UNIVERSITY OF OSLO
Department of Informatics

Evaluation of TCP retransmission delays

Master thesis

Espen Søgård Paaby (espensp)

24th May 2006

Abstract

Many applications today (e.g., game servers and video streaming servers) deliver time-dependent data to remote users. In TCP based systems, retransmission of data might give high and varying delays. In applications with thin data streams (e.g., interactive applications like games), the interaction between game players raises stringent latency requirements, and it is therefore important to retransmit lost or corrupted data as soon as possible.

In the current version of the Linux kernel (2.6.15), several variations of TCP are included. In this thesis, these variations are compared, tested and evaluated with respect to retransmission latency in different and varying RTT and loss scenarios. The variations are tested for thin and thick data streams, respectively. Thick streams transmit as much data as possible, while the thin streams only need to transfer a small amount of data every once in a while, thus potentially having considerable time intervals between the sending of a few packets.

Due to poor performance experienced for the TCP variations in the thin stream part of the tests, several enhancements are proposed, implemented and tested for use in thin stream scenarios. The loss detection mechanisms of TCP do not perform adequately in a thin stream scenario, resulting in high and varying retransmission delays, something which might violate the stringent latency requirements of interactive games. The implemented enhancements provide considerable improvements in the retransmission delay, reducing both the level and the variation of the retransmission delay values.

Contents

Acknowledgments

First, I would like to thank my supervisors, Carsten Griwodz and Pål Halvorsen, for keeping up with all my bickering, patiently answering all the questions and providing invaluable feedback, as well as providing facilities, machines and useful tools.

Next, I would like to thank my fellow student, Jon Pedersen, for all the long discussions (mostly football, but some technical), we have had during this semester. The technical ones have been quite beneficial for the final result.

Thanks to Rick Jones for providing clarifying feedback on various Netperf issues.

And finally, thanks to my friends and family for their moral support.

Chapter 1

Introduction

1.1 Background

In many applications today, there is a need to deliver time-dependent data to an unknown number of remote users. Video stream servers and game servers are examples of such applications. For the interactive game applications common today, there are typically stringent latency requirements, due to the interaction between players. An example of such game applications are so-called massive multi-player online games (MMOGs), which have become quite comprehensive and have increased considerably in complexity and size. Today they typically involve several thousands of concurrent players, frequently interacting with each other [34]. Typical game situations include role-playing, first person shooter and real-time strategy games. When players are interacting with each other, it is important that they receive the same information more or less at the same time to have a consistent view of the game world. To obtain this, stringent latency requirements are needed, as loss of data should not affect the outcome of the game. E.g., in a first shooter scenario, a player firing his shot before his opponent should not end up being shot himself, because of the loss of a data packet.

Most MMOGs today apply a central server approach for collecting and processing game events generated by players, and point-to-point communication to distribute game state updates. In case of errors, the servers usually use standard protocol implementations available in the operating system kernels. Thus, often using TCP. TCP must live up to the strict latency requirements these applications require, otherwise the game perception experienced by the players will diminish. In this thesis, we are running the Linux 2.6.15 kernel, and modifications are thus made for this kernel.

A considerable number of TCP variations is present in the Linux 2.6.15 kernel. Quite a lot of them are designed for specific network scenarios, and most of these are concerned with achieving a bigger throughput in high-

speed long distance networks. A high-speed network means having a high bandwidth, while transferring over long distances results in a high delay, i.e., these are networks with a high **bandwidth-delay product** (BDP).

Trying to achieve a high throughput means running thick streams. Thick streams have no limit on the sending rate, they try to transmit as much data as possible. TCP is well optimized for thick streams when it comes to achieving a low retransmission delay. However, interactive applications do not necessarily have the need to achieve the highest throughput possible. They are typically thin data stream applications. Thin stream applications only need to send a small amount of data every once in a while. Thus, considerable time intervals between the sending of a few packets with a limited payload are typical.

1.2 Problem definition

As the focus of the TCP protocol has mainly been on thick streams, and thick stream related issues, TCP and it's variations are not optimized for thin streams, and their incorporated mechanisms do not perform satisfactorily in these surroundings. This results in high and varying delay, which is not acceptable in interactive applications.

To fulfill the strict latency requirements of interactive applications, this thesis proposes several enhancements for improving the thin stream performance and reduce the retransmission delay. The enhancements are sender-side modifications only, as they are developed for use in MMOGs. Modifications at the receiver will be difficult to accomplish in such scenario, possibly requiring a change in tens of thousands of receiver machines, which presumably are running several different operating systems (OS) and versions. Thus, this would not only require a change in all of these, but it would also mean that several different implementations had to be developed for use in the different OSes and versions. Additionally, we probably will not even have access to all receiver machines. Assuming a central server approach, applying changes at the sender (server) only require one implementation.

1.3 Outline

The document is organized as follows, in Chapter 2 the different TCP variations present in the Linux 2.6.15 kernel are presented. In Chapter 3, these variations are then tested and evaluated for thick and thin streams, in different RTT and loss scenarios. Based on this evaluation, Chapter 4 presents several enhancements for improving performance in thin stream settings. Chapter 5 then describes the implementation, testing and evaluation of the proposed enhancements, before we summarize our findings in Chapter 6.

Chapter 2

The TCP protocol variations

Transmission Control Protocol (TCP) is one of several transport protocols. It is reliable, stream-oriented and flow-controlled. Additionally, TCP is connection-oriented, meaning it establishes a connection when data needs to be transferred. The data is sent in packets and in an ordered manner at the transport layer.

TCP is considerate to other traffic, limiting the sending rate based on perceived network congestion. This is **congestion control**. If there is little traffic, the individual TCP connections increase the sending rate, and if the opposite is true, the individual connections reduce their rate. TCP follows a scheme called **additive-increase, multiplicative-decrease** (AIMD) in adjusting the rate. The sending rate is governed by the **congestion window** (cwnd), important for congestion control.

The cwnd is a sender-side limit on the amount of data the sender can transmit into the network before receiving an **acknowledgment** (ACK), while the **receiver's advertised window** (rwnd) is a receiver-side limit on the amount of outstanding data. The minimum of cwnd and rwnd governs data transmission. This is **flow-control**. [11]

Cwnd and rwnd changes throughout the connection. Exactly how it is changed depends on the TCP protocol variation used. This is explained under the individual TCP variations in the following section. The receiver informs the sender of packet arrival by sending acknowledgments for the packets it receives. A big part of today's traffic is TCP-based. Examples are e-mail, file transfers and web-traffic.

TCP is said to be fair, that is, streams that share a path get an equal share of the bandwidth [33]. This is not entirely true, as long distance traffic is disadvantaged by it's higher **round-trip times** (RTT), resulting in higher loss probability per RTT and slower recovery. Non-TCP protocols are said to be **TCP-friendly** if their long-term throughput does not exceed the throughput of a conformant TCP connection under the same conditions [31].

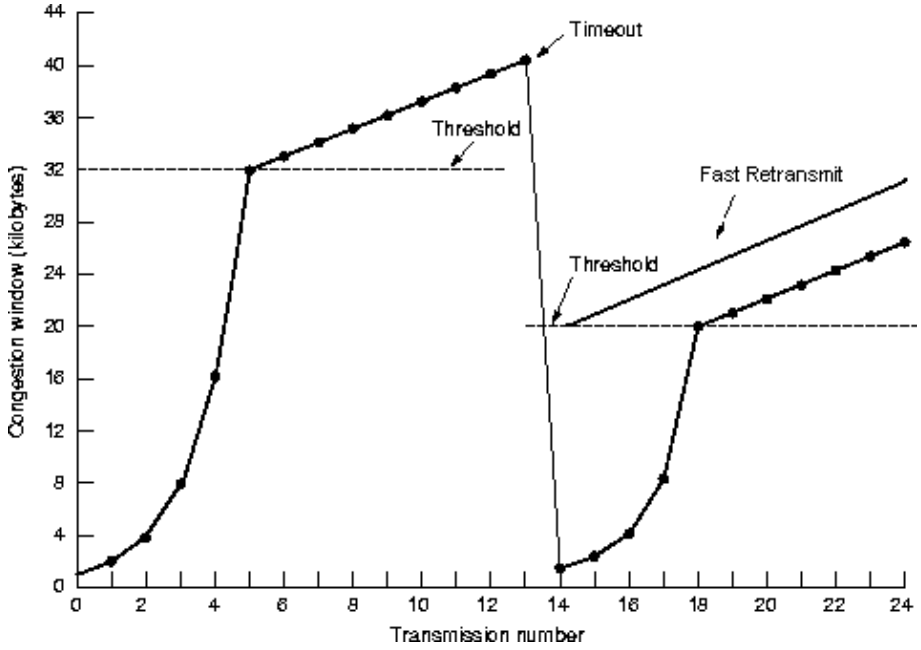

Figure 2.1: Cwnd dynamics

2.1 TCP Reno

TCP Reno [24] [11] [26] is the default TCP protocol in many systems today.

Reno uses a **Slow-start** algorithm when it has opened a connection, or when a retransmission timeout has occurred (these terms will be explained shortly). During this phase it probes the network in order to find the available bandwidth. **Sender Maximum Segment Size** (SMSS) is the maximum segment size the sender can transmit in one packet. The initial value of cwnd must be less than or equal to $2 * SMSS$, and never be more than 2 segments. During slow-start, cwnd is increased by SMSS (at most) for each received ACK, until it exceeds the **slow-start threshold** (ssthresh), or congestion occurs. Ssthresh is the limit for slow-start. The ssthresh can be set arbitrarily high, but might be reduced later due to congestion. Slow-start is used when $cwnd < ssthresh$, while the **Congestion Avoidance** algorithm is deployed when $cwnd > ssthresh$. If cwnd equals ssthresh, the choice is yours to make. While in congestion avoidance, the cwnd is increased by one SMSS for each RTT. Thus, we have exponential growth during slow-start, but linear growth in congestion avoidance. Congestion avoidance continues until packet loss is discovered by a retransmission timeout. Ssthresh is then set to $cwnd/2$ (see Figure 2.1).

Reno uses 2 different mechanisms to discover and retransmit lost segments. The first one is the **retransmission timeout** (RTO). When this is in-

voked the ssthresh is halved and slow-start is entered. The RTO is based on RTT and variance estimates computed by sampling the time between when a segment is sent and an ACK arrives. But this mechanism contains serious weaknesses, due to it's dependency to a clock too coarse for this type of service. In BSD-based implementations, the clock used only "ticks" once every 500 ms, and timeouts can only be checked for each "tick". In Linux, the minimum RTO value is set to 200 ms ($Hz/5$), but increases with the RTT, thus maintaining a high RTO value. This is insufficient, so in addition, Reno incorporates the Fast Retransmit and Fast Recovery mechanisms.

Reno sends a **duplicate ACK** (dupACK) when it receives a segment it cannot confirm, because it has not received all of the previous segments yet. Reno does not only retransmit when a timeout occurs, but also when it receives n dupACKs (**Fast Retransmit**), where 3 is the usual value of n. Thus, n is the **Fast Retransmit threshold** (frexmt_thresh). When the third dupACK arrives, the first unacknowledged segment is retransmitted, and **Fast Recovery** is entered. First, the ssthresh is set to $cwnd/2$. The cwnd is then set to $ssthresh + 3 * SMSS$. The reason for not jumping to slow-start is that the dupACKs do not only imply that a segment has been lost, but also that segments are leaving the network. The receiver can only send a dupACK when a segment arrives. This segment has then left the network and is contained in the receiver's buffer.

Each time the sender receives a dupACK, the cwnd is increased by one SMSS, since the arrival of a dupACK suggests that a segment has left the network. A new segment can then be transmitted, if permitted by the new cwnd. When the sender receives a non-duplicate ACK, it sets the cwnd to ssthresh. This ACK should acknowledge the retransmitted segment, and all the intermediate segments between the lost segment and the receipt of the third dupACK.

2.1.1 TCP New Reno

New Reno [16] [24] modifies the original's Fast Retransmit and Fast Recovery in order to deal with multiple packet loss. Reno leaves Fast Recovery when it receives the first non-duplicate ACK. But if multiple packet loss has occurred, this ACK will be followed by additional dupACKs, and Reno will enter another cycle of Fast Retransmit and Fast Recovery, resulting in another decrease of cwnd and ssthresh. New Reno remains in Fast Recovery until every lost packet has been retransmitted and acknowledged, thus preventing a further decrease in throughput.

New Reno also uses the concept of **partial acknowledgments**. If multiple packet losses occur, the ACK for the retransmitted packet will acknowledge some, but not all of the segments transmitted before Fast Retransmit. Thus, a partial ACK indicates that another packet was lost, and New Reno retransmits the first unacknowledged packet.

Because of this, New Reno saves the highest sequence number (seqnr) sent so far. It then proceeds with retransmission and Fast Recovery as usual. When a non-duplicate ACK arrives, it checks if the ACK covers the highest seqnr. If this is not the case, the first unacknowledged packet is retransmitted, and the retransmission timer is reset, but it does not leave Fast Recovery. Otherwise, if the ACK covers the seqnr, it leaves Fast Recovery in the regular way, i.e., the cwnd is set to ssthresh.

Additionally, New Reno is able to discover "false" dupACKs, sent because of multiple packet loss. Due to multiple loss, the ACK for the retransmitted packet is followed by more dupACKs, indicating the loss of another packet. This will bring Reno into another Fast Retransmit scenario, causing further reduction of throughput. New Reno checks if the dupACKs cover the highest seqnr (mentioned earlier). If so, Fast Retransmit is invoked again (this is a new incident). Otherwise, these are acknowledgments sent before the timeout, and the mechanisms mentioned in the previous paragraph will ensure their retransmission.

Still, New Reno only retransmits one packet per RTT. It cannot predict the next lost packet until it receives the ACK for the previous one. Thus, it could take a substantial amount of time to recover from a loss episode, depending on the number of packets lost and the size of the RTT.

It is the New Reno version of the TCP Reno protocol that is implemented in the 2.6.15 kernel. Thus, the old Reno will not be tested here. Despite this, the name of the congestion control parameter for setting New Reno congestion control is still **reno**. The TCP variations presented in the following sections provide alternate congestion control strategies, and can be enabled to replace New Reno congestion control.

2.2 TCP Vegas

TCP Vegas [26] [29] [40] [36] is a modification of Reno, and provides alternative implementation of 3 key mechanisms. First of all, Vegas introduces a new retransmission strategy. Even though Reno's Fast Retransmit and Fast Recovery are very successful, and often prevent retransmission timeouts, it still partially depends on this technique. Vegas eliminates this dependency by dynamically computing the timeout value. Vegas reads and records the system clock each time a segment is sent. When an ACK arrives, the time is read again, and *Rtt* is calculated using these values. This estimate is used for deciding when to retransmit for the 2 following scenarios:

- The first one is the arrival of a dupACK. Vegas checks if the difference between the recorded sending time and the present is higher than the previously calculated timeout value (*Rtt*). If so, the segment is retransmitted without reducing cwnd.

- The second is the arrival of the first or second non-duplicate ACK following a retransmission. Again Vegas checks if the time interval exceeds the timeout value. If so, the segment is retransmitted. This catches any segments lost previous to the retransmission.

This dynamic calculation of the timeout value provides a quicker reaction to packet loss. Additionally, the estimate is used to discover other lost packets, preventing a further reduction of throughput.

Next, Vegas provides a different congestion avoidance algorithm. Reno uses the loss of segments as an indication of congestion in the network, i.e. Reno needs to create packet loss in order to find the available bandwidth. Thus, Reno contributes to the congestion, creating it's own losses. This might not be expensive if this is caught by Fast Retransmit and Fast Recovery, but unnecessarily filling the routers buffers might cause packet loss for the other connections using these routers.

Vegas' alternate congestion avoidance algorithm is able to discover congestion tendencies, and then perform adequate adjustments to the sending rate, without causing packet loss. In addition, it provides a better utilization of the bandwidth, as it responds quickly if temporary increases in the available bandwidth should emerge.

First the connection's *BaseRTT* is defined as the RTT when congestion is absent in the network, meaning the lowest measurement of the RTT. This is usually the RTT for the first segment sent in the connection. Assuming the connection is not overflowed, the expected throughput is $expected = WindowSize/BaseRTT$. *WindowSize* is the current size of the cwnd. Vegas then computes the actual sending rate, *actual*. This is done by recording the transmission time for a given segment and measuring the number of bytes transmitted before the ACK arrives. Then, the RTT for the segment is calculated, and the number of bytes transmitted is divided by this RTT, which gives the rate (*actual*). *Expected* and *actual* are then compared, and appropriate adjustments to the window are performed. This is done by calculating $Diff = expected - actual$. *Diff* is positive or zero by definition, since a negative value would suggest that we change *BaseRTT* to the last sampled RTT.

Next, 2 thresholds α and β are defined, $\alpha < \beta$. α and β specify having too little or too much extra data in the network, respectively. α and β are defined in Linux 2.6.15 as 2 and 8 segments, respectively. Extra data is specified as data that would not have been sent if the bandwidth used by the connection exactly matched the available bandwidth [26]. This leads to buffering of data in the routers, which will obviously cause congestion if too much extra data is sent. However, if not enough extra data is sent, the connection will not be able to respond rapidly enough in case of transient increases in the available bandwidth. Vegas' goal is to maintain the "right" amount of extra data in the network. If $Diff < \alpha$, the congestion window

is linearly increased during the next RTT, while $Diff > \beta$ causes a linearly decrease. If $\alpha < Diff < \beta$, the window remains the same.

