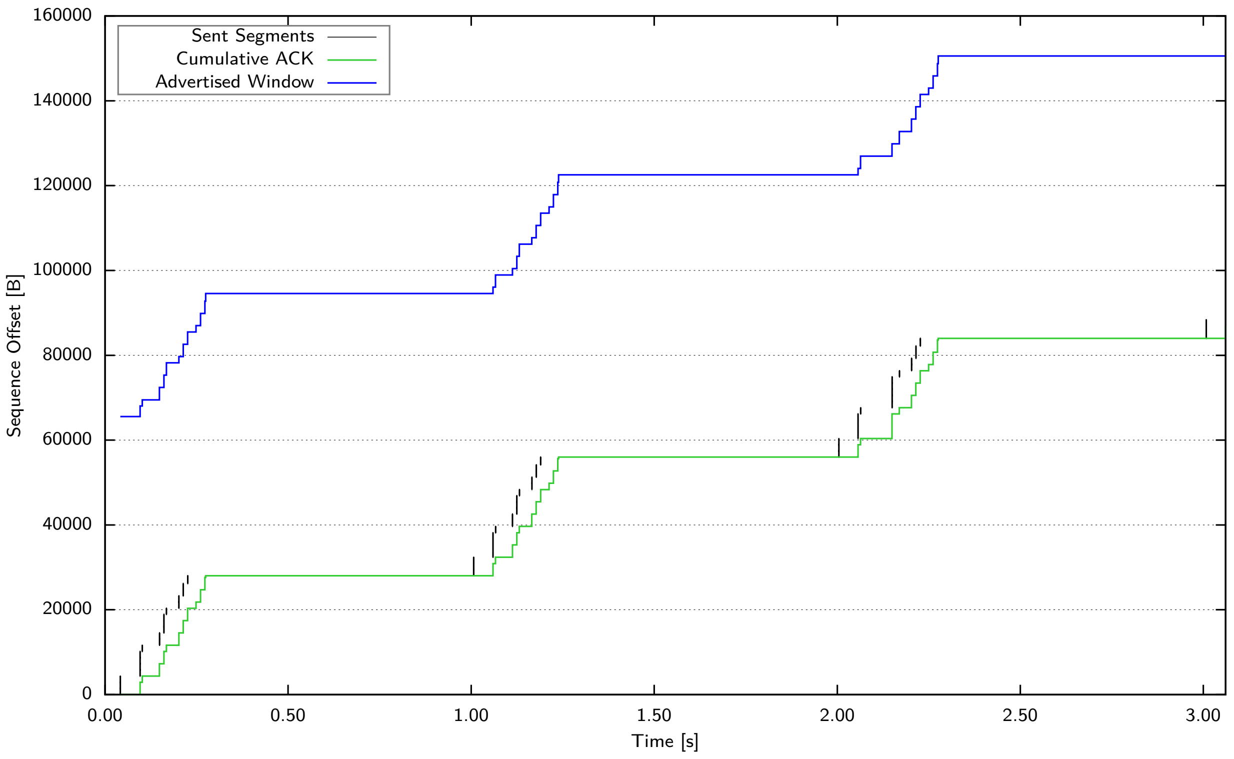

Figure 5.1: Bursty youtube application flow with Standard TCP

being transmitted with an even larger congestion window than the second burst since it is still preserved.

The technique of preserving the congestion window for idle/rate-limited periods much longer than allowed by the Standard TCP is clearly effective for uncongested scenarios. In the measurement section, it is shown that the choices made in regards to the post recovery value of the congestion window, do also have a significant effect in a congested network.

5.3.4 Implementation Choices

For the needs of this Thesis, the New-CWV method was implemented in the TCP stack of a real operating system, which was FreeBSD.

In order to implement the method as part of the TCP/IP stack of FreeBSD, a number of implementation choices were made.