Vegas' last modification concerns the slow-start algorithm. When slow-start is used at the start of a connection, one has no idea what the available bandwidth is. However, if slow-start is invoked later in the connection, it is known that this was due to packet loss with window size *cwnd*, and ssthresh is set to half. But, at the start, it is difficult to predict a proper value for ssthresh. If it is set to low, the exponential increase will stall too early, and it will take a substantial amount of time to reach the available bandwidth. On the other hand, if ssthresh is set too high, the increase will overshoot the available bandwidth, potentially causing congestion. In order to discover and prevent this during slow-start, Vegas only allows an increase of the window every other RTT. In between, the window stays fixed, so valid comparisons between the expected and actual throughput rate can be performed. When the difference between *expected* and *actual* rate (i.e., *Diff*) corresponds to one router buffer, Vegas changes from slow-start to linear increase/decrease of the window (congestion avoidance). However, despite these precautions, Vegas might still overshoot.

Only allowing an increase of the cwnd every other RTT means that Vegas needs more time to reach the ssthresh, thus staying longer in slow-start. This has a massive effect for the throughput, and is extremely considerate, as connections running other TCP variations will not follow the same approach.

According to [36], Vegas got considerable problems coexisting with Reno, which will make it considerably more difficult for Vegas to expand, considering that Reno is the dominant protocol used in the Internet today. Vegas' performance decreases below Reno's if it gets competition from the latter. Vegas' modified slow-start and congestion avoidance algorithms are too considerate towards Reno, politely backing off in case of congestion, something Reno greedily will exploit.

Unfortunately, the Linux 2.6.15 kernel only implements the congestion avoidance algorithm of Vegas, not the Fast Retransmit and Recovery or slow-start modifications (slow-start is fortunate though). Since this is one of the parts that causes Vegas trouble, this might weaken Vegas' impression, as the other (possibly excellent) modifications will not be tested.

2.3 TCP BIC

TCP BIC [1] [43] is a TCP variation designed for high-speed long distance networks, i.e., high BDP. BIC is short for **Binary Increase Congestion**. The demand for fast download of large-sized data is increasing, and hence, networks that can handle this demand are expanding. However, regular TCP cannot satisfy these demands. TCP does not respond quickly enough

in high-speed long distance networks, leaving sizeable amounts of bandwidth unused.

TCP BIC is a congestion control protocol designed to solve this problem. What makes BIC special is it's unique window growth. When packet loss occurs, the window is reduced by a multiplicative factor. The window size just before the loss is set to *maximum*, while the window size immediately after the reduction is set to *minimum*. Then, BIC performs a binary search using these 2 parameters. It jumps to the midpoint between these values. Since packet loss occurred at *maximum*, the window size currently manageable by the connection must be somewhere in between these 2 values. However, jumping to the midpoint could be too much of an increase within one RTT. Thus, if the distance between the midpoint and the current *minimum* is greater than a fixed constant, S_{max}, the window size is instead increased by S_{max} ([43] uses $S_{max} = 32$). If the new window size does not cause packet loss, the window size becomes the new *minimum*. However, if packet loss occurs, the window size becomes the new *maximum*. This process continues until the increase is less than a small constant, S_{min} ([43] uses $S_{min} = 0.01$). The window is then set to the current *maximum*. Thus, the growth function following packet loss will most likely be a linear one followed by a logarithmic one.

Since the growth rate decreases as BIC gets closer to the equilibrium, the potential overshoot is typically less than for other protocols, reducing BIC's influence on other connections. BIC also provides a better utilization of the bandwidth due to it's fast convergence towards the available bandwidth.

If the window grows past the *maximum*, the equilibrium window size must be bigger than the current *maximum*, and a new *maximum* must be found. BIC then enters a phase called "max probing". At first, the window grows slowly assuming that the new *maximum* is nearby. If no packet loss is experienced during this period, it assumes that the equilibrium maximum is further away, and changes to linear growth by increasing the window by a large, fixed constant. Contrary to the binary search phase, BIC's growth function is in this phase exponential at first (which is very slow in the beginning), followed by linear growth.

BIC's growth function might still prove aggressive to other TCP variations, especially in low-speed networks or networks with low RTT. Thus, a BIC variant called **CUBIC** has been developed. CUBIC enhances the TCP-friendliness and provides a modified growth function, which grows much more slowly than binary increase (which is logarithmic) near the origin. BIC can also exhibit extremely slow convergence following network disturbances such as the start-up of new flows [3]. Despite this, BIC is the default TCP congestion control variation in the Linux 2.6.15 kernel.

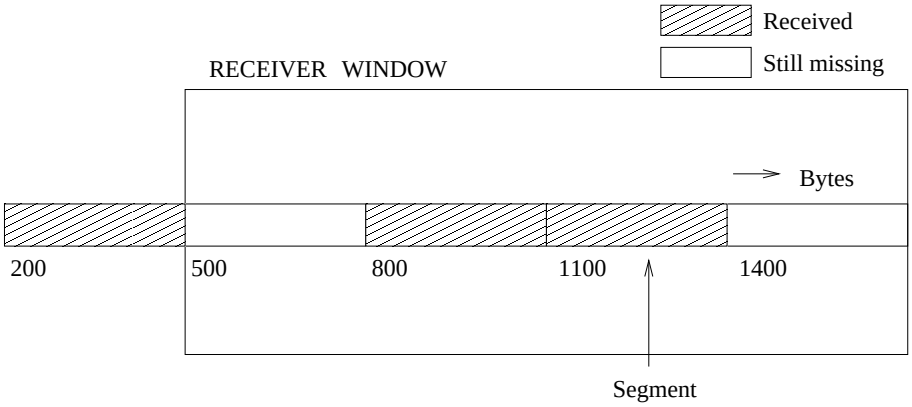

Figure 2.2: Dynamics of the receiver window

2.4 TCP SACK

SACK [8] [10] is short for **Selective Acknowledgments** and is a strategy provided to satisfactory handle multiple packet loss. Multiple packet loss from a window can have a significant influence on TCP throughput. The problem is that TCP does not acknowledge packets that are not located at the left edge of the window (see Figure 2.2). If the packet received is not the expected one, it is not acknowledged even though it indeed successfully arrived. This forces the sender to wait an entire RTT to discover each packet loss, or to unnecessarily retransmit packets which may have already been successfully received. Thus, multiple packet loss generally causes a significant reduction of throughput.

By using SACK the receiver informs the transmitter which packets have arrived successfully, enabling the sender to only retransmit packets that were actually lost. SACK provides this information by putting each received block's first and last 32 bits in the header. SACK can be enabled by using the **SACK-permitted option** in a SYN message.

2.4.1 TCP DSACK

DSACK [12] is an extension of TCP SACK and is short for **Duplicate SACK**. DSACK is able to acknowledge duplicate packets. When duplicate packets are received, the first block of the SACK-field is used to denote the seqnrs of the packet that triggered the ACK. This enables the sender to infer the order of packets received by the receiver, allowing it to infer when a packet was unnecessarily retransmitted.

This information might prove to be useful for the sender, securing a more robust execution of it's tasks in environments exposed to for example

packet replication, packet reordering or loss of ACKs.

2.4.2 TCP FACK

FACK [39] [2] is short for **Forward Acknowledgment** and is based on TCP Reno with SACK. FACK is using the information provided by SACK to compute a better estimate of the amount of data currently in transit (outstanding data). This information is essential for any congestion control algorithm.

To estimate the amount of outstanding data, FACK introduces a new variable, *fack*, denoting the highest seqnr known to have been received plus 1. The variables *next* and *una* represent the first byte of data yet to be sent and the first unacknowledged byte, respectively. The window starts from *una* and ends with *next*. This means that some blocks from *una* to *fack* have been acknowledged, but not all of them. Blocks that have not been acknowledged by SACK are still outstanding, i.e., they are retransmitted but not acknowledged. Thus, the amount of data currently in transit is $next - fack + retransmitted$. However, FACK might in some cases underestimate the amount of outstanding data.

Additionally, FACK addresses another unfortunate side effect experienced by Reno. When halving the window, there is a pause in the sender's transmission until enough data has left the network. This pause is reflected in the ACKs, and leads to the transmission of a group of segments at the start of each RTT. This uneven distribution can be avoided by a gradual reduction of the window, something FACK does. This allows congestion to go away and thereby reduces the probability of a double loss.

2.5 TCP Westwood

TCP Westwood [32] [42] [18] is a modification of New Reno developed to provide increased performance in surroundings subject to high BDP, packet loss and/or dynamic bandwidth. High-speed heterogeneous or wireless networks are examples. In these surroundings, packet loss might occur because of noise or other types of external disturbances. Westwood is able to separate this form of packet loss from the usual one (packet loss due to congestion), ensuring appropriate adaptions for both types of loss. Westwood additionally includes a mechanism to utilize dynamic bandwidth, called **Agile Probing**.

ERE is short for **Eligible Rate Estimation** and is a kind of bandwidth estimate. Westwood's key idea is the use of ERE methods, employed to set the cwnd and ssthresh intelligently following packet loss. Westwood calculates the ERE estimate over a time *Tk*, and is computed as the amount of acknowledged data during *Tk*. *Tk* depends on the congestion level. The

latter is measured as the difference between the *expected* and *actual* rate (as in TCP Vegas, Section 2.2). *Tk* is computed as:

$$Tk = RTT * ((expected - actual)/expected)$$

where *RTT* is an estimate of the last RTT as in Vegas. When there is no congestion, the difference between *expected* and *actual* (i.e. *Diff*) will be small, producing a small *Tk*. Westwood then uses the last collected ERE when computing the new cwnd and ssthresh following packet loss. Ssthresh is set to $(ERE * BaseRTT)/segmentsize$, while cwnd depends on the scenario. If 3 dupACKs are received and cwnd is larger than the new ssthresh, cwnd is set to ssthresh. Otherwise, nothing is done. If the scenario was caused by a timeout, the cwnd is set to 1. However, if the new ssthresh is less than 2, provided the scenario was a timeout, ssthresh is set to 2.

Reno is not able to tell one situation from the other, it blindly assumes congestion following 3 duplicate ACKs. This leads to a drastic decrease in throughput, which might not have been required.

According to [22], Westwood might overestimate the available bandwidth, something which will lead to an aggressive and unfair behaviour towards new TCP-sessions.

2.6 High-speed TCP

High-speed TCP (HSTCP) [30] [23] [17] is a TCP modification developed to deal with high-speed networks and the challenges they offer. Regular TCP is not able to utilize the available bandwidth in a high-speed environment, as mentioned under TCP BIC (Section 2.3). TCP's problem is that it's AIMD mechanism reacts too conservatively to packet loss, thus preventing reaching a large enough congestion window to utilize the available bandwidth. The limitation is the growth rate of the window, which is not rapid enough in this environment. Few of the TCP variations are concerned with this aspect. Thus, even though they are useful, the SACK and ECN [14] options are limited by their functionality in high-speed surroundings.

In a steady-state environment with a loss rate p, the average cwnd for TCP is roughly $1.2/sqrt(p)$ (*sqrt* - square root). This is called **TCP's response function**, and places a serious constraint on the achievable window size in realistic surroundings. As an example, to achieve an average throughput rate of 10 Gbps, with a packet size of 1500 B and RTT of 100 ms, an average cwnd of 83.333 would be required, and thus, a packet loss rate of 1/5.000.000.000 [30]. This is **not** realistic in current networks.

HSTCP suggests a modification to the response function in order to remove this constraint. It introduces 3 parameters: *Low_Window*, *High_Window* and *High_P*. To provide TCP compatibility, HSTCP uses the same response function as TCP when the current cwnd is less or equal to *Low_Window*.

Thus, if the loss rate is high (implies congestion), the cwnd will consequently be relatively small. Introducing a new, more aggressive response function will only add to the congestion in this case.

When the current cwnd is greater than *Low_Window*, HSTCP switches to it's own response function. [30] sets *Low_Window* to 38, corresponding to a loss rate of 10^{-3} for TCP. *High_Window* is set to 83.000, which is the average cwnd size required to achieve 10 Gbps as mentioned earlier. *High_P* is set to 10^{-7}, specifying that for HSTCP, a packet loss rate of 10^{-7} allows the average window size to reach 83.000. HSTCP's response function, for average window size W greater than *Low_Window*, is then defined as:

$$W = (p/Low\_P)^S Low\_Window$$

S is a constant defined as:

$$S = (log(High\_Window) - log(Low\_Window))/(log(High\_P) - log(Low\_P))$$

Low_P is the packet loss rate corresponding to *Low_Window*, thus 10^{-3} as mentioned earlier. Using these values, we get the response function

$$W = 0.12/p^{0.835}$$

which enables much larger values of W. Thus, HSTCP is less sensitive to packet loss and should be able to maintain a high-speed connection.

HSTCP might impose a certain degree of unfairness as it does not reduce it's transfer rate as much as regular TCP. Additionally, during congestion control, it's slow-start might be more aggressive. This is especially the case when the loss rate is relatively high, as HSTCP will have a bigger cwnd and thus be able to send more data into the network. HSTCP can also exhibit extremely slow convergence following network disturbances such as the start-up of new flows [3].

2.7 Scalable TCP

Scalable TCP (STCP) [35] is another attempt to provide better utilization in high-speed wide area networks. Regular TCP's poor performance in this environment has already been thoroughly introduced, and STCP aims to better utilize the bandwidth in high BDP networks. In order to achieve this, STCP suggests alternative algorithms for cwnd increase and decrease (in congestion avoidance).

STCP proposes to increment the cwnd by $cwnd = cwnd + 0.01$ for each ACK received in an RTT. Regular TCP increments the window as $cwnd = cwnd + (1/cwnd)$, which gives $cwnd + 1$ for each RTT, as we may send cwnd number of packets per RTT. At the first detection of congestion in an RTT, the window is decreased as $cwnd = cwnd - (0.125 * cwnd)$.

Thus, following a loss event, it will take STCP about 70 RTTs to recover from the loss at any rate. TCP needs $cwnd/2$ RTTs, which could be quite some time if the window and/or RTT are large. Thus, the time it takes to recover may exceed the interval between congestion events, leading to a gradually decrease of throughput.

These modifications result in a different response function for STCP, i.e., $(0.01/0.125) * (1/p)$. However, this gives STCP a certain advantage over regular TCP in the recovery process following a loss event, which could lead to an unfair share of the bandwidth. To prevent this, STCP uses the same algorithm as regular TCP until a **legacy window** (lwnd) size is reached. As previously explained in HSTCP, regular TCP has some practical limitations of the cwnd size, meaning the window tends to not be greater than a certain size, lwnd. When cwnd exceeds lwnd, STCP switches to the it's own algorithm.

Providing this flexibility makes STCP fair towards other regular TCP connections until lwnd is reached, at which point the other connections should have serious concerns about their packet loss recovery times anyway. STCP is simpler to implement than HSTCP, due to it's use of constants in the window update algorithm, opposed to the parameterized HSTCP. However, it is known that it may fail to converge to fairness in drop-tail networks [3].

2.8 H-TCP

H-TCP [3] [41] [37] is yet another approach to improve TCP's performance when the cwnd is large, implying networks with high BDP. H-TCP's algorithm resembles those of HSTCP, BIC and STCP, but aims to remove certain weaknesses present in these protocols (explained under the spesific protocol).

H-TCP defines 2 values, Δ and Δ_L. Δ is the time elapsed since the last congestion incident, measured in seconds. Δ_L is the threshold for switching from the legacy to the new increase function. H-TCP proposes to change the additive increase element of the AIMD mechanism to

$$cwnd = cwnd + \alpha/cwnd$$

for each ACK received. α is first calculated as

$$1 + 10(\Delta - \Delta_L) + 0.5 * (\Delta - \Delta_L)^2$$

provided that $\Delta > \Delta_L$. If $\Delta <= \Delta_L$, α is just 1. This is done to provide backward compatibility with regular TCP. If $\Delta > \Delta_L$, α is set as

$$\alpha = 2(1 - \beta)\alpha\Delta$$

β is calculated as $\beta = RTTmin/RTTmax$, but must stay between 0.5 and 0.8 ($\beta \in [0.5, 0.8]$). When a congestion event occurs the cwnd is changed accordingly, $cwnd = \beta * cwnd$.

Using Δ maintains a certain degree of symmetry between competing flows, since it is based on the time elapsed since the last congestion event. If packet drops are synchronized, Δ will be the same for all flows. Otherwise, Δ is still the same on average, provided that each flow shares the same probability of backing off in case of congestion. Hence, symmetry is still maintained in an average sense.

2.9 TCP Hybla

TCP Hybla [28] [27] is a recently developed modification designed to remove the penalties from which connections with long RTTs suffer. The well-known slow-start mechanism of TCP maintains an exponential increase of the cwnd per RTT, while the congestion avoidance mechanism causes a linear increase per RTT. Thus, the increment rate depends on the RTT, meaning connections with longer RTTs require more time to increase their cwnds compared to connections with short RTTs, resulting in corresponding throughput penalty.