The first implementation choice is that the method was not implemented as a congestion control algorithm. This choice allows the user to use the method in conjuction with various different congestion control algorithms (e.g. CUBIC, Vegas) in order to control the growth

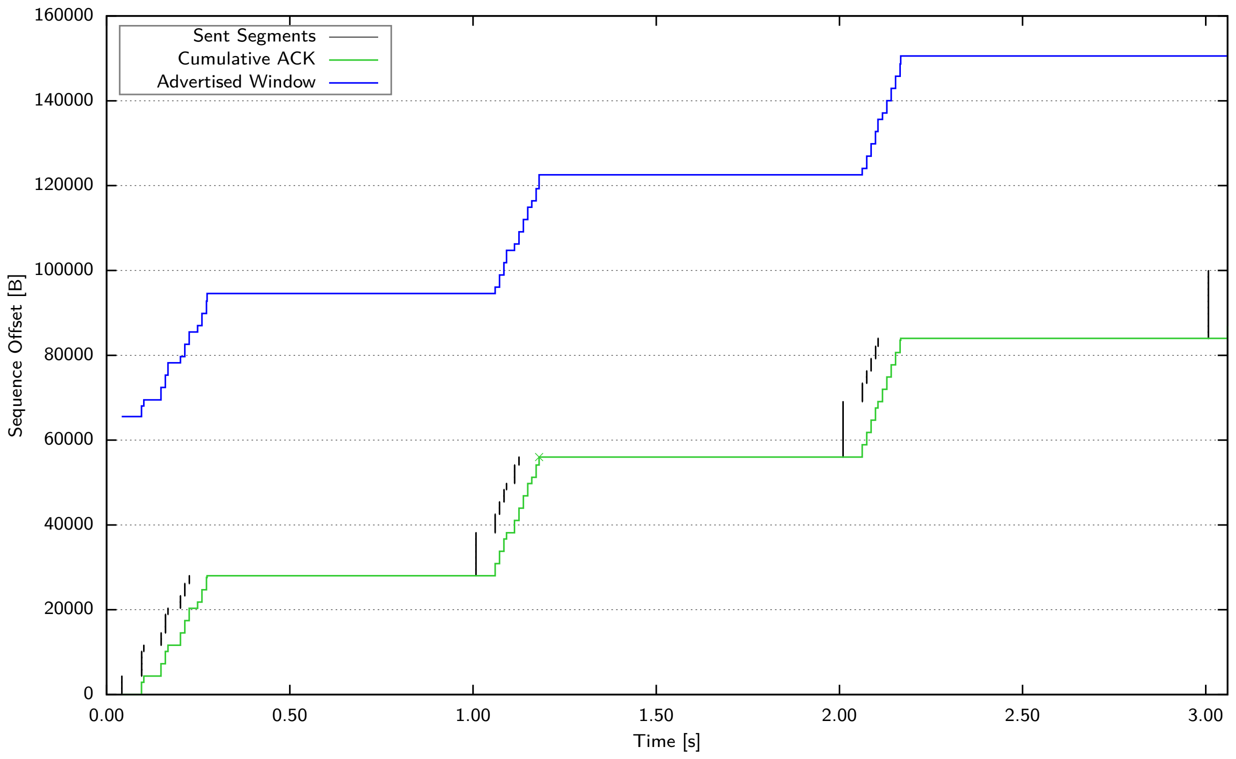

Figure 5.2: Bursty youtube application flow with New-CWV

of the congestion window and thus not being limited by the default algorithm which is NewReno.

The second implementation choice, is in regards with the measurement of *pipeACK*. This is the most important variable introduced by the method and is used to compute the phase that the sender currently lies in. In order to compute *pipeACK* the approach suggested by the Internet Draft (ID) is followed [FSS13]. During initialization, *pipeACK* is set to the maximum possible value. A helper variable *prevHighACK* is introduced that is initialized to the initial sequence number (ISN). *prevHighACK* holds the value of the highest acknowledged byte so far. *pipeACK* is measured once per RTT meaning that when an ACK covering *prevHighACK* is received, *pipeACK* becomes the difference between the current ACK and *prevHighACK*. This is called a *pipeACK* sample. A newer version of the draft suggests that multiple *pipeACK* samples can be used during the *pipeACK* sampling period [FSS13] in order to achieve a more robust calculation.

5.4 Measurements

Two experiments were performed in order to evaluate the performance boost that New-CWV offers for rate-limited applications against Standard TCP congestion control. The implementation of New-CWV in FreeBSD, discussed in the previous chapter, was used in order to evaluate the method. In order to evaluate the performace of both extensions that this thesis addresses when used together, both experiments were also run with PRR instead of the standard SACK based Fast Recovery of FreeBSD. The experiment environment and setup were presented in chapter 3.