The cwnd size in segments at time t, *W(t)*, depends on the RTT accordingly:

$$\text{W(t)} = \begin{cases} 2^{(t/RTT)} & \text{for } 0 \leq t < t_\gamma \text{ (slow-start)} \\ \frac{t-t_\gamma}{RTT} + \gamma & \text{for } t \geq t_\gamma \text{ (congestion avoidance)} \end{cases}$$

t_γ denotes the time the ssthresh value γ is reached.

The same problem remains if we look at loss recovery. Taking New Reno as an example, it only recovers 1 packet per RTT. Thus, the time New Reno stays in the recovery phase depends on the RTT.

The basic idea of TCP Hybla is to equalize the performance of connections with different RTTs. Thus, the connections must have the same **instantaneous transmission rate** (i.e., the amount of segments transmitted per second), $B(t) = W(t)/RTT$. This can be achieved in two steps, first by making *W(t)* independent of the RTT, and second by compensating the effect of the division by RTT. [27]

Hybla introduces a normalized RTT, ρ, defined as $\rho = RTT/RTT_0$. RTT_0 is the RTT of a reference connection to which we want to equalize our performance. The first step is then performed by multiplying the time t by ρ, if in slow-start. If in congestion avoidance, ρ is multiplied by the time that has elapsed since the reaching of the ssthresh. This results in a *W(t)* independent of the RTT. The second step is then executed by multiplying this cwnd by ρ. This gives the following modified algorithm:

$$\text{W(t)} = \begin{cases} \rho 2^{(\rho t/RTT)} & \text{for } 0 \leq t < t_{\gamma,0} \text{ (slow-start)} \\ \rho\left(\rho\frac{t-t_{\gamma,0}}{RTT} + \gamma\right) & \text{for } t \geq t_{\gamma,0} \text{ (congestion avoidance)} \end{cases}$$

As a consequence of the modification introduced in the second step, the switching time $t_{\gamma,0}$ (the time when the cwnd reaches $\rho\gamma$), is the same for all RTTs. Thus, $t_{\gamma,0} = RTT_0\ log_2\ \gamma$. In [28], RTT_0 is set to 25 ms, the Linux 2.6.15 implementation has chosen the same value. Hybla acts exactly like New Reno when the minimum RTT estimated by the sender is less or equal to RTT_0.

Hybla's modified growth algorithm is thus:

$$\text{cwnd} = \begin{cases} cwnd + 2^{\rho} - 1, & \text{(slow-start)} \\ cwnd + \rho^2/cwnd, & \text{(congestion avoidance)} \end{cases}$$

Due to the difference in the congestion control algorithm, Hybla will have a larger average cwnd for connections with high RTTs than standard TCP. Thus, multiple losses within the same window will happen more frequently in Hybla's case, and as the time the protocol stays in the recovery phase depends on the RTT, appropriate measures to deal with this are recommended to be in place, i.e. SACK is recommended in the TCP Hybla proposal.

2.10 Summary

In this chapter, the different TCP variations in the Linux 2.6.15 kernel have been presented. The considerable number of variations suggests that the Reno variation has some shortcomings, as most of the variations aim to improve some limitation in the Reno variation (often related to a specific environment).

Most of the modifications are designed for high-speed environments, indicating limitations of the Reno variation in this kind of surrounding, and additionally where the main focus of TCP research is. The most important characteristics of the different variations will now be shortly summarized.

2.10.1 Characteristics of the TCP variations

TCP Reno

- The default TCP protocol in many systems today
- Cwnd increase in congestion avoidance: $cwnd = cwnd + (1/cwnd)$ (per ACK received in an RTT)
  giving $cwnd = cwnd + 1$ (per RTT)

- Upon packet loss: $ssthresh = cwnd/2$

TCP New Reno

- Stays in Fast Recovery (opposed to original Reno)

TCP Vegas

- Dynamically calculates an additional timeout value, allowing it to potentially retransmit prior to the reception of the third dupACK, if the time exceeded this timeout value
- Adjusts the cwnd in congestion avoidance based on perceived network congestion
- Only allows an increase of the cwnd every other RTT, while in slow-start. This has a massive effect on throughput, and is presumably one of the main reasons for the problems Vegas experience in many systems

TCP BIC

- Reduces the cwnd with a multiplicative factor (*minimum*), in case of packet loss. Performs a binary search between (*minimum*) and the cwnd at the time of loss (*maximum*)
- Additionally performs 'max probing', if the cwnd grows past the current *maximum*
- Designed for high-speed environments

TCP SACK

- Informs the sender which packets were successfully received, enabling the sender to discover multiple packet loss, and to only retransmit packets that were actually lost

TCP DSACK

- Extension of SACK, additionally informs the sender about duplicate packets received as well

TCP FACK

- Uses the information provided by SACK to compute a better estimate of outstanding data
- Reduces the cwnd gradually

TCP Westwood

- Able to separate between different types of packet loss
- Uses a kind of bandwidth estimate, *Eligible Rate Estimation* (ERE), in calculating the new ssthresh and cwnd following packet loss.
  $ssthresh = (ERE * BaseRTT)/segmentsize$
- Designed to handle wireless environments

High-speed TCP

- Modifies the TCP response function to packet loss rate p, $(1.2/sqrt(p))$, in order to reach much bigger cwnd sizes in high-speed environments. HSTCP's modified response function is:
  $0.12/p^{0.835}$
- Designed for high-speed environments

Scalable TCP

- Cwnd increase in congestion avoidance: $cwnd = cwnd + 0.01$ (per ACK received in an RTT)
- Upon the first detection of congestion, the cwmd is decreased accordingly: $cwnd = cwnd - (0.125 * cwnd)$
- Gives a different response function: $(0.01/0.125) * (1/p)$
- Designed for high-speed environments

H-TCP

- Cwnd increase in congestion avoidance: $cwnd = cwnd + \alpha/cwnd$ (per ACK received in an RTT)
  $\alpha = 2(1-\beta)\alpha\Delta$
  Δ is the time elapsed since the last congestion incident, in seconds
- If congestion occur, the cwnd is changed accordingly: $cwnd = \beta * cwnd$ ($\beta = RTTmin/RTTmax$, $\beta \in [0.5, 0.8]$)
- Designed for high-speed environments

TCP Hybla

- Equalize performance for connections with different RTTs
- Cwnd increase:

$$\text{cwnd} = \begin{cases} cwnd + 2^{\rho} - 1, & \text{(slow-start)} \\ cwnd + \rho^2 / cwnd, & \text{(congestion avoidance)} \end{cases}$$

  ρ is a normalized RTT: $\rho = RTT / RTT_0$. RTT_0 is in Linux set to 25 ms
- Designed for connections with long RTTs

Now that the TCP variations have been introduced, we move on to test the different variations individual strategies, and see how they perform under various circumstances.

Chapter 3

Testing of the different TCP variations

The introduction of the considerable number of TCP variations in the Linux 2.6.15 kernel has presented us with a number of different strategies. Now that they have been introduced, it is time to see how they perform in various network environments. As our main focus is on retransmission delay (RTD), we want to test how these different strategies affect the RTD, or determine that they do not influence it. As the variations are designed for a number of different network settings, it is necessary to test the variations in several different RTT and loss scenarios.

Additionally, we want to test the variations for different kinds of streams. More precisely, we want to test the protocols' performance when transmitting thin and thick streams, respectively. For thick streams there are no limits on the sending rate. We just want to transmit as much data as possible, as fast as we can. A lot of the variations are high-speed designed and should boost the stream performance in such surroundings. Testing the variations for thick streams should provide a useful impression of their performance.

However, not all applications have the need to transfer as much data as possible, they mainly want TCP's reliability and ordering. Some just need to transfer a small amount of data every once in a while, i.e., they are thin. However, once data needs to be transmitted, it is beneficial if the transfer is fast. Thus, as these thin streams are limited in the amount of data sent, they have a limited number of packets in flight, and there could be considerable time intervals between the sending of packets. Additionally, the amount of data sent in each packet might be limited as well.

3.1 Test configurations

3.1.1 The testbed

To perform testing of the different TCP variations a network was constructed, consisting of 3 computers, simulating sender, network environment and receiver, respectively (see Figure 3.1). Various network restrictions were set at the middle computer in order to simulate different network types (see own section 3.1.3 below for details).

As we are considering game scenarios, the sender is meant to simulate the game server sending game data to an unknown number of game players. The receiver simulate an arbitrary player. Thus, data is only sent from the sender to the receiver, and additionally, we only change the sender side, as we would have little possibility to alter an arbitrary number of player computers in a real game scenario.

Each computer is running the Linux 2.6.15 kernel, and the TCP variations tested are the ones present in this release. The network cards used are Fast Ethernet cards, resulting in a 100 Mbit/s link between the sender and the receiver.

The individual TCP variations are in turn enabled at the sender. Using **netperf** [7], a stream was sent through the router to the receiver for each variation. **Tcpdump** [19] was used to listen and gather the headers of packets sent by the stream, and **tcptrace** [20] was used to evaluate this information to provide essential RTD statistics for the stream. These programs are described more closely in the following section.

3.1.2 Emulator and simulation settings

The different TCP protocols were enabled using the **sysctl** command, which activates the specified protocol.

Netperf

Netperf is used to send TCP streams. It is a tool for measuring various aspects of networking performance. Netperf is essential for these tests as it provides options to limit the sending rate, useful for simulating thin streams. An example, taken from the tests, is:

```
netperf -H 192.168.2.2
        -w 1000
        -b 4
        -l 1800
        -t TCP_STREAM
        --
        -m 100
```

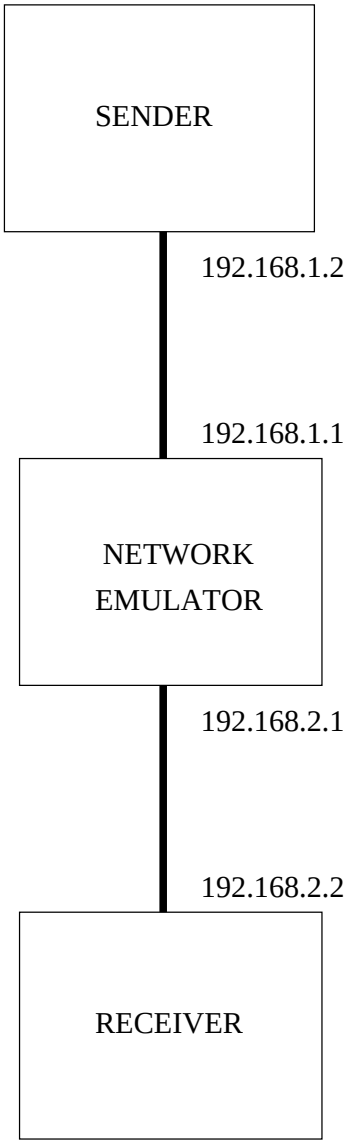

Figure 3.1: The Testbed

The **-H** option specifies the receiver, the **-w** option is used to denote the interval between the bursts in ms, while **-b** specifies the number of messages sent in each burst. **-l** is the length of the test in seconds, while **-t** denotes the type of the stream. **- -** denotes that the remaining options are test specific. Only one option is used, **-m** which is the message size in bytes. Thus, the given command runs a TCP stream test, sending 4 messages of 100 B each, every second (1000 ms) for 30 min (1800 s) to the receiver 192.168.2.2.

Netem

Different constraints on the network are set and modified at the emulator. The **tc** [21] [5] command is used to configure **Traffic Control** in the Linux kernel. Through it, the **netem** [6] module is controlled, to impose delay and packet loss in the network. An example is:

```
tc qdisc add dev eth0 root netem delay 50ms loss 5%
```

The *qdisc* and *add* parameters specifies that we want to add an queuing discipline. *dev eth0* specifies that we want to add it on Ethernet device eth0. *root* specifies that we want to add it to the root hook, while *netem* specifies that we want to use Netem to add delay (*delay*) of 50 ms and a loss rate (*loss*) of 5%.

Tcpdump

Tcpdump is a packet capturing program used to capture the header information of packets sent on a network. An example is:

```
tcpdump -i eth0 -w reno_s.dump &
```

In this example, tcpdump listens on the interface specified in the **-i** option and saves the packet header information to the file given in the **-w** option for later analysis. Here tcpdump listens on *eth0* and saves the packet header information in the *reno_s.dump* dump file.

Tcptrace

Tcptrace is the tool used for analysis of TCP dump files. It can produce several different types of output information. In the tests it was run with the **-lr** options, specifying long output information (**l**) and RTT statistics (**r**), respectively. The information was written to file accordingly:

```
tcptrace -lr reno_s.dump > reno_s.txt
```

```
1  # Connection (192.168.1.2:46803-192.168.2.2:12000)
2  # Retransmission region for packet 21
3  #    18     PKT t 98.3273 seq  410 len  29
4  #    19     ACK t 98.5343 ack  439
5  #    20     PKT t 98.6194 seq  439 len  20
6  #    21 RTR PKT t 99.0793 seq  439 len  20
7  #    22     ACK t 99.2862 ack  459
8  #   Retransmission count: 1
9  #   Time between first trans and last retrans: 459.848 ms
10 #   Time between first trans and final ACK: 666.854 ms
```

Figure 3.2: Loss_details example with RTT 200 ms

Loss_details

This program is provided by my supervisors. It is run against the dump file provided by tcpdump, and offers a detailed view of each loss incident in a given connection. It is useful for understanding the exact execution performed for a given loss incident, which is beneficial when trying to explain certain values.

Figure 3.2 show the output for one specific loss incident. The values in line 3-7 needs further explaining. The first number (*18-22*) is the number of the packet, the next row indicate if this is a retransmission (*RTR*) or a dupACK (*DUP*, not shown). The next says whether the packet is a data packet (*PKT*) or an acknowledgment (*ACK*). Then there is the time of capturing, *t* (*98.3273-99.2862*), in seconds, from the start of the connection. The next row says whether the sequence number following is the first byte in a packet (*seq*) or the last acknowledged byte (*ack*). Then comes the sequence number (*410-459*), and if this is a (re)transmission of a packet (*seq*) we have the length of the packet (*len*) in bytes.

3.1.3 Description and justification of the network types simulated in the tests

Thick streams

Most of the protocols present in the 2.6.15 kernel are designed to improve the performance in high BDP settings (BIC, STCP, HSTCP, H-TCP, Hybla). Thus, to perform a relevant comparison of the protocols, they should be tested in such environments. Hence, no limitations on the sending rate should be imposed, and the loss rate should be low. However, this has an impact on the time needed for each test, as a certain number of retransmission samples is required to secure the tests' validity, allowing any conclusions to be made. A low loss rate imposes an extended run time for each test. In order to avoid an escalation of the time needed for each test run, the loss rate used is not as low as might be requested for some of the protocols. This could partly constrain the performance of some of the high-speed

protocols. However, this provides a satisfactory number of retransmission samples in decent time. 0.5% was chosen as the lowest loss rate.

To test the robustness of the individual protocols, it is necessary to perform the tests with varying delay, simulating different distances in the network, and thus including high-speed long distance networks in the tests. This will expose any impact that varying delay has on the performance of the protocols. 50, 100 and 200 ms delay (each way) were chosen for the tests.

Additionally, it is necessary to perform tests for a different loss scenario, exposing any possible influence this might entail. Thus, a higher loss rate must be introduced in order to observe what the protocols can handle. Westwood is designed for increased performance in wireless networks suffering from high loss rates, thus it should be appropriate to perform tests in such a setting. A loss rate of 5% was chosen, and additionally, delays of 0, 50 and 100 ms (each way).

Thin streams

The most important part of the tests is however for the thin streams, as the goal was to test the performance of these. To simulate thin streams it is necessary to limit the number of packets sent, and to have intervals between the sending of an arbitrary unit of packets. This might be as low as a couple of packets per second, not containing more than a few hundred bytes of data. In the thin stream tests, 4 messages of 100 B each were sent each second in one burst (i.e., 400 Bps). To get a thorough view of the protocols' thin stream performance, a considerable number of network settings was tested. Limiting the number of packets means a limitation of the number of samples per time unit, and thus consequently in the number of retransmission samples. If each message is sent in a separate packet, a maximum of 240 packets can be sent each minute. If the loss rate is 0.5%, a packet loss will occur every two minutes on average, thus requiring quite some time to collect a minimum number of samples of approximately 300-400. To avoid an escalation of the time needed for the tests, the loss rate used is quite high, 5% and 10%, respectively. Even though this is too high for the variations designed for high-speed environments, it is still acceptable. These protocols would not have gained anything on throughput, regardless of the loss rate, as the streams run here are thin. For each loss rate, the protocols were run with delays of 0, 50, 100 and 200 ms (each way).