The traffic that was used for the measurements, is similar to the traffic that a video streaming application like Youtube exhibits. Youtube shows a bursty behavior meaning that the data to send are available in bursts that are nominally around 64kB. Between those bursts long or short idle times are present [GCJM12] meaning that Standard TCP uses the Restart Window every time that a new burst is available to send. This type of traffic is suitable in order to depict the advantage that New-CWV offers to such rate-limited applications.

Scenario	Bottleneck [Mbit/s]	RTT [ms]	Application Rate [kBytes/s]	No of flows
1	20	40	10 - 1200	1
2	15	40	64	1 - 10

Table 5.1: Summary of the evaluation scenarios New-CWV

During the experiments, the transmission time needed for each block of data (burst) was measured.

Two experiments were done in order to evaluate the method behavior in the case of a congested and a non-congested environment. Table 5.1 summarizes the scenarios that were used.

Figure 5.3 shows the results for the first scenario. This scenario focuses on an environment where only one flow is running. The method is able to fully operate as designed with little or no congestion. The bottleneck of the channel is 20 Mbit/s and the application rate varies from 10kB/s to 1200kB/s.

Note that application rate means that a certain number of bytes is available at the socket at the beginning of every second for TCP to transmit. For example, an application rate of 64kB/s in this context means that at the beginning of every second, 64kBytes are available at the socket buffer for TCP to transmit. After the transmission of the burst (64kBytes here) is completed, the time needed to successfully do it is measured and reported. Each measurement point in scenario 1 represents a flow of 60 seconds meaning that the average time needed for 60 64kB bursts is reported. There were 50 such one minutes flows performed for each point. Figure 5.3 depicts the result.