3.2 Test results and evaluation

It is necessary to get a certain number of samples in order to get representative results, that are not affected by a few extreme values, and to be sure

that the values gotten were representative for this variation configuration. It would always be nice to have more samples, but unfortunately time is an inevitable issue (as it is in most scenarios). This puts some restrictions on the number of samples if these are to be collected within a reasonable time. Being able to collect a sufficient number of samples within an affordable time is most urgent in the thin stream scenarios, as they have a limited transmission of packets, and thus require more time to collect the necessary samples. A total number of samples of approximately 400 at least are chosen to be sufficient in these tests. As mentioned earlier, it would always be beneficial to have more samples, but this is the value that could be reached within the time at our disposal for this thesis. This minimum affects the time length of the run, as the time needed to collect the necessary number of samples varies according to the loss rate. Thus, the length of the run vary from setting to setting.

3.2.1 Evaluation of the TCP variations for thick streams

Since a considerable number of the TCP variations present in the 2.6.15 kernel are developed for use in high BDP networks, it seemed reasonable to perform some tests in such an environment, i.e. without limitations on the stream. The tests were run in 6 different surroundings (network types), varying by RTT and loss rate. Chosen loss rates were 0.5% and 5%. The tests were run with a delay of 50, 100 and 200 ms (each way) for 0.5% and 0, 50 and 100 ms for 5%.

For each setting, each protocol were run, and for each protocol all combinations of SACK, DSACK and FACK were tested in addition to the plain protocol, giving 5 tests per protocol: plain, with SACK, with SACK DSACK, with SACK DSACK FACK and with SACK FACK (SACK must be enabled in order to test the others).

For the settings with 0.5% loss, the tests were run for 30 minutes each, while the settings with 5% loss were run for 10 minutes.

In the tables displaying the test results, the following abbreviations are used:

```
P - plain
S - SACK
D - DSACK
F - FACK
Rexmt - Retransmitted packets
Stddev - Standard Deviation
```

The information presented is extracted from tcptrace's (Section 3.1.2) statistics. The values in the **Retransmission Time Statistics** part are measured as the time between two consecutive (re)transmissions of the same

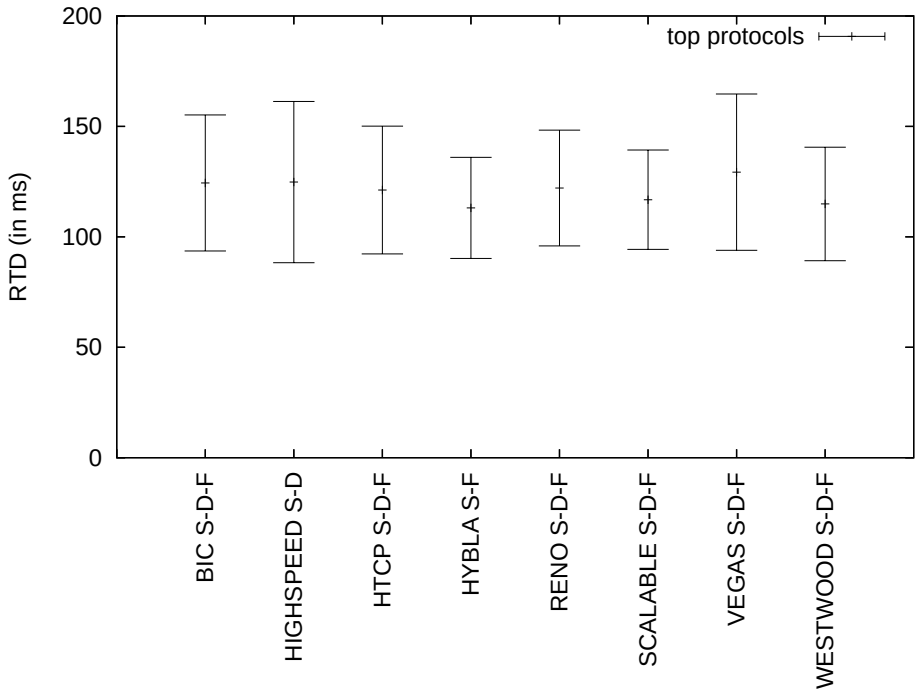

Figure 3.3: The top combination for each variation, thick streams, 50 ms delay, 0.5% loss

segment. That is, the time between the original transmission and the first retransmission, or between 2 retransmissions [20].

Additionally, plots of the RTD with standard deviation, for the best run of each variation, are shown to provide better visualization.

Thick streams, 50 ms delay, 0.5% loss

The delay might be a little too low to correspond to long distance networks, but it is approaching the necessary requirements. More importantly, 0.5% packet loss is quite high, possibly high enough to partly constrain the protocols designed for high-speed.

The results (Table 3.1) for this environment vary little from each other, except for HSTCP which suffers severe problems. Of HSTCP's 5 tests, only the protocol with the SACK DSACK option does not suffer from a surprisingly low throughput (between 5 to 15 times lower than the other protocols). This limitation in throughput is reflected in the average RTD, which is twice as high as for the other protocols. The problem is not that the average is affected by higher max or min values, the average level is much higher in general.

This makes sense, especially when the difference is so great. If the

TCP Variation	Packet Statistics			Retransmission Time Statistics (in ms)			
	Total	Rexmt	Rexmt rate	Min	Max	Avg	Stddev
BIC P	293 096	1 508	0.00514	103.2	598.3	136.0	49.9
BIC S	302 163	1 519	0.00502	102.0	325.8	127.8	31.7
BIC S D	303 609	1 506	0.00496	102.9	632.0	127.7	38.0
BIC S D F	298 192	1 544	0.00517	102.2	330.4	124.4	30.8
BIC S F	299 716	1 578	0.00526	101.0	416.4	124.8	29.7
HIGH-SPEED P	42 877	205	0.00478	104.0	450.5	300.0	79.4
HIGH-SPEED S	43 423	223	0.00513	104.1	548.6	256.4	53.0
HIGH-SPEED S D	255 148	1 302	0.00510	103.9	416.0	132.0	36.5
HIGH-SPEED S D F	42 975	198	0.00460	104.1	655.3	251.4	59.3
HIGH-SPEED S F	43 557	195	0.00447	104.1	654.6	258.1	75.7
H-TCP P	315 903	1 603	0.00507	102.9	588.9	136.2	51.5
H-TCP S	325 265	1 627	0.00500	102.9	404.0	125.9	32.1
H-TCP S D	320 869	1 601	0.00498	101.8	322.3	126.1	33.7
H-TCP S D F	328 861	1 656	0.00503	102.5	326.7	121.2	28.9
H-TCP S F	326 057	1 650	0.00506	102.5	380.0	122.7	31.2
HYBLA P	627 520	3 048	0.00485	102.4	728.8	134.7	66.0
HYBLA S	642 991	3 235	0.00503	102.2	419.9	122.0	35.6
HYBLA S D	640 486	3 309	0.00516	100.9	420.0	121.3	32.6
HYBLA S D F	659 460	3 224	0.00488	101.7	416.7	114.0	24.5
HYBLA S F	663 258	3 261	0.00491	101.2	359.8	113.1	22.9
RENO P	292 644	1 517	0.00518	103.7	632.0	134.4	49.2
RENO S	299 720	1 518	0.00506	103.7	322.1	127.7	31.3
RENO S D	305 551	1 493	0.00488	103.9	330.0	126.8	30.0
RENO S D F	310 689	1 494	0.00480	103.8	423.6	122.1	26.2
RENO S F	307 787	1 512	0.00491	102.8	640.0	123.3	31.5
SCALABLE P	432 117	2 164	0.00500	103.9	640.0	129.8	48.6
SCALABLE S	448 877	2 254	0.00502	103.9	323.7	122.8	29.4
SCALABLE S D	458 946	2 240	0.00488	103.9	322.0	123.3	30.8
SCALABLE S D F	460 971	2 193	0.00475	102.8	329.1	116.8	22.5
SCALABLE S F	454 832	2 260	0.00496	103.7	401.3	116.9	23.3
VEGAS P	224 051	1 085	0.00484	103.9	519.5	136.5	46.0
VEGAS S	223 377	1 106	0.00495	103.9	330.1	132.7	39.0
VEGAS S D	219 631	1 116	0.00508	103.3	323.7	131.9	39.4
VEGAS S D F	219 505	1 131	0.00515	103.9	326.2	129.3	35.4
VEGAS S F	216 998	1 103	0.00508	103.9	322.8	131.0	37.9
WESTWOOD P	492 691	2 521	0.00511	103.1	626.2	130.1	51.6
WESTWOOD S	510 371	2 577	0.00504	101.8	395.1	119.8	29.6
WESTWOOD S D	515 738	2 544	0.00493	102.7	640.0	118.8	30.0
WESTWOOD S D F	523 922	2 593	0.00494	102.7	401.9	114.9	25.7
WESTWOOD S F	529 025	2 643	0.00499	102.4	411.1	115.2	23.3

Table 3.1: Thick streams 50 ms delay, 0.5% loss, Retransmission Statistics

throughput is high, more packets will be in flight. Thus, dupACKs are generated at a higher rate if a packet loss should occur. A fast generation of dupACKs means quicker feedback to the sender in form of 3 dupACKs, causing a retransmission. However, this does not hold when the throughput gets bigger.

None of the other plain protocols differ much from each other, even though their throughput does significantly. However, when SACK or FACK are enabled, the protocols with the highest throughput gather up in front, though not by much.

As mentioned earlier, the other protocol results for this environment vary little from each other (Figure 3.3), with only $136.5 - 113.1 = 23.4$ *ms* separating the top value (Hybla with SACK and FACK, 113.1 ms) from the bottom one (Vegas plain, 136.5 ms), and the minimum standard deviation value is approximately the same size, 22.5 ms.

Some interesting results can still be observed. In the following discussion, HSTCP is not considered due to it's abnormal behaviour compared to the other protocols. Every protocol shows a decrease in the RTD when SACK is activated (HSTCP too), even though it is not big (between 3.8 and 12.7 ms). Furthermore, an even smaller decrease is observed in enabling FACK (1.7-8 ms). STCP (129.8 ms) holds the top value for the plain protocols. Westwood then represents the top value until FACK is enabled, then Hybla obtains the pole position. Vegas remains safely at the other end for all options. However, these differences are too small to draw any conclusions, but the tendency is there.

Still, this corresponds well with the earlier prediction of the connection between throughput and RTD. Hybla (627 520-663 258) transmits a triplicate number of packets compared to Vegas (216 998-224 051), while Westwood (492 691-529 025) and STCP (432 117-460 971) are transmitting approximately twice as many. The other protocols stay around 300 000.

The reason is that the loss rate is high enough to have an effect on throughput. Westwood is favoured through it's bandwidth estimate detecting no bandwidth limitations, thus assuming packet loss did not occur due to congestion, and hence not reducing the cwnd as much. Hybla is comparing itself with an imaginary connection with an RTT of 25 ms, modifying it's growth functions to achieve an equal sending rate. Thus, it's ssthresh and growth rate in congestion avoidance are higher than the other protocols, resulting in a higher throughput. STCP's high throughput should indicate the invocation of it's own modified growth functions, providing a similar effect as Hybla's, though not as effective.

The protocols' minimum value is just above 100 ms, which is the delay imposed on the network (i.e., RTT), indicating the justification of the fast part in Fast Retransmit. Furthermore, even though the maximum values are quite high (from 322.0-728.8 ms), the average stays close to the minimum values, proving the efficiency and domination of the Fast Retransmit

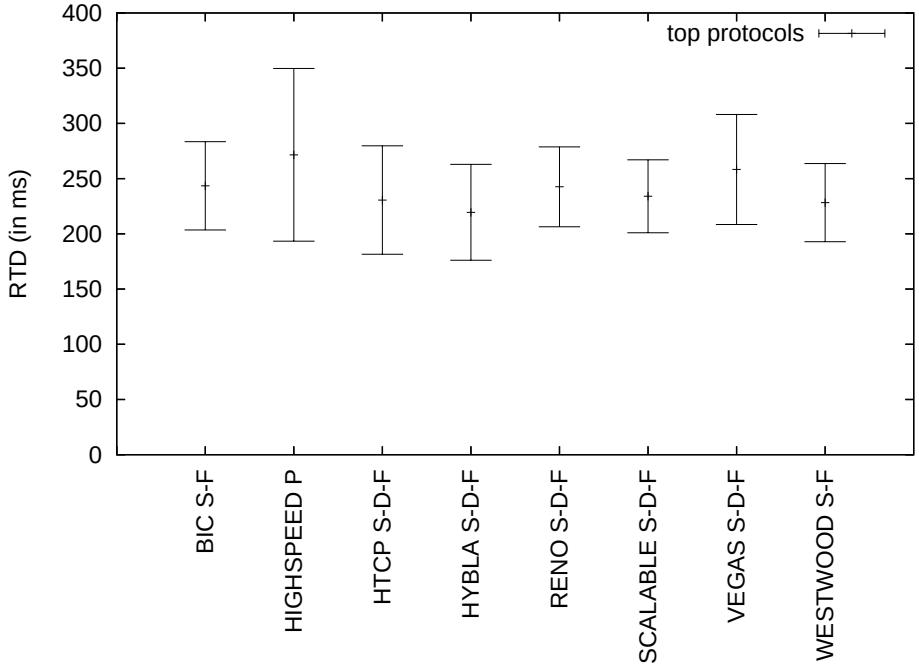

Figure 3.4: The top combination for each variation, thick streams, 100 ms delay, 0.5% loss

mechanism.

Thick streams, 100 ms delay, 0.5% loss

The tendency remains roughly the same when doubling the delay (Table 3.2), but the differences become clearer here (see Figure 3.4). The setting should now fully simulate a high-speed long distance network, bringing H-TCP up among the leading protocols. However, the difference in throughput is somewhat less than in the previous setting, except for Hybla which is crushing it's opponents.

HSTCP gets the same depressing results, except this time it is the plain protocol that is the lucky exception. Thus, it is again excluded from the following discussion.

Furthermore, SACK's impact gets bigger (especially on the top protocols) with the increase in RTT, as it should according to the protocol specification. More importantly, the top protocols fail to differ from the others without some SACK option enabled (i.e., the plain protocol).

BIC struggles to keep up with the other high-speed protocols in throughput, failing to distance itself from the remaining protocols. This might be due to the relatively high loss rate, which restricts the binary search pro-

TCP Variation	Packet Statistics			Retransmission Time Statistics (in ms)			
	Total	Rexmt	Rexmt rate	Min	Max	Avg	Stddev
BIC P	151 677	745	0.00491	204.2	1040.5	268.1	89.2
BIC S	156 567	732	0.00467	204.1	512.0	248.7	48.3
BIC S D	155 713	774	0.00497	204.2	448.0	246.6	46.6
BIC S D F	155 498	771	0.00495	204.2	544.0	243.5	43.0
BIC S F	153 230	752	0.00490	206.1	421.3	243.5	40.0
HIGH-SPEED P	128 125	635	0.00495	207.9	941.8	271.5	78.2
HIGH-SPEED	22 457	107	0.00476	208.0	866.3	502.4	89.4
HIGH-SPEED S D	22 244	116	0.00521	416.0	661.3	505.5	79.2
HIGH-SPEED S D F	22 799	119	0.00521	208.1	912.1	495.3	89.8
HIGH-SPEED S F	22 387	116	0.00518	260.1	880.0	504.6	98.1
H-TCP P	214 437	1 131	0.00527	206.0	1074.1	285.6	136.4
H-TCP S	222 983	1 163	0.00521	204.2	544.1	237.1	56.7
H-TCP S D	229 299	1 143	0.00498	204.2	618.4	237.2	57.4
H-TCP S D F	230 720	1 142	0.00494	204.2	624.0	230.6	49.1
H-TCP S F	229 452	1 142	0.00497	204.2	633.3	233.4	51.5
HYBLA P	364 130	1 813	0.00497	205.5	1246.3	278.8	157.6
HYBLA S	410 612	2 037	0.00496	204.2	633.4	226.2	55.3
HYBLA S D	395 366	2 096	0.00530	204.3	687.4	225.6	57.7
HYBLA S D F	415 155	2 046	0.00492	204.4	624.0	219.5	43.4
HYBLA S F	405 269	2 071	0.00511	204.1	626.7	221.2	48.6
RENO P	144 684	793	0.00548	208.0	979.0	272.4	78.4
RENO S	148 339	772	0.00520	207.7	492.0	253.0	50.0
RENO S D	150 892	753	0.00499	207.9	572.0	254.3	50.9
RENO S D F	149 591	787	0.00526	204.7	420.0	242.6	36.2
RENO S F	147 971	785	0.00530	208.0	423.1	244.2	35.6
SCALABLE P	214 772	1 304	0.00607	204.0	1380.1	334.2	147.5
SCALABLE S	221 377	1 081	0.00488	203.5	452.1	243.9	46.2
SCALABLE S D	221 349	1 126	0.00508	207.9	496.1	247.9	50.4
SCALABLE S D F	221 907	1 120	0.00504	206.9	511.4	234.0	33.0
SCALABLE S F	213 267	1 150	0.00539	206.5	533.0	236.0	33.0
VEGAS P	113 378	528	0.00465	207.9	1051.0	272.7	79.1
VEGAS S	110 842	562	0.00507	207.8	504.0	261.9	59.6
VEGAS S D	101 743	528	0.00518	206.0	624.0	271.4	60.7
VEGAS S D F	112 991	555	0.00491	208.0	427.0	258.3	49.8
VEGAS S F	108 007	569	0.00526	206.6	427.2	261.7	53.6
WESTWOOD P	238 393	1 248	0.00523	206.2	923.8	271.3	99.9
WESTWOOD S	249 560	1 304	0.00522	204.3	455.1	236.4	47.2
WESTWOOD S D	249 110	1 249	0.00501	204.1	452.1	238.0	52.2
WESTWOOD S D F	258 190	1 306	0.00505	204.2	625.0	229.4	37.2
WESTWOOD S F	256 856	1 260	0.00490	204.3	428.0	228.3	35.4

Table 3.2: Thick streams 100 ms delay, 5̇% loss, Retransmission Statistics

cess, and thus limits the cwnd size compared to the other high-speed protocols. The point is that these losses are random and do not indicate congestion, the wrong conclusion is drawn by the classic protocols.