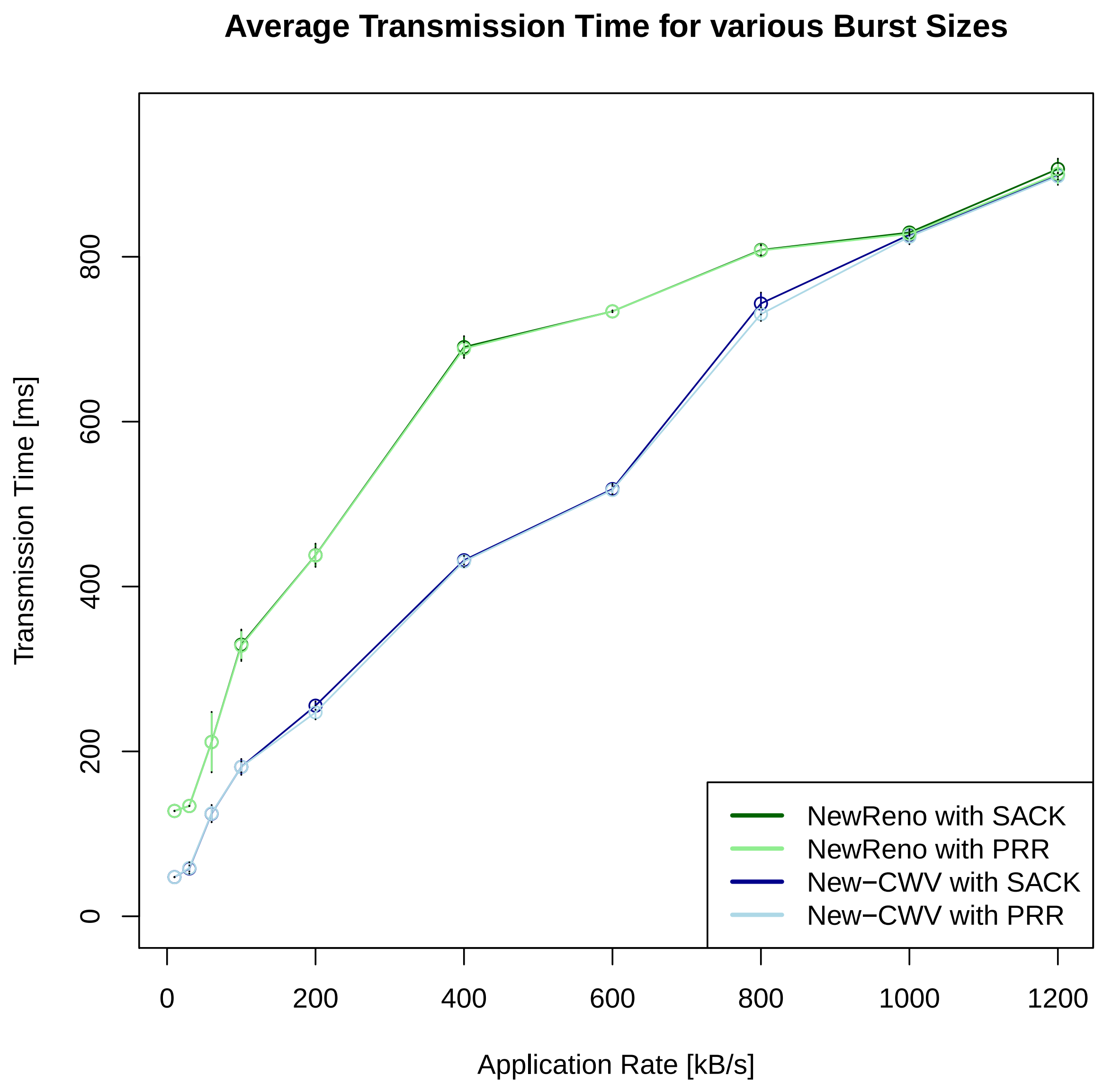

Figure 5.3: Average burst transmission time for varied Application Rate

Depending on the burst size a gain compared to Standard TCP (NewReno) is observed. This gain lies between 0% and 59.86% for this particular scenario and setup. The 0% is observed in cases where the link is actually saturated typically for high burst sizes. This means that the technique of preserving the congestion window has no effect since the high number of packets that a big burst implies, causes the queue to overflow and thus no benefit is observed. The maximum benefit is observed with a rate of 400kB/s which means that this rate, for this particular setup shows the biggest gain. It is also noticed that the use of PRR improves the performance in the range of 0% and 1.7% for each respective flow. The improvement can mostly be noticed in flows with a higher application rate where a loss is more likely to happen.