The interval between the top value for each protocol, regardless of the individual SACK options, is quite small. Vegas with all SACK options enabled has the highest value of 258.3 ms, getting an interval of 38.8 ms. As soon as any SACK option is enabled, Hybla is the winner. In fact, these are the top 4 values in this setting. This makes sense, as the Hybla proposal strongly requests the use of SACK (see the Hybla description, Section 2.9).

The average RTD values still roughly correspond to the throughput.

Thick streams, 200 ms delay, 0.5% loss

No major deviations from the results seen earlier. The former evaluation fits quite well here too, although increasing the delay leads to an increase for everything else as well. The RTDs vary more, leading to higher standard deviations for the individual protocols. Therefore, even though the gap between Hybla and the other protocols' values is widening, this is partly covered by the increase in the standard deviations.

SACK's impact is even more massive in this setting, with a decrease in RTD between 112.9 and 279.9 ms. An interesting fact that arises in this setting is the dramatic drop in the number of retransmissions when a SACK option is enabled. The retransmission rate for the plain protocols stays between 0.883 and 1.619%, even though the imposed loss rate in the network is only 0.5%. However, enabling SACK reduces the retransmission rate to a rate that is within a close proximity of the imposed loss rate.

A strange incident occurs for some of the protocols though (6 of them, BIC Plain, HSTCP Plain, H-TCP Plain, Reno Plain, Hybla with SACK, DSACK, FACK and Westwood with SACK, DSACK). They get a minimum value which is lower than the imposed delay of the network, and thus lower than the supposed RTT. This is due to the retransmission of a packet that has just been sent. Actually, the arrival time of the ACK indicates that the first transmission in fact arrived safely and within reasonable time at the receiver. Thus, the retransmission was spurious.

Prior to the unnecessary retransmission of this packet, a considerable number of retransmissions of successive segments occur (see Figure 3.5, *packets 99490-99500*). This causes a time gap between the sending of any new packets. It is the first packet (*packet 99502*) sent after these retransmissions that is unnecessarily retransmitted (*packet 99536*) a couple of ms later (way below the RTT imposed on the network). Prior to this retransmission, an ACK (*packet 99526*) acknowledging the previous transmitted packet arrives, cumulatively acknowledging all the retransmitted packets sent. This ACK is followed by 4 dupACKs (*packets 99530, 99531, 99533, 99535*), triggered by the previous retransmissions. The problem is that each

```
# Connection (192.168.1.2:60745-192.168.2.2:36428)
# Retransmission region for packet 99536
# 99489     PKT t 1500.89 seq 88160505 len 1448
# 99490 RTR PKT t 1500.89 seq 88093897 len 1448
# 99491 DUP ACK t 1500.93 ack 88093897
# 99492 RTR PKT t 1500.93 seq 88095345 len 1448
# 99493 DUP ACK t 1500.93 ack 88093897
# 99494 RTR PKT t 1500.93 seq 88096793 len 1448
# 99495 DUP ACK t 1500.97 ack 88093897
# 99496 RTR PKT t 1500.97 seq 88098241 len 1448
# 99497 DUP ACK t 1500.97 ack 88093897
# 99498 RTR PKT t 1500.97 seq 88099689 len 1448
# 99499 DUP ACK t  1501.3 ack 88093897
# 99500 RTR PKT t  1501.3 seq 88101137 len 1448
# 99501 DUP ACK t  1501.3 ack 88093897
# 99502     PKT t  1501.3 seq 88161953 len 1448 <-- Transmitted
# 99503 DUP ACK t  1501.3 ack 88093897
# 99504     PKT t  1501.3 seq 88163401 len 1448
# 99505 DUP ACK t  1501.3 ack 88093897
# 99506     PKT t  1501.3 seq 88164849 len 1448
# 99507 DUP ACK t  1501.3 ack 88093897
# 99508     PKT t  1501.3 seq 88166297 len 1448
# 99509 DUP ACK t  1501.3 ack 88093897
# 99510     PKT t  1501.3 seq 88167745 len 1448
# 99511 DUP ACK t  1501.3 ack 88093897
# 99512     PKT t  1501.3 seq 88169193 len 1448
# 99513 DUP ACK t  1501.3 ack 88093897
# 99514     PKT t  1501.3 seq 88170641 len 1448
# 99515 DUP ACK t  1501.3 ack 88093897
# 99516     PKT t  1501.3 seq 88172089 len 1448
# 99517 DUP ACK t  1501.3 ack 88093897
# 99518 DUP ACK t  1501.3 ack 88093897
# 99519 DUP ACK t  1501.3 ack 88093897
# 99520 DUP ACK t  1501.3 ack 88093897
# 99521 DUP ACK t  1501.3 ack 88093897
# 99522 DUP ACK t  1501.3 ack 88093897
# 99523 DUP ACK t  1501.3 ack 88093897
# 99524 DUP ACK t  1501.3 ack 88093897
# 99525 DUP ACK t  1501.3 ack 88093897
# 99526     ACK t  1501.3 ack 88161953
# 99527     PKT t  1501.3 seq 88173537 len 1448
# 99528     PKT t  1501.3 seq 88174985 len 1448
# 99529     PKT t  1501.3 seq 88176433 len 1448
# 99530 DUP ACK t 1501.34 ack 88161953
# 99531 DUP ACK t 1501.34 ack 88161953
# 99532     PKT t 1501.34 seq 88177881 len 1448
# 99533 DUP ACK t 1501.38 ack 88161953
# 99534     PKT t 1501.38 seq 88179329 len 1448
# 99535 DUP ACK t 1501.38 ack 88161953
# 99536 RTR PKT t 1501.38 seq 88161953 len 1448  <-- Retransmitted
# 99537 DUP ACK t 1501.71 ack 88161953
# 99538     ACK t 1501.71 ack 88163401
#   Retransmission count: 1
#   Time between first trans and last retrans: 75.915 ms
#   Time between first trans and final ACK: 408.013 ms
```

Figure 3.5: Strange minimum value for BIC plain, extracted with **loss_details**

of these indicates the loss of the packet sent only a couple of ms ago, thus triggering the retransmission of this packet way too early.

Though this behaviour is quite odd, it does not affect the average RTD, as this only happens once per connection (twice for plain HSTCP), and none of RTDs of the involved protocols are among the top values (closer to the contrary in fact).

The one exception is Hybla with all SACK options enabled (Figure 3.6). It holds a minimum RTD (371.8 ms) that is a bit below the imposed delay. However, this is not due to same behaviour as outlined above. This seems in fact to be the use of New Reno's partial acknowledgments in practice. The segment is considered lost and then retransmitted (*packet 274070*), but other segments were previously lost and therefore have also retransmissions in flight (*packets 274042, 274077*). The ACK for these retransmissions is then received (*packet 274078*), but it does not cover the last retransmitted packet (*packet 274070*). This is because the time passed since the last retransmission took place is below the RTT (371 ms, RTT is approximately 400), thus the acknowledgment for this retransmission has not had a chance to arrive yet, but regardless of this, the segment is still instantly retransmitted again (*packet 274079*). Approximately 30 ms later, an ACK acknowledging the just retransmitted packet arrives (*packet 274080*), probably triggered by the arrival of the first retransmission (*packet 274070*), indicating that a spurious retransmission just took place. The packets from *273994-274040* and *274044-274068* were dupACKs acknowledging the same byte *240157377*, and were replaced with ..... instead.

Thick streams, (wireless) 0 ms delay, 5% loss

A high loss rate and no delay should simulate a wireless LAN (through the high loss rate), supposedly favourizing Westwood. With no delay, none of the protocols designed for high BDP networks should have any particular advantages. This prediction holds for the test results (Table 3.3). Westwood holds the top value for each option, though it distances itself from the others only after the SACK options are enabled.

The SACK option still has a major impact, with a solid increase in throughput and corresponding decrease in the average RTD. The exceptions are the still struggling HSTCP protocol, now partly joined by Vegas. This should be due to Vegas considerate congestion avoidance strategy, heavily penalizing the protocol when the loss rate is high. Westwood gets the biggest boost by the enabling of SACK, quintupling the sending rate, further increased by the enabling of FACK.

Westwood's results are quite impressive, with an average RTD between 6.4-10.6 ms when the SACK options are enabled. These also hold the lowest standard deviations with values between 33.1-52.3 ms. The only protocol that can partly keep up is STCP, with average RTDs of 19.3-25.2 ms with

```
# Connection (192.168.1.2:32963-192.168.2.2:53985)
# Retransmission region for packet 274079
#273982     PKT t 1414.75 seq 240271769 len 1448
#273983     PKT t 1414.75 seq 240273217 len 1448 <-- Transmitted
#273984     PKT t 1414.75 seq 240274665 len 1448
#273985     PKT t 1414.75 seq 240276113 len 1448
#273986     PKT t 1414.75 seq 240277561 len 1448
#273987     PKT t 1414.75 seq 240279009 len 1448
#273988     PKT t 1414.75 seq 240280457 len 1448
#273989     PKT t 1414.75 seq 240281905 len 1448
#273990     PKT t 1414.75 seq 240283353 len 1448
#273991 DUP ACK t 1414.75 ack 240157377
#273992 RTR PKT t 1414.75 seq 240157377 len 1448
#273993 DUP ACK t 1414.75 ack 240157377
.......
#274041 DUP ACK t 1415.12 ack 240157377
#274042 RTR PKT t 1415.12 seq 240232673 len 1448
#274043 DUP ACK t 1415.12 ack 240157377
.......
#274069 DUP ACK t 1415.15 ack 240157377
#274070 RTR PKT t 1415.15 seq 240273217 len 1448 <-- 1. Retransmission
#274071 DUP ACK t 1415.15 ack 240157377
#274072 DUP ACK t 1415.15 ack 240157377
#274073 DUP ACK t 1415.15 ack 240157377
#274074 DUP ACK t 1415.15 ack 240157377
#274075 DUP ACK t 1415.15 ack 240157377
#274076 DUP ACK t 1415.15 ack 240157377
#274077 RTR PKT t 1415.37 seq 240157377 len 1448
#274078     ACK t 1415.53 ack 240273217
#274079 RTR PKT t 1415.53 seq 240273217 len 1448 <-- 2. Retransmission
#274080     ACK t 1415.56 ack 240284801
#   Retransmission count: 2
#   Time between first trans and last retrans: 778.106 ms
#   Time between first trans and final ACK: 813.944 ms
```

Figure 3.6: Minimum value for Hybla SACK DSACK FACK, extracted with **loss_details**

TCP Variation	Packet Statistics			Retransmission Time Statistics (in ms)			
	Total	Rexmt	Rexmt rate	Min	Max	Avg	Stddev
BIC P	274 150	14 362	0.05238	0.5	3233.7	53.7	132.0
BIC S	388 617	20 104	0.05173	0.6	3241.1	38.5	111.6
BIC S D	387 099	19 952	0.05154	0.7	5434.5	39.3	118.1
BIC S D F	428 230	21 874	0.05108	0.5	5675.1	35.6	120.3
BIC S F	454 152	23 097	0.05085	0.5	3239.9	34.7	104.8
HIGH-SPEED P	115 261	6 504	0.05642	0.5	2392.1	106.4	160.8
HIGH-SPEED	121 841	6 926	0.05684	0.9	3799.7	93.1	159.9
HIGH-SPEED S D	120 310	6 766	0.05623	0.9	7552.5	95.7	175.9
HIGH-SPEED S D F	117 876	6 526	0.05536	0.9	7488.5	99.3	209.7
HIGH-SPEED S F	122 260	6 808	0.05568	0.9	5473.2	95.4	172.6
H-TCP P	280 497	14 693	0.05238	0.5	3659.7	51.5	127.6
H-TCP S	424 928	21 933	0.05161	0.7	3298.0	35.9	105.5
H-TCP S D	403 847	20 960	0.05190	0.7	6656.7	38.9	115.6
H-TCP S D F	451 341	23 169	0.05133	0.5	2044.6	33.5	98.4
H-TCP S F	455 290	23 221	0.05100	0.5	5888.4	32.8	114.3
HYBLA P	288 316	15 112	0.05241	0.5	2000.1	45.7	104.9
HYBLA S	331 552	17 121	0.05163	0.8	2999.0	41.6	104.7
HYBLA S D	319 418	16 503	0.05166	0.8	3237.9	43.3	112.1
HYBLA S D F	356 801	18 555	0.05200	0.5	1511.7	38.1	98.1
HYBLA S F	359 685	18 684	0.05194	0.5	1512.4	38.0	99.5
RENO P	302 867	15 714	0.05188	0.5	2997.1	46.6	113.5
RENO S	340 097	17 432	0.05125	0.7	13608.2	42.1	188.3
RENO S D	341 468	17 546	0.05138	0.7	6912.4	40.1	121.0
RENO S D F	372 521	19 126	0.05134	0.5	3238.4	36.9	108.0
RENO S F	402 113	20 416	0.05077	0.5	3456.2	34.6	98.8
SCALABLE P	333 953	17 271	0.05171	0.5	3239.4	48.3	121.9
SCALABLE S	590 865	30 321	0.05131	0.7	1896.1	25.2	85.7
SCALABLE S D	630 837	31 971	0.05068	0.7	2544.2	23.2	80.4
SCALABLE S D F	720 229	36 618	0.05084	0.5	3443.3	21.1	87.7
SCALABLE S F	735 812	37 267	0.05064	0.5	5001.4	19.3	79.7
VEGAS P	229 439	12 273	0.05349	0.5	1728.1	56.4	119.9
VEGAS S	235 247	12 562	0.05339	0.6	3415.7	56.6	135.2
VEGAS S D	250 181	13 249	0.05295	0.6	3456.2	52.4	124.7
VEGAS S D F	260 555	13 684	0.05251	0.5	3237.5	49.7	117.4
VEGAS S F	264 405	13 773	0.05209	0.5	2560.2	49.6	115.1
WESTWOOD P	367 687	18 814	0.05116	0.5	6693.5	45.5	131.7
WESTWOOD S	1 356 888	68 474	0.05046	0.5	1728.1	10.6	49.9
WESTWOOD S D	1 359 979	68 399	0.05029	0.6	3229.5	10.4	52.3
WESTWOOD S D F	2 058 230	102 901	0.04999	0.6	1729.0	6.4	33.1
WESTWOOD S F	1 879 806	93 675	0.04983	0.5	1728.1	7.0	38.9

Table 3.3: Thick streams 0 delay, 5% Loss, Retransmission Statistics

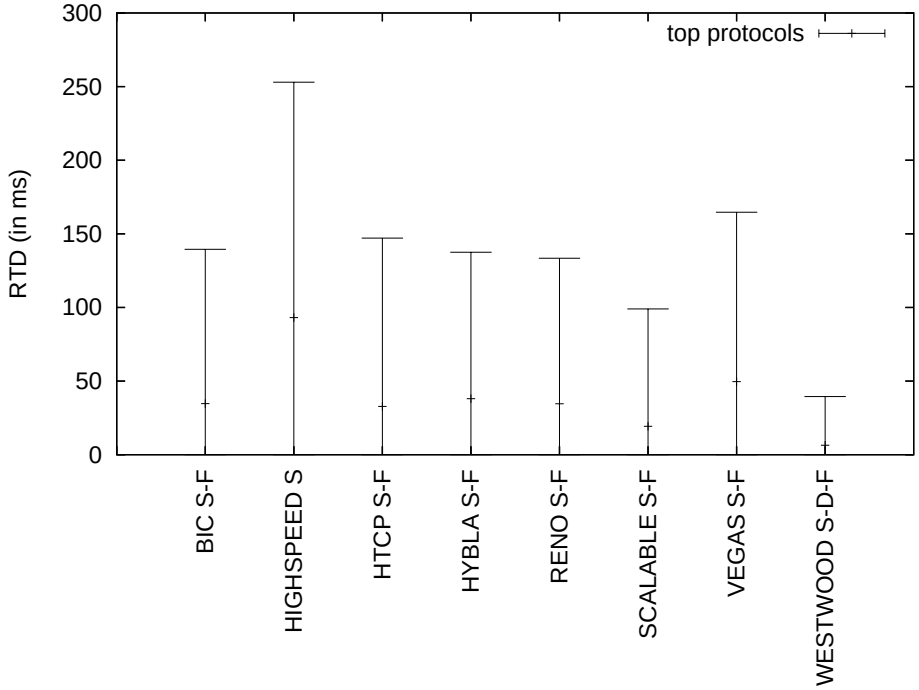

Figure 3.7: The top combination for each variation, thick streams, 0 ms delay, 5% loss

the SACK options enabled (see Figure 3.7), but with standard deviations between 79.7-87.7 ms. The other protocols mostly stay in the region 35-50 ms. Hyblas downfall is due to the lack of delay, as it needs at least a 25 ms RTT to activate it's improved mechanisms. Otherwise, it acts like the default New Reno.

Thick streams, (wireless) 50 ms and 100 ms delay, 5% loss

Adding delays of 50 and 100 ms bring things "back to normal". The results closely resemble those collected for the same settings with 0.5% loss. The only major difference is the increase of the loss rate causes greater variations in the RTDs, leading to a higher average and standard deviation for each protocol. Still, the surroundings seems to be close to the limit of what some of the protocols can handle satisfactory, as some of the averages and standard deviations are getting quite high. This is directly related to the throughput rate, and thus the throughput now have a major influence on the RTD. Of course, Hybla is benefitting on this.

Thick stream test evaluation

The tests were performed, and are thus mainly considered, with respect to the RTD. When examining the results with respect to this, some but no dramatic differences between the protocols exist. The Fast Retransmit mechanism seems to be performing adequately, and is thus the dominant controlling factor when it comes to the RTD. The retransmission timeout mechanism is seldom invoked, as should be the case in a thick stream scenario.

Enabling SACK can be confirmed to be a considerable improvement in every setting, thus it is strongly recommended when available. It's impact just increases with the RTT, as it should. FACK provides a further improvement, though a smaller one, and is thus recommended as the preferred SACK option whenever available.

Throughput only plays a small part in affecting the RTD, but as the effect is a positive one, it is well worth considering. Thus, Hybla seems like an excellent choice when the RTT exceeds 25 ms. This could be expected, as it is the newest protocol and thus has been able to observe the others' weaknesses and avoid these in it's proposal. At the other end, HSTCP does not impress and gives a rather negative impression. Vegas is not as bad, but it's considerate tendencies emerge in the high loss scenarios, weakening it's impression and strengthening the allegations, presented earlier in Section 2.2, against it.

Otherwise, Westwood makes a fairly good impression, though favoured in these tests, due to the way packet loss is simulated. Especially in the network setting with no delay and 5% loss, it's performance is impressive. Of the remaining high-speed protocols, H-TCP and STCP perform quite well too, while BIC (default in the Linux 2.6.15 kernel) is a bit disappointing as it fails to keep up with the others.

Still, the optimization of the RTD seems to be fairly well covered for thick streams, leaving limited room for further major improvements. This is of course provided that the right protocol is used in the specific setting. Not all of the variations perform satisfactory in all settings, but there are always some of them that do.

3.2.2 Evaluation of the TCP variations for thin streams

To simulate thin streams, netperf was used to limit the number of packets sent each second to 4 messages of 100 B each. The 4 messages are sent in one burst. The choice is based on findings in [34], observing that Anarchy Online traffic was limited to a few packets sent per second, each packet containing very little data (about 120 B data on average). However, limiting the number of packets means fewer samples that might be lost, thus fewer retransmissions. Hence, longer time is needed to collect an accept-

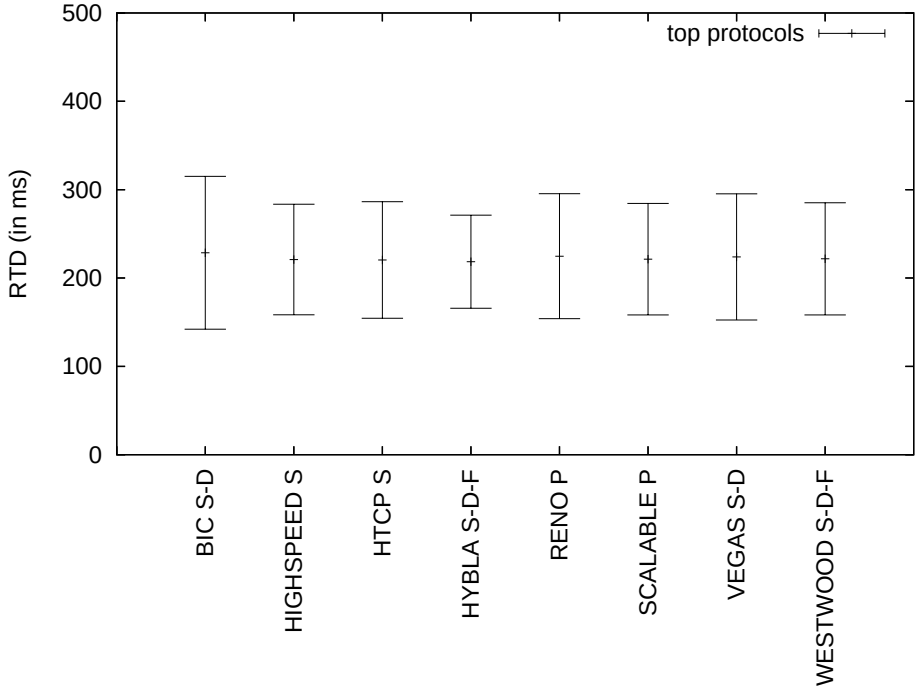

Figure 3.8: The top combination for each variation, Thin Streams, 0 ms delay, 5% loss

able number of loss events. In order to avoid an escalation of the time needed to run the tests, the loss rate used was quite high.

Thin streams were tested for a delay of 0, 50, 100 and 200 ms, with a loss rate of 5 and 10%. For 5% loss, the tests were run for 30 min each, while 20 min were chosen for 10%.

0 ms delay, 5% loss

The results (Table 3.4) show no significant differences between the protocols, at least no indisputable ones.

The Linux 2.6.15 kernel implements the TCP New Reno version. When a given TCP variation is enabled, only the methods that it has it's own modifications for are overridden. Otherwise, it is using the methods already present (i.e., New Reno's). When looking at the different variations, they are only concerned with various strategies for congestion control issues, in particular for adjusting the cwnd and ssthresh in specific situations. 3 of the 8 protocols tested apply their modifications only in high-speed settings, and otherwise act like Reno (use Reno's methods, provided Reno is the implemented default). These are HSTCP, STCP and H-TCP. As it is the New Reno version which is implemented here, these 3 act like New Reno

TCP Variation	Packet Statistics			Retransmission Time Statistics (in ms)			
	Total	Rexmt	Rexmt rate	Min	Max	Avg	Stddev
BIC P	3 983	391	0.09816	202.8	2998.9	244.0	192.3
BIC S	3 995	397	0.09937	202.5	816.1	231.6	86.2
BIC S D	4 011	413	0.10296	202.7	816.1	228.5	86.5
BIC S D F	3 994	396	0.09914	212.3	864.1	242.0	84.4
BIC S F	3 972	376	0.09466	200.9	816.1	228.8	92.1
HIGH-SPEED P	3 999	409	0.10227	203.0	1632.1	239.2	118.1
HIGH-SPEED S	3 960	357	0.09015	201.3	816.1	220.9	62.6
HIGH-SPEED S D	4 034	422	0.10461	201.2	816.1	231.4	85.2
HIGH-SPEED S D F	3 969	366	0.09221	202.6	816.1	227.2	75.0
HIGH-SPEED S F	3 983	381	0.09565	202.4	816.1	231.1	82.6
H-TCP P	4 008	406	0.10129	203.0	816.1	220.4	66.0
H-TCP S	3 938	342	0.08684	203.3	816.1	227.7	85.1
H-TCP S D	3 980	380	0.09547	202.8	816.1	223.7	70.7
H-TCP S D F	4 004	407	0.10164	203.0	816.1	227.4	82.0
H-TCP S F	3 984	384	0.09638	202.4	816.1	231.0	86.2
HYBLA P	3 988	392	0.09829	200.2	1632.1	232.9	118.6
HYBLA S	3 978	391	0.09829	200.4	3264.2	241.4	190.7
HYBLA S D	4 006	389	0.09710	202.8	816.1	223.7	65.8
HYBLA S D F	3 971	363	0.09141	202.1	408.0	218.4	52.7
HYBLA S F	4 009	408	0.10177	202.8	816.1	226.8	73.6
RENO P	4 039	438	0.10844	201.1	816.1	224.7	70.7
RENO S	3 979	383	0.09625	205.7	1664.1	238.8	116.9
RENO S D	4 000	399	0.09975	202.5	816.1	229.0	79.9
RENO S D F	3 989	393	0.09852	214.2	3456.2	258.6	216.2
RENO S F	3 998	402	0.10055	213.8	864.1	245.4	95.0
SCALABLE P	3 965	361	0.09104	202.6	816.1	221.3	63.1
SCALABLE S	3 950	384	0.09721	214.4	12240.8	294.0	699.0
SCALABLE S D	3 997	401	0.10032	200.2	1632.1	232.8	109.4
SCALABLE S D F	3 982	385	0.09668	202.5	1632.1	228.7	104.8
SCALABLE S F	4 019	405	0.10077	202.2	816.1	222.4	68.6
VEGAS P	3 932	343	0.08723	201.8	1632.1	239.0	155.6
VEGAS S	3 987	386	0.09681	200.4	816.1	227.2	74.5
VEGAS S D	3 970	365	0.09193	202.3	816.1	223.9	71.4
VEGAS S D F	3 975	373	0.09383	214.8	864.1	236.7	74.8
VEGAS S F	3 997	382	0.09557	213.4	864.1	235.6	73.3
WESTWOOD P	3 986	387	0.09708	214.6	864.1	242.6	89.4
WESTWOOD S	3 976	367	0.09230	213.6	864.1	237.6	80.9
WESTWODD S D	4 030	434	0.10769	202.6	816.1	232.1	88.7
WESTWOOD S D F	3 968	365	0.09198	202.3	816.0	221.7	63.5
WESTWOOD S F	3 963	366	0.09235	201.4	1632.1	234.5	110.4

Table 3.4: Thin stream (0 ms delay, 5% loss) Retransmission Statistics

in this thin stream setting. However, even though the other 4 variations use their own modified methods, this should not make any difference as they are related to adjusting the cwnd and ssthresh. In this thin stream setting, with a needed throughput of only 400 Bps, the amount of data requested to be transmitted is below the segment size and should therefore typically be far less than the cwnd, regardless of the variation used. Thus, the different variations' various adjustments do not have any impact in this setting, and their behaviour should be approximately the same. This should produce approximately the same results, only with individual test variations separating them. Figure 3.8 visualizes this fact quite well.

Some relevant observations can still be made. SACK does not necessarily decrease the RTD in this setting, 4 of the protocols get a considerable increase instead. And there is no clear pattern indicating improvement when any of the SACK option combinations are enabled. Plain H-TCP (220.4 ms) and Reno (224.7 ms) get almost as good results as the top value, Hybla (218.4 ms) with all SACK options enabled. The interval between top and bottom values is 75.6 ms, but the bottom value, STCP with SACK, contains a potential outlier for the maximum (will soon be explained). Removing this value from the result drops the average from 294.0 to 262.8 ms, reducing the interval to 44.4 ms. The minimum standard deviation is 52.7 ms, making it difficult do draw any conclusions.

The one extremely interesting fact is that none of the protocols achieve an RTD below 200 ms, even though there is no delay. Presumably, this indicates the invocation of the RTO as the only detector of packet loss (which is set to minimum Hz/5 which means 200 ms). This makes sense, since only 4 messages are sent per second. The messages are transmitted in one burst each second, making it possible to bundle several messages into one packet if allowed by the cwnd size. This can be seen in the results, as the number of packets transmitted stays around 3900 (this number include retransmitted packets). The number to be expected if each message was sent in an individual packet should be $1800 * 4 = 7200$, added with the number of retransmitted packets.

Since the maximum size needed to be sent within one cwnd is 400 B (all 4 messages) and the MSS of one packet is 1448 B, only one packet will be sent per cwnd independently of it's size. If the cwnd is less than 400 B, all the allowed data will still be transmitted in a single packet. Thus, a dupACK is seldom triggered as it requires the reception of a second packet succeeding a packet loss. In fact, most of the runs did not contain a single dupACK. And when there are almost no dupACKs, Fast Retransmit will not be triggered, leaving the RTO as the only means to discover packet loss.

Using **loss_details**, detailed loss information could be extracted, but additionally, this provided a good impression of the traffic flow. It showed that the messages were transmitted in 2 packets, the first contained only the

```
# Connection (192.168.1.2:38888-192.168.2.2:55066)
# Retransmission region for packet 26
#    22     PKT t 8.00558 seq 2001 len 100
#    23     ACK t 8.00583 ack 2101
#    24     PKT t 8.00587 seq 2101 len 300 <-- Transmitted
#    25     PKT t 12.0058 seq 2401 len 1448
#    26 RTR PKT t 14.1258 seq 2101 len 300 <-- 1. Retransmission
#    27 RTR PKT t 26.3666 seq 2101 len 300 <-- 2. Retransmission
#    28     ACK t 26.3669 ack 3849
#   Retransmission count: 2
#   Time between first trans and last retrans: 18360.7 ms
#   Time between first trans and final ACK: 18361.1 ms
```

Figure 3.9: Strange maximum value for STCP with SACK, extracted with **loss_details**

first message, while the second contained the rest. Additionally, the second packet was not transmitted until the first one had been acknowledged.

When an RTO occurs, which will be quite frequent in the absence of Fast Retransmit, the unacknowledged packet is retransmitted and the cwnd is reset to 1. The RTO is doubled [13] until an RTT sample can be calculated based on a received ACK. Thus, the RTO increases exponentially for each retransmission of the same segment (**Exponential Backoff**), explaining the high maximum values for the tests. Most of these values correspond to their individual test's maximum number of retransmissions. However, the average RTDs stay close to the minimum value, indicating that the Exponential Backoff RTO is not triggered very often.

It is worth noting that the latency experienced by the user would be considerably higher than the maximum value shown in the tables. If it represents the first retransmission, only the RTT needs to be added to the value. However, if this is the second retransmission, the time elapsed for the first retransmission must be added as well, in addition to the RTT. If it is the third, the time for the first and second retransmission must be added, and so on.

The only protocols which diverge dramatically are plain BIC and STCP with SACK in their first retransmission scenarios. BIC has to wait 3 seconds for the RTO to expire in it's first retransmission scenario. This is because the RTO is initally set to 3 s, as recommended in [9]. If packet loss occurs before any RTO estimates can be made (requires RTT samples), the first RTD will be approximately 3 s. This is what happens in BIC's case.

STCP's value is a bit more odd, as it for some reason has to wait 6(!) seconds for the RTO to expire in it's first retransmission scenario (Figure 3.9). To make it worse, the retransmitted packet is lost, forcing the sender to wait another 12 seconds due to the doubling of the RTO. As the first RTO is exactly twice that of the supposed initialized value, it seems that the RTO for some reason is doubled without any retransmissions having been

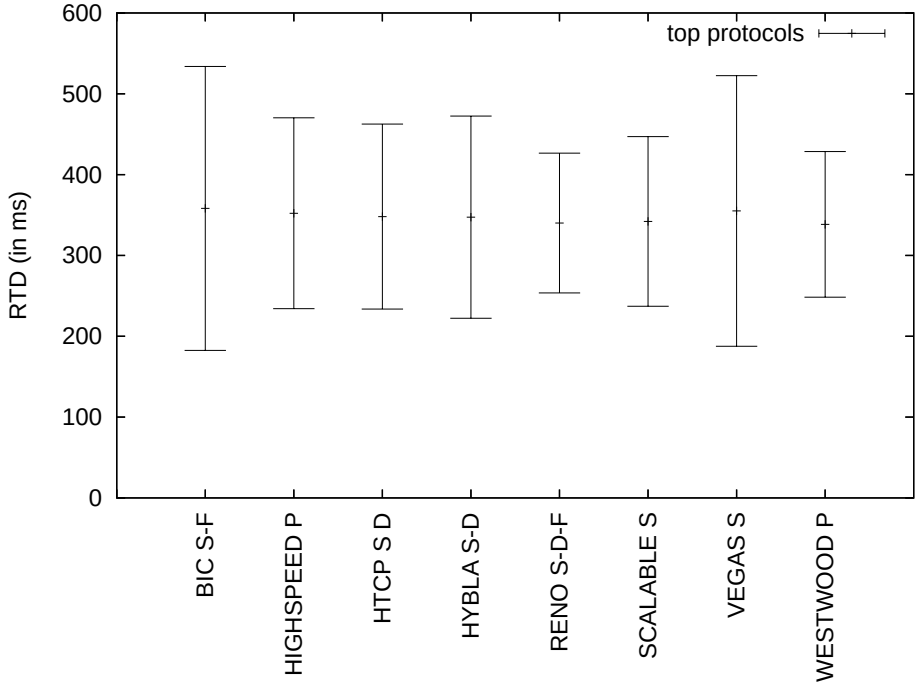

Figure 3.10: The top combination for each variation, Thin Streams, 50 ms delay, 5% loss

triggered. The total latency experienced by the user, would in this case have been approximately 18 s. However, this "abnormality" has no effect as it is the only aberration that can be found in the results. As mentioned earlier, removing the extreme value from STCP drops it's RTD average to a level within a close proximity of the others.