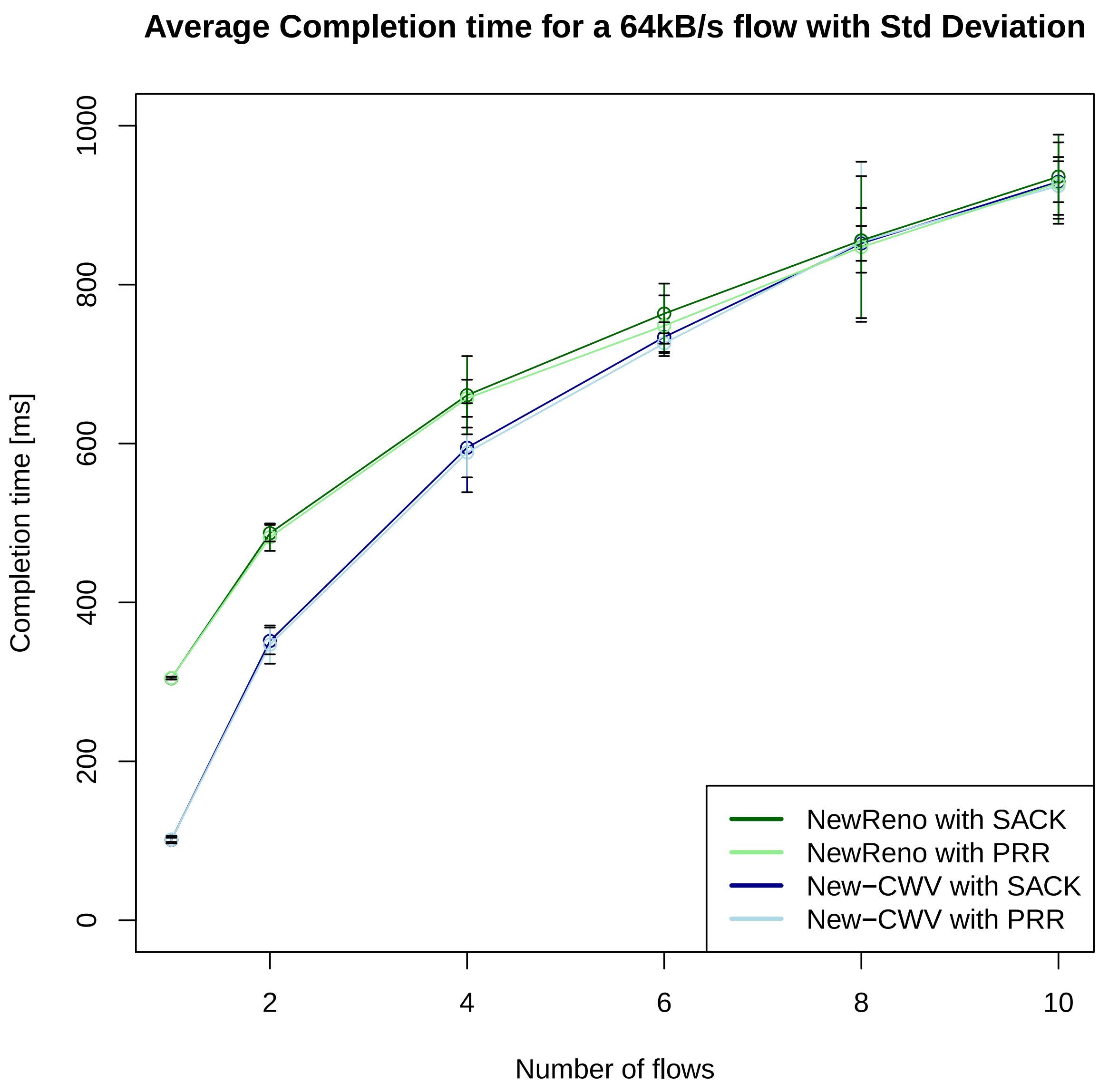

Figure 5.4: Average burst transmission time for a varied flow number

Figure 5.4 depicts the results of the second scenario, which measures the transmission time of each block of 64kB and varies the number of flows that run simultaneously at the same path. There were 30 iterations for each point. The number of flows present simultaneously in the channel, guarantee that there will be congestion. The result is that New-CWV shows a 0.1%-38.7% improvement. It can be seen from Figure 5.4 that having more than 8 flows causes the link to saturate and TCP to show a similar behavior for both New-CWV and NewReno. This happens because of the TCP fairness property, meaning that since there is no available bandwidth for accommodating all flows, an equal share of the path is obtained. It can also be noticed that the use of PRR improves the performance in the range of 0% and 2% for each point.

5.5 Conclusion

Past experience has shown that TCP and TCP-CWV were not suited for rate-limited applications that do not consume the whole of the congestion window. Hence there is a need to update the current standards in order to more efficiently support rate-limited traffic.

In this chapter an experimental method called New-CWV was presented that addresses this issue by proposing a number of changes to the standard congestion control algorithm, so that rate-limited applications are more suitably addressed. New-CWV introduces the concept of a TCP sender being in the Non-validated or Validated phase. When a sender is in the Non-Validated phase, the congestion window freezes, whereas in the Validated phase, standard congestion control applies.

In the measurements section, it is shown that an example rate-limited application (Youtube video streaming) experiences significant performance boost (up to 59.86%) by using New-CWV when compared against Standard TCP congestion control.

Chapter 6

Conclusion

TCP is the most widely used transport layer protocol. It provides a connection-oriented, reliable, byte stream way for transporting data over any kind of network. The term connection-oriented means that a TCP connection has to first be established between the communicating parties. TCP achieves reliability through various mechanisms such as congestion control, loss recovery etc. There have been many proposals that aim into extending the functionality of TCP. The main focus of these extensions is to provide the application level with an ever increasing performance, aiming mostly into decreasing the latency experienced in a connection. [DBEB06] provides a roadmap on the various extensions and standards that have been proposed after [Pos81] where TCP was first introduced.

For the purposes of this thesis, two recently proposed extensions for TCP were implemented in FreeBSD and their performance was evaluated. These extensions aim into increasing the application performance for various traffic types such as bulk and rate-limited traffic. Application performance is measured as the total time needed for a block of data to be transmitted to the receiver by a sender. The comparisons were done between the standard algorithms of FreeBSD and the respective extension.

The first extension that was investigated is the Proportional Rate Reduction algorithm (PRR). Packet losses are the main reason that the completion of data transfer is delayed i.e. the application experiences latency. TCP has two means for recovering from a loss, a scheme called Fast Recovery and a timeout that, when expired indicates that a packet is lost. PRR aims into improving the Fast Recovery scheme of TCP under practical network conditions. It is robust against many issues (e.g. ACK reordering, ACK loss, burst losses, application stalls etc.) present in a real world network, such as the internet. It achieves better performance by taking into account several information that is available in the TCP stack, such as the number of bytes that are SACKed by the receiver and the total number of bytes transmitted during recovery. The measurements, showed that PRR was able to reduce the transmission time of various data blocks between 0% and 9% when compared against the standard SACK based Fast Recovery of FreeBSD.

The second extension that was implemented and evaluated is New-CWV. This method aims to address the issue that standard congestion control algorithms are not suitable for rate-limited applications, that is applications that do not consume the whole of the congestion window that is available to them and may also experience idle periods during the connection. The current behavior is not suited for rate-limited applications since it can potentially lead to congesting the network in the case that the congestion window grows unnecessarily large. This may happen if the application suddenly changes its rate and thus injects more packets into the network by using an inappropriate congestion window. New-CWV proposes that the congestion window freezes during idle and rate-limited periods and describes a method to identify them. The measurements have shown that for a Youtube-like traffic pattern, connections operating with New-CWV experience a significant performance boost (up to 62.4%) when compared against Standard FreeBSD TCP congestion control (NewReno).

6.1 Future Work

Proportional Rate Reduction improves Fast Recovery under practical network conditions. As discussed in Chapter 4, PRR proposes two reduction bounds, as the second part of the algorithm, a conservative one, named PRR-CRB and a more aggressive one named PRR-SSRB. Both reduction bounds inject back into the network the number of bytes that the currently received ACK indicates that were delivered to the receiver, with PRR-SSRB sending one extra packet. PRR-CRB keeps the queues in the intermediate nodes at a stable size meaning that exactly the number of packets that were served at the head of the queue, will be replaced at the end of the queue. If the queue is full then it will stay exactly full, meaning that we can achieve a great utilization of the resources. An interesting future work, would be to study exactly the behavior of PRR-SSRB on queue length and utilization and determine scenarios that PRR-SSRB is more beneficial in real network conditions. An example would be a way to identify that the queue has reached its limit size and thus switching to PRR-CRB from PRR-SSRB should be done.

New-CWV addresses applications that are rate-limited meaning that they don't use the whole of the available congestion window. Consideration has to be taken regarding the chosen value of the congestion window after recovery concludes. As discussed in Chapter 5 the congestion window after a loss recovery event is reset to:

```
cwnd = ((LossFlightSize - R)/2)
```

where R is the number of retransmitted bytes during recovery. This is a more conservative approach than standard TCP that resets the congestion window to the half of the flightsize that was present when the loss occurred. The problem with this approach is that due to excess losses and possible multiple retransmissions of the same data, R might actually grow bigger than *LossFlightSize* resulting in a negative value for the congestion window.

Implementation has to take care of this problem. A possible future work is to measure the difference in performance seen in real networks for the two approaches and decide what is better to use for a possible future RFC.

List of Figures

List of Tables

Bibliography

[AMF+08] Lachlan L. H. Andrew, Cesar Marcondes, Sally Floyd, Lawrence Dunn, Romaric Guillier, Wang Gang, Lars Eggert, Sangtae Ha, and Injong Rhee. Towards a common TCP evaluation suite. In *Protocols for Fast, Long Distance Networks (PFLDnet)*, 5-7 Mar 2008.

[APB09] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681 (Draft Standard), September 2009.

[BAFW03] E. Blanton, M. Allman, K. Fall, and L. Wang. A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP. RFC 3517 (Proposed Standard), April 2003. Obsoleted by RFC 6675.

[BBM+06] Salman A. Baset, Eli Brosh, Vishal Misra, Dan Rubenstein, and Henning Schulzrinne. Understanding the behavior of tcp for real-time cbr workloads. In *Proceedings of the 2006 ACM CoNEXT conference*, CoNEXT '06, pages 57:1–57:2, New York, NY, USA, 2006. ACM.

[BP95] L.S. Brakmo and L.L. Peterson. Tcp vegas: end to end congestion avoidance on a global internet. *Selected Areas in Communications, IEEE Journal on*, 13(8):1465–1480, 1995.

[Bra89] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122 (INTERNET STANDARD), October 1989. Updated by RFCs 1349, 4379, 5884, 6093, 6298, 6633, 6864.