Since the RTO is the only detector of packet loss, the difference between the protocols is limited. The RTO is independent of any of their individual enhancements, and only affected by the RTT, which has nothing to do with any of the protocols, and thus does not vary among them. The small variations that can be seen are due to the protocols' varying maximum RTD value, i.e., the maximum number of retransmissions of one segment. This might be a bit arbitrary as the drop sequence is not the same for the individual runs, only the drop frequency will be approximately equal. Thus, a continuing loss of the retransmissions for one segment will lead to a high maximum RTD value, resulting in a higher average RTD.

50 ms delay, 5% loss

The results (Table 3.5 and Figure 3.10) deviate little from those found in the previous setting. The differences get even smaller in this setting, except for

TCP Variation	Packet Statistics			Retransmission Time Statistics (in ms)			
	Total	Rexmt	Rexmt rate	Min	Max	Avg	Stddev
BIC P	3 867	387	0.10007	308.1	10759.2	439.9	849.1
BIC S	3 890	383	0.09845	308.1	7044.6	406.7	577.6
BIC S D	3 897	352	0.09032	308.1	11108.4	381.9	600.6
BIC S D F	3 942	398	0.10096	308.1	2996.5	359.7	185.3
BIC S F	3 907	363	0.09291	308.1	2528.2	358.2	175.7
HIGH-SPEED P	3 946	403	0.10212	308.1	1264.1	352.2	118.1
HIGH-SPEED S	3 927	391	0.09956	308.1	2528.2	364.0	177.9
HIGH-SPEED S D	3 922	388	0.09892	308.1	1264.1	358.6	141.1
HIGH-SPEED S D F	3 930	411	0.10458	308.1	4992.3	375.9	290.2
HIGH-SPEED S F	3 910	369	0.09437	308.1	2997.7	360.6	182.2
H-TCP P	3 915	369	0.09425	308.1	2528.2	354.6	165.9
H-TCP S	3 947	408	0.10336	308.1	2528.2	356.6	167.3
H-TCP S D	3 920	364	0.09285	308.1	1264.1	348.1	114.5
H-TCP S D F	3 876	352	0.09081	308.1	3064.2	382.7	250.9
H-TCP S F	3 916	395	0.10086	308.1	2824.2	377.7	234.4
HYBLA P	3 914	375	0.09580	308.1	1264.1	365.0	155.4
HYBLA S	3 937	398	0.10109	308.1	2528.2	360.5	177.1
HYBLA S D	3 974	424	0.10669	308.1	1264.1	347.3	125.1
HYBLA S D F	3 967	415	0.10461	308.1	2528.2	350.8	162.6
HYBLA S F	3 921	364	0.09283	308.1	1264.1	348.6	125.1
RENO P	3 920	393	0.10025	304.1	2496.2	360.2	176.5
RENO S	3 917	393	0.10033	308.1	1264.1	364.7	155.1
RENO S D	3 962	416	0.10499	308.1	1264.1	344.5	100.3
RENO S D F	3 953	397	0.10043	308.1	632.0	340.1	86.4
RENO S F	3 916	380	0.09703	308.1	5056.3	365.9	291.4
SCALABLE P	3 917	373	0.09522	308.1	1264.1	352.4	124.0
SCALABLE S	3 968	418	0.10534	308.1	1264.1	342.1	104.9
SCALABLE S D	3 917	368	0.09394	308.1	1264.1	350.5	122.7
SCALABLE S D F	3 922	373	0.09510	308.1	1264.1	347.8	112.7
SCALABLE S F	3 939	386	0.09799	308.1	1264.1	345.5	109.6
VEGAS P	3 914	381	0.09734	308.1	1264.1	361.8	154.2
VEGAS S	3 939	396	0.10053	308.1	2528.2	355.1	167.4
VEGAS S D	3 888	346	0.08899	308.1	2528.2	359.4	172.2
VEGAS S D F	3 906	369	0.09447	308.1	2998.1	368.7	246.9
VEGAS S F	3 922	377	0.09612	308.1	1264.1	358.5	147.3
WESTWOOD P	3 938	381	0.09674	304.1	624.0	338.4	90.1
WESTWOOD S	3 919	396	0.10104	308.1	1264.1	358.1	128.6
WESTWODD S D	3 907	350	0.08958	308.1	1264.1	349.5	122.9
WESTWOOD S D F	3 967	423	0.10662	308.1	1264.1	347.6	123.0
WESTWOOD S F	3 947	412	0.10438	308.1	1264.1	350.7	107.7

Table 3.5: Thin stream (50 ms delay, 5% loss) Retransmission Statistics

```
# Connection (192.168.1.2:34765-192.168.2.2:40947)
# Retransmission region for packet 9
#     2     PKT t 4.31193 seq    1 len 100 <-- Transmitted
#     3     PKT t 7.31359 seq  101 len 1448
#     4 DUP ACK t 7.41585 ack    1
#     5     PKT t 11.3138 seq 1549 len 1448
#     6 DUP ACK t 11.4201 ack    1
#     7     PKT t 15.3141 seq 2997 len 1448
#     8 DUP ACK t 15.4203 ack    1
#     9 RTR PKT t 15.4203 seq    1 len 100 <-- Retransmitted
#    10     ACK t 15.5243 ack 4445
#   Retransmission count: 1
#   Time between first trans and last retrans: 11108.4 ms
#   Time between first trans and final ACK: 11212.4 ms
```

Figure 3.11: Strange maximum value for BIC with SACK DSACK, extracted with **loss_details**

BIC which gets some pretty high values (439.9 ms at the most), until FACK is enabled. However, this is due to similar abnormalities as the ones experienced by STCP (with SACK) and BIC in the previous setting. Additionally, the succeeding retransmissions are influenced by the extreme RTO value, which affects the following RTOs, and thus the RTD. H-TCP with FACK also experiences the abnormality, but the RTO value is not as extreme as BIC's, and thus the average RTD is not as heavily penalized.

Interestingly enough in this scenario, a packet is sent every 4 seconds while waiting for the ACK, containing 1448 B (Figure 3.11, *packets 3, 5, 7*). This seems to occur when the messages requested to be transmitted exceed the MSS of one packet (1448 B), thus triggering the transmission of the packet. When a packet containing only the first message in a burst is lost, a transmission can be triggered after 3 seconds, since 3 messages of 100 B each are still waiting to be transmitted, and 3 additional bursts of 400 B each are then more than enough to fill a packet. The RTO is for some reason not triggered even though 4 s pass between each transmission of a full packet, indicating that the RTO in this case must have been higher than 4 s. Additionally, it is then reset. This transmission will result in the reception of a dupACK (happens here, *packets 4, 6, 8*), if neither the packet nor the dupACK are lost on the way. Thus, Fast Retransmit might eventually be triggered if the RTO does not expire before 3 dupACKs are received, but since the sender is only transmitting once every 4 seconds (at best 3), it is not particularly fast. This is the case for BIC with SACK and DSACK.

For BIC with SACK, only 1 full packet is sent and 1 dupACK for this is received before the RTO suddenly goes off, 3 seconds after the transmission of the full packet. However, 4 s have still passed before the full packet was sent without the RTO going off. When the full packet is sent the RTO is reset to 3 s.

Common for all runs affected by some abnormality is that it is in their

first retransmission scenario the problems arise. This strongly indicates that the problems are related to the initial 3 s value of the RTO. The simplest and least penalizing incident is if the RTO is triggered after 3 s, or close to this value. The following RTO samples are affected by the initial value, and thus packet loss in this phase will not be as quickly discovered. Additionally, exponential backoff in this phase causes heavy penalization, but these values are all explainable.

It is considerably more difficult to explain why the RTO gets above 3 s initially. For some reason the RTO is higher, even though no prior retransmissions have taken place, that could have increased the value. However, accepting that this occurs explain the successive behaviour. These abnormalities are occasionally present throughout the tests.

The minimum RTD increases approximately with the delay introduced, as expected. The average increases a bit more with the top value being 338.4 (plain Westwood), 120 ms higher than in the previous setting. The interval between top and bottom is 101.5 ms, but excluding BIC's high values drops the interval to 44.3 ms, and the minimum standard deviation is still 86.4 ms. The SACK options fail to produce any detectable improvements in this setting as well.

100 and 200 ms delay, 5% loss

There is nothing new to report in these settings, except that as the RTO increases with the delay, and that the effect of multiple retransmissions of the same segment becomes more significant.

0 ms delay, 10% loss

Increasing the loss rate to 10 % should result in a higher number of retransmissions, providing more samples. Thus, the time needed to collect the necessary number of samples is shorter and the length of each run was reduced from 30 to 20 minutes.

Doubling the loss rate has a considerable impact on the range of values, leading to a dispersal of the values (Figure 3.12). However, this makes sense, since the higher loss rate means more packets lost, which should cause a higher number of retransmissions of the same packet. This results in more dispersed RTD samples, reflected in the higher maximum RTD value. Additionally, some of the runs suffer from the by now familiar abnormality.

But as for the others, there are no significant differences between the protocols. HSTCP with SACK is at the top with 247.1 ms, also holding the minimum standard deviation of 116.2 ms. Removing the extreme samples from the affected runs drops their RTD average down among the others. The remaining differences between the protocols can be explained by the

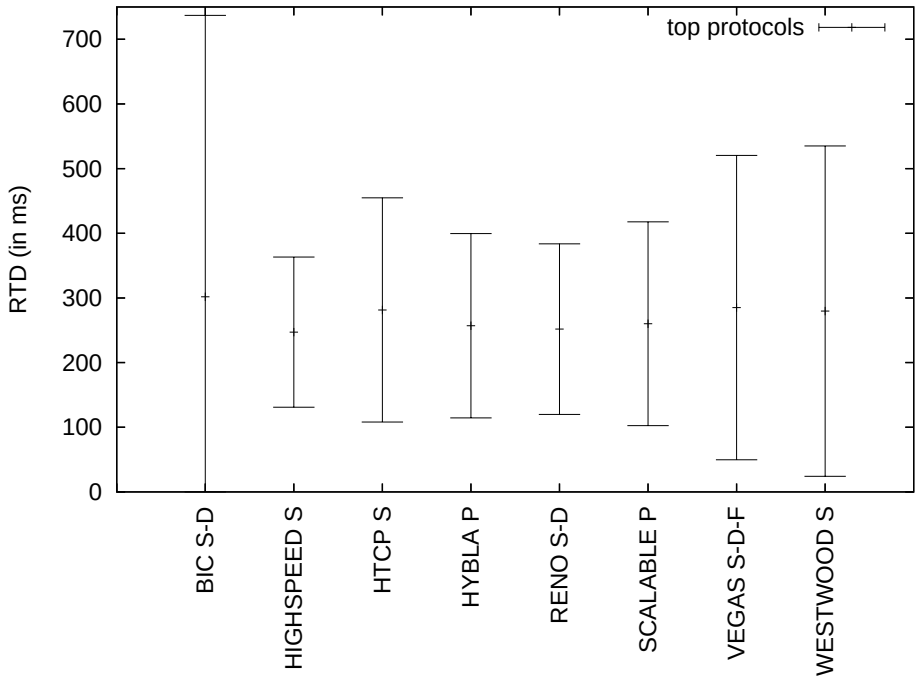

Figure 3.12: The top combination for each variation, Thin Streams, 0 ms delay, 10% loss

protocols' varying maximum RTD value, as ranking the protocols by average or maximum RTD produces more or less the same result list.

50, 100 and 200 ms delay, 10% loss

As this remains the case for the following settings, there is no need to go into details for any of these. They do not provide any new information. Figure 3.13 provides a good representation for the plots, with similar averages and high standard deviations.

Thin streams test evaluation

For the thin stream settings, it is hard to find any differences between the protocols. That is, some differences exist, but they are not at all consistent and seem to be arbitrary for the settings. It seems quite random which one ends up at the top in each setting, as no protocol remains among the top protocols throughout the tests.

However, this is expected as none of them are designed for thin streams, thus none of them contain any advantages compared to the others. As mentioned earlier, most of them are designed for high-speed environments, and

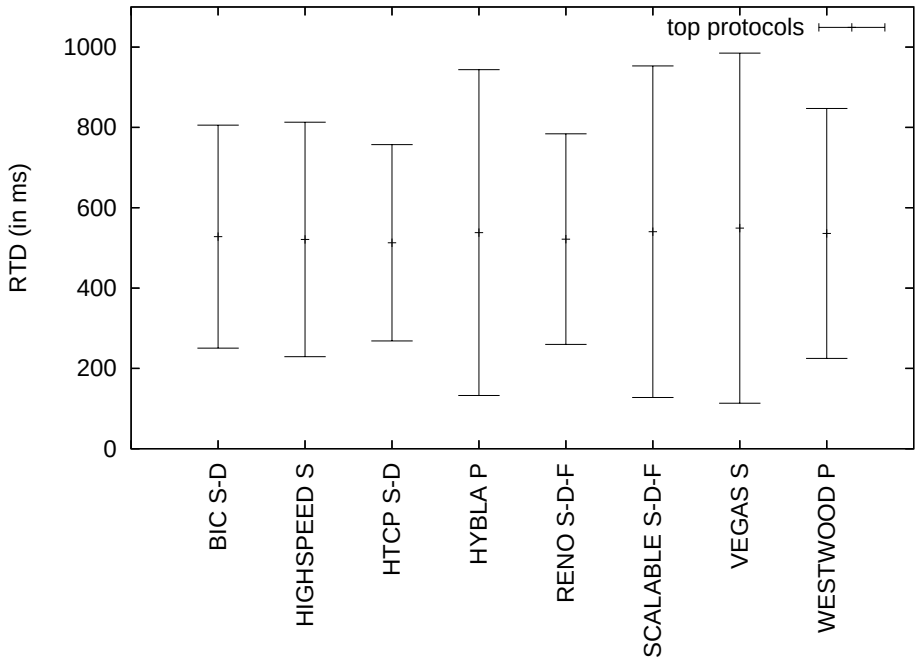

Figure 3.13: The top combination for each variation, Thin Streams, 100 ms delay, 10% loss

a bigger cwnd does not make any difference in these thin stream settings. Additionally, some of them act like New Reno in these specific settings. The differences that can be seen are due to arbitrary incidents, uncontrollable elements in the tests, e.g. the difference in the loss sequence.

The most relevant observation from these tests is the total absence of the Fast Retransmit mechanism, and the RTO as the means of detecting packet loss. Together with the unfortunate effects of expontential backoff, this means that there is a huge potential for improvements. Another important issue is the limited reception of ACKs, as only a few packets are sent at a time. As the extended loss detection mechanisms of the TCP protocols depend on the reception of ACKs, this is essential to keep in mind when considering possible enhancements.

As mentioned in the previous paragraph, a relevant issue present in these tests is the high maximum RTDs, with values of a considerable number of seconds, more than high enough to affect a user's perception of quality. Despite the fact that these values do not occur frequently, their mere presence has a strong influence on perceived quality, and as a consequence stringent latency is not achievable in these scenarios. As an example, a delay of say 20 s is totally unacceptable for users playing interactive games. Similarly, for users playing in an interactive, virtual reality, it is important

to maintain a (roughly) consistent view. High delays could cause some users to have an inconsistent view of the virtual world, which could have unfortunate consequences.

Some limitations of the tests performed exist, e.g., the way thin streams are simulated gives only 1 transmitted packet per cwnd, which more or less annihilate dupACKs. Thus, dupACK-related enhancements cannot be tested with exactly the same parameters. However, the primary goal of the tests were to test the thin stream performance, which has been done. The poor behaviour experienced for this kind of streams were still revealed, even though some tests could have been performed slightly different to ease potential comparisons between recorded performance and proposed enhancement tests, requiring certain demands to be fulfilled in order to work properly. Slightly different tests should be run to test the performance for such enhancements, but they should still be comparable with the earlier tests.

3.3 Summary

In this chapter, extensive tests have been presented to examine the performance, and especially the RTD, of the different TCP variations present in the Linux 2.6.15 kernel. The variations were tested for 2 types of streams.

The thick stream testing provided us with a thorough impression of how well the individual variations performed compared to each other. The RTD differences between the protocols were related to the performance. The SACK and FACK options provided considerable improvements, and should be used whenever available. As Hybla showed the best performance, and had the lowest RTD, it is the recommended variation when available. However, the Fast Retransmit mechanism keeps the RTD fairly low for this sort of stream, regardless of the variation used. Thus, the optimization of the RTD (i.e., minimization) is well covered for thick streams.

No significant differences between the protocols could be detected in the thin stream settings. However, this was expected since the protocols behaviour should be more or less the same in these settings. The important discovery here was the absence of dupACKs, and thus the RTO as the only means of detecting packet loss. As the preferred packet loss detection mechanism, Fast Retransmit, was not invoked in these settings, this resulted in a high average RTD, and additionally, in some extremely high maximum values (due to the RTO's exponential backoff).

As this chapter has revealed serious weaknesses concerning the packet loss detection mechanisms for thin streams, we introduce several enhancements to improve the performance in thin stream settings in the following chapter.