[CDCM13] J. Chu, N. Dukkipati, Y. Cheng, and M. Mathis. Increasing TCP's Initial Window. RFC 6928 (Experimental), April 2013.

[CR09] Marta Carbone and Luigi Rizzo. Dummynet revisited, 2009.

[DBEB06] M. Duke, R. Braden, W. Eddy, and E. Blanton. A Roadmap for Transmission Control Protocol (TCP) Specification Documents. RFC 4614 (Informational), September 2006. Updated by RFC 6247.

[DCCM13] N. Dukkipati, N. Cardwell, Y. Cheng, and M. Mathis. Tail loss probe (tlp): An algorithm for fast recovery of tail losses. Internet Draft, 2 2013.

[DMCG11] Nandita Dukkipati, Matt Mathis, Yuchung Cheng, and Monia Ghobadi. Proportional rate reduction for tcp. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, IMC '11, pages 155–170, New York, NY, USA, 2011. ACM.

[FHG04] S. Floyd, T. Henderson, and A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782 (Proposed Standard), April 2004. Obsoleted by RFC 6582.

[FSS13] G. Fairhurst, A. Sathiaseelan, and R. Secchi. Updating tcp to support rate-limited traffic. Internet Draft, 7 2013.

[GCJM12] Monia Ghobadi, Yuchung Cheng, Ankur Jain, and Matt Mathis. Trickle: rate limiting youtube video streaming. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 17–17, Berkeley, CA, USA, 2012. USENIX Association.

[GMPG00] Tom Goff, James Moronski, D. S. Phatak, and Vipul Gupta. Freeze-tcp: A true end-to-end tcp enhancement mechanism for mobile environments. In *In Proceedings of IEEE INFOCOM'2000, Tel Aviv*, pages 1537–1545, 2000.

[hHK98] Chung hsing Hsu and Ulrich Kremer. Iperf: A framework for automatic construction of performance prediction models. In *IN WORKSHOP ON PROFILE AND FEEDBACK-DIRECTED COMPILATION (PFDC*, 1998.

[Hoe95] Janey C. Hoe. Startup dynamics of tcp's congestion control and avoidance schemes, 1995.

[HPF00] M. Handley, J. Padhye, and S. Floyd. TCP Congestion Window Validation. RFC 2861 (Experimental), June 2000.

[HRX08] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: A new tcp-friendly high-speed tcp variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, 7 2008.

[Jac88] Van Jacobson. Congestion avoidance and control. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM'88)*, pages 314–329. ACM Press, 1988.

[Jon] Rick Jones. Netperf.

[JT07] Wolfgang John and Sven Tafvelin. Analysis of internet backbone traffic and header anomalies observed. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement (IMC'07)*, pages 111–116. ACM Press, 2007.

[MDC13] M. Mathis, N. Dukkipati, and Y. Cheng. Proportional Rate Reduction for TCP. RFC 6937 (Experimental), May 2013.

[MMFR96] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), October 1996.

[MSM99] M. Mathis, J. Semke, and J. Mahdavi. The rate-halving algorithm for tcp congestion control. Internet Draft, 2 1999.

[Ost] Shawn Ostermann. tcptrace.

[PCUKEN09] Marcin Pietrzyk, Jean-Laurent Costeux, Guillaume Urvoy-Keller, and Taoufik En-Najjary. Challenging statistical classification for operational usage: the adsl case. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement (IMC'09)*, pages 122–135. ACM Press, 2009.

[Pos81] J. Postel. Transmission Control Protocol. RFC 793 (INTERNET STANDARD), September 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[RCC$^+$11] Sivasankar Radhakrishnan, Yuchung Cheng, Jerry Chu, Arvind Jain, and Barath Raghavan. Tcp fast open. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, CoNEXT '11, pages 21:1–21:12, New York, NY, USA, 2011. ACM.

[RFB01] 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.

[Sam11] Christian Samsel. Generating diverse internet traffic for the analysis of tcp behavior, 2011.

[She] Tim Shepard. xplot.

[Ste93] W. Richard Stevens. *TCP/IP illustrated (vol. 1): the protocols.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[Tcp] Tcpdump Developers. tcpdump/libpcap.

[ZHK10] A. Zimmermann, A. Hannemann, and T. Kosse. Flowgrind - a new performance measurement tool. In *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, pages 1–6, 2010.