Chapter 4

Proposed enhancements

The results of the previous chapter showed that there was a great need for improvements in thin stream settings. Thus, based on these results, we consider several enhancements to TCP in a thin stream scenario. As we have Internet gaming in mind, the enhancements proposed are sender-side modifications only (see Section 1.2).

Since most of the enhancements proposed in this section should only be used in a thin stream setting, it is essential to provide a thorough definition of the term thin stream, and a justification of this. It is necessary to come up with a means for determining when a stream is thin. I.e. absolute requirements that must be fulfilled in order to classify the stream as thin. These requirements must be measurable in order to be tested.

The term thin stream is somewhat vague, but typical characteristics of thin streams involve limited sending of packets, often with a limited payload as well, and additionally intermediate pauses when there is no data to transfer. It is essential to identify the relevant characteristics of thin streams, since they limit the usual packet loss detection mechanisms. Thus, we must examine these and consider them in order to provide an optimized performance in this setting as well. Thus, the enhancements must be customized to these characteristics.

The thickness of the stream can be determined based on the number of packets that have been sent but not acknowledged, also called the packets that are **in flight**. This is the best measure to determine if the stream is thin, as a limited number of packets in flight subsequently limits the invocation frequency of the Fast Retransmit mechanism, which is the least penalizing mechanism when it comes to handling packet loss. Additionally, it is potentially a lot faster than the retransmission timeout alternative.

In order to trigger Fast Retransmit, 3 dupACKs are required by the sender. Thus, at least 4 packets need to be in flight at the time of detection, as 1 packet must be lost and 3 additional packets are required to each create a dupACK. Thus, a stream can definitely be classified as thin if it has

less than 4 packets in flight.

However, if 4 packets are in flight, Fast Retransmit will only be invoked if it is the first packet that is lost. Additionally, the following 3 packets must be received successfully, and none of the corresponding dupACKs must be lost on their way to the sender. Otherwise, the sender is forced to wait until at least one other packet is transmitted that can create another dupACK. If there is a time interval between the sending of packets, this delay could be considerable, maybe even long enough for the RTO to expire in the mean time. Time intervals between sending of packets are quite common in thin stream settings, and thus this raises a serious problem for the Fast Retransmit invocation. To take at least some measure against this problem, we use the following thin stream definition in these experiments:

A stream is thin if it has 4 or fewer packets in flight

However, this value is easily tunable if another value is found to be more reasonable.

Considerable time intervals between the sending of packets are not an absolute requirement to define a stream as thin, but it is a typical feature of this type of stream. Thus, it should be taken into account when considering possible improvements. The burstiness of the stream might vary throughout the connection, meaning that both time intervals between sending of data and the amount of data (and thus subsequently the amount of packets) transferred in one burst typically vary. This introduces another common feature in thin streams. The amount of data transferred at a given time is often small, typically far less than the MSS, and this is worth examining when considering possible improvements.

4.1 Removal of the exponential backoff mechanism

The simplest enhancement to improve thin stream performance would be the elimination of the Exponential Backoff mechanism. When a stream is thin, this will not have any major impact on either fairness or the network traffic. The streams will not gain anything as far as throughput is concerned as they have a limited sending rate anyway (by definition of being thin).

The high maximum values, regardless of their origin, will have a massive influence if we look at game perception, as a pause of 15 s will be quite detectable and probably highly annoying. Thus, it is quite relevant to remove these high values to obtain considerably more stringent deviations in the RTD. Turning off exponential backoff should provide this nicely. Thus, for thin streams the number of retransmissions will not have any influence on the RTO value.

Even though this will be quite efficient, it might not be the best solution, as an RTO resets the cwnd and slow-start is entered. Thus, if possible,

retransmissions should be triggered by some other means. Some proposals are presented in the following sections.

4.2 Implementation of "thin stream Fast Retransmit"

The ideal enhancement would be a Fast Retransmit mechanism that works in thin stream settings. To expect that it could be invoked as often as the regular Fast Retransmit for thick streams might however be unrealistic as this would require a steady, continuing generation of dupACKs. Due to the thinness of the streams, it is unlikely that a dupACK would be generated for many of the packet loss incidents. However, the number of dupACKs created during a connection could still be considerable. Incorporating some sort of mechanism that utilizes these dupACKs will still be a great benefit, as retransmissions triggered by dupACKs potentially can be much faster than the retransmission triggered by a retransmission timeout (and slow-start will be avoided, even though this might not always be as important for thin streams).

4.2.1 The Vegas modification

Vegas' proposed improved Fast Retransmit mechanism seems promising in order to improve the retransmission latency further. A timeout check for each dupACK would allow a retransmission to be triggered prior to the reception of 3 dupACKs, possibly on the reception of the first one. This would increase the chances of Fast Retransmit being invoked even in thin stream settings.

However, it will be time-consuming to include, as this mechanism might not just be added as an extension of the Vegas code, due to the implementation architecture of the TCP protocol in the Linux 2.6.15 kernel.

The Linux TCP protocol implementation is open for several different suggestions (the TCP variations) about how to handle congestion control. This is provided by allowing certain specified methods to be replaced by alternative ones. These alternative methods (for a given TCP variation) are registered with TCP and can then be invoked. The methods which are replaceable are related to congestion control issues, as it is this the TCP variations are meant to be addressing, but the basics, RTO and Fast Retransmit, are not replaceable. As a complete Vegas extension is additionally concerned with Fast Retransmit, this is not plugable with the current TCP implementation in the Linux kernel, it does not exist today.

Additionally, implementing this modification as part of the Vegas module could constrain it's performance and impression in our thin stream scenario, as it would suffer from the potential drawbacks of Vegas' congestion avoidance mechanism.

However, incorporating it as a general mechanism in the extensive TCP protocol is extremely time-consuming, and seems quite excessive as it might not be considered beneficial in all scenarios. For thicker streams one might have to include spurious retransmissions [25] in the consideration.

TCP attempts to avoid (or at least minimize) spurious retransmissions as they waste network resources, but the Vegas modification could lead to an increased number of these. Minimizing spurious retransmissions and the RTD seems to be goals opposed to one another. The improvement of one of them will typically cause a decline of the other. It depends on which of them is found to be most essential for a particular scenario, as it is difficult for them to coexist and still perform optimally individually.

Spurious retransmissions will have the biggest impact for thick streams as they transmit a high amount of packets. However, in this scenario the original Fast Retransmit should be invoked fairly early, and it might not be devastating to wait for this invocation instead. Thus, a Fast Retransmit modification is most essential for thinner streams (we know this already..), and the number of spurious retransmissions will then be lower in this case. A couple of unnecessary retransmissions might not have that much of an impact on network traffic. However, implementing the Vegas modification for thin stream use only does not seem to be worth the trouble. It will be extremely challenging and, depending on the timeout value, the stream might still need to wait for 3 dupACKs anyway. Thus, the Vegas modification is not implemented for this thesis, and another simpler scheme is proposed instead in the next section.

4.2.2 Modified Fast Retransmit for thin stream use only

The Fast Retransmit mechanism is triggered after n dupACKs are received, and n is usually 3 (frexmt_thresh, see Section 2.1). A scheme to adjust this mechanism for thin streams settings could be to reduce n. As the number of dupACKs in thin stream settings is limited, it makes sense to utilize them whenever one occurs, n is therefore reduced to one. However, as this should only be done for thin streams (otherwise a lot of spurious retransmissions would be triggered), the implementation should be flexible, and able to adapt to changes in the streams' thickness. Thus, it is not enough to just change the frexmt_thresh value, as this will make the stream unable to adapt. The approach taken here is to adjust the frexmt_thresh dynamically, where the number of dupACKs needed in order to trigger a retransmission will depend on the current thickness of the stream. If the stream currently is considered thicker than the thin stream definition, the traditional approach is taken, waiting for the 3 musketeers to save the day (waiting for the usual 3 dupACKs), otherwise Fast Retransmit is triggered on the reception of the first dupACK.

This idea could be expanded to include a Fast Retransmit invocation

for the reception of 2 dupACKs, which would then be triggered for another specific stream thickness. It would then be necessary to define a "chubby/slim stream", ranging from the thin stream limit to an appropriate threshold when the stream is considered thick. Then, the transition to the original Fast Retransmit mechanism should be carried out. Implementing this would provide a dynamic adaption of the Fast Retransmit strategy, which would be able to adjust it's invocation requirements depending on the streams nature. Thus, as the stream thickness increases, so does the conditions for the Fast Retransmit mechanism invocation. This is however not implemented in this thesis.

It is important to keep in mind that even though Fast Retransmit can be a lot faster than an RTO (potentially just above the RTT), provided that the dupACK generation is fast, this is not necessary so. Additionally, it is essential to remember that dupACKs might be lost as well, delaying the Fast Retransmit invocation. This is possibly devastating for the Fast Retransmit invocation if there is a considerable time interval until the next transmission.

4.3 Fill every packet sent (or fill each retransmitted packet)

As mentioned earlier, the invocation of the Fast Retransmit mechanism depends entirely on the generation of dupACKs, and thus their limited presence in thin streams constrains it's influence. However, as the RTO and the Fast Retransmit mechanisms are the incorporated means to discover packet loss, there is little more that can be done in order to reduce the RTD further. Thus, as the delay of the retransmission has been reduced, the attention turns towards reducing the number of retransmissions. And additionally, to reduce the possibility of retransmissions of successive segments being needed when multiple packet loss occur. That is, reduce the possibility of another retransmission (of a succeeding segment) following the retransmission of this segment being needed, as this retransmission will extend the time the receiver will have to wait before being able to deliver the data to the application.

Thin streams are often not only limited in the number of packets transmitted, but also in the amount of data transmitted in each packet. The amount of data is often considerably smaller than the MSS. Thus, there is more room available in the packet that could be exploited.

4.3.1 Piggybacking in original packets (Fill every packet sent)

A promising scheme could be to fill every packet sent on the network to the limit. That is, if the packet to be sent only contains a limited amount of

```
/* Check segment sequence number for validity.
 *
 * Segment controls are considered valid, if the segment
 * fits to the window after truncation to the window.
 * Acceptability of data (and SYN, FIN, of course) is
 * checked separately.
 * See tcp_data_queue(), for example.
 *
 * Also, controls (RST is main one) are accepted using RCV.WUP
 * instead of RCV.NXT. Peer still did not advance his SND.UNA
 * when we delayed ACK, so that hisSND.UNA<=ourRCV.WUP.
 * (borrowed from freebsd)
 */

static inline int tcp_sequence(struct tcp_sock *tp, u32 seq,
                               u32 end_seq)
{
     return  !before(end_seq, tp->rcv_wup) &&
             !after(seq, tp->rcv_nxt + tcp_receive_window(tp));
}
```

Figure 4.1: Sequence number validity check

data (less than MSS), the rest of the packet could be filled with previously sent, but still unacknowledged data. This does not result in any increased routing costs on the network, as there is no increase in the number of packets, only in the number of bytes. It does, however, improve the possibility of a successful reception of the data without the invocation of the retransmission mechanisms.

However, this would mean sending a lot of packets with the same sequence number (starting byte), but different length. This requires the receiver to check for both sequence number and length when it is processing the packets. If it just checks the sequence number, only the first packet will be successfully received, the others (with the same sequence number) will be discarded as duplicates (as the receiver now expects a later sequence number). The TCP implementation in the Linux 2.6.15 kernel checks both sequence number and length (Figure 4.1), and thus this should be possible to implement. However, due to the limitation of the disposable time for this thesis, this modification is not implemented here. It is however strongly recommended that this is done, as this should have a significant effect on the thin stream performance.

4.3.2 Piggybacking in retransmitted packets (Fill every retransmitted packet)

A scheme that should work is to fill each retransmitted packet to the limit. That is, when a retransmission occurs, if there is more room in the packet, this is filled with data sent after the segment that is now being retransmitted. This reduces the possibility of another retransmission succeeding this one, if multiple packet losses occurred.

This scheme could be expanded to include retransmission of all additional unacknowledged data that did not fit in this packet as well, at least as many as the cwnd allows. As this concept is only to be used in thin stream settings, the number of additional packets will not be many, and thus, it should not have any major impact on the network traffic. It will mean at most a retransmission of $('the\ definition\ of\ thin\ streams\ limit' - 1)$ additional packets (the triggered retransmission is of course not included, hence minus 1). This will reduce the chances of several retransmissions of successive segments being required when multiple packet losses occur.

However, since thin streams typically do not carry a lot of data in the transmitted packets, the first modification might work quite well, and will in most of the cases cover all the unacknowledged packets, thus questioning the necessity of the second expansion. Thus, it is not implemented in this thesis.

Since thin streams typically transmit data in intervals, there might be a lot of occurrences when there is no unacknowledged data to add to the packets, and so this mechanism might not be invoked often. Additionally, there is a substantial chance that the original of the added segment successfully arrived at the receiver, and in that case the addition of this segment to another retransmission does not have any effect. Thus, this might not have a dramatic impact on the logistics of the stream, but when it does, it provides a positive effect, and should improve game performance, or at least avoid a quality reduction detection from the user.

4.4 Summary

The previous chapter revealed considerable room for improvements in thin stream settings, and several enhancements were thus proposed in this chapter. They are only used in thin stream settings.

The removal of the Exponential Backoff mechanism should eliminate the undesirable high maximum values, which could be unfortunate for the quality perceived by users. However, a retransmission timeout is an expensive packet loss detection mechanism, as slow-start is entered. Thus, 2 modified Fast Retransmit mechanisms were presented. The Vegas mechanism was found to be too time-consuming to implement, compared to what

it had to offer. A simpler modification was proposed instead, the reduction of the frexmt_thresh to 1 for thin streams, and thus retransmitting for the first dupACK received, as their existence was found to be limited in thin stream settings.

Additionally, mechanisms to reduce the need for retransmissions of succeeding segments were proposed. The first idea, filling every transmitted packet, was not implemented in this thesis, due to the limited time at hand. However, a mechanism filling every retransmitted packet (piggybacking) was implemented.

As several enhancements have been proposed, we move on to implement and test these, to see if they indeed improve the performance.

Chapter 5

Implementation and testing of the proposed enhancements

The TCP protocol files are mainly located in the **/usr/src/linux/net/ipv4/** directory. Additional **tcp.h**-files are located in **/usr/src/linux/include/net/** and **/usr/src/linux/include/linux/**, respectively.

The implementation of the TCP protocol is extensive and a lot of files are involved. However, we are only concerned with the parts involved in retransmission scenarios, thus we only need to look into the files involved in this particular process.

The **tcp_input.c** file is as the name implies involved in the processing of input from the network. Whenever data is received from the network, some part of this file eventually gets involved in the process of deciding what this packet contains and what should be done with it. The modified methods in this file are *tcp_time_to_recover()* and *tcp_fastretrans_alert()*. These methods are not related to alternate congestion control schemes, and thus cannot be overridden by the different TCP variations.

The **tcp_output.c** file is similarly concerned with output related issues. Whenever some data needs to be sent, some part of this file is involved in the sending. Methods which have been modified here are *tcp_retrans_try_collapse()* and *tcp_retransmit_skb()*.

The **tcp_timer.c** file is as the name implies involved in timer related issues. Only one method is modified here, the *tcp_retransmit_timer()* method.

As the implemented enhancements are mainly concerned with retransmissions, it is worth to describe briefly how TCP keeps track of it's outgoing packets in Linux, as we are going to retransmit some of these.

Since TCP needs to handle unreliable data delivery, it is necessary to hold onto the sent data frames until acknowledgments for this data are received [4]. This is necessary in case that an RTO occurs, or some other indication arrives claiming that some data packet was lost, which requires a retransmission.

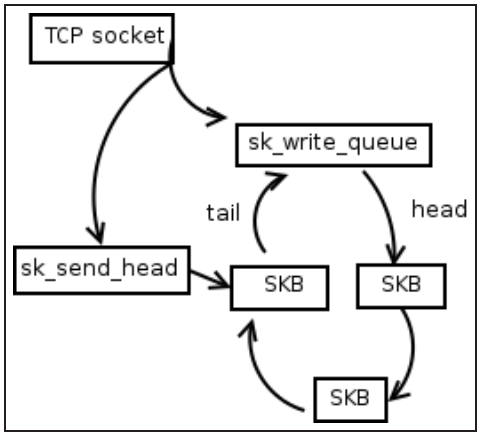

Figure 5.1: TCP output queue

Together with a couple of other complicating matters, this results in the TCP output engine being non-trivial. Figure 5.1 shows the TCP output queue in Linux. The TCP socket maintains a doubly linked list of all pending output data packets, kept in socket buffers (skb). *sk_write_queue* points to the head of the list. *sk_send_head* keeps track of where in the queue the next packet to be sent is located. If *sk_send_head* is NULL it means that all packets on the
sk_write_queue have been sent once already. When ACK packets arrive from the receiver and more send window space becomes available, the *sk_write_queue* is traversed from *sk_send_head* and as many frames as allowed are sent.

Linux calculates the RTO differently from [13]. The complicated calculations are done in the computations of the **Smoothed Round-Trip Time** (SRTT) and **Round-Trip Time Variation** (RTTVAR) variables, and the RTO is then set as $(1/8 * SRTT) + RTTVAR$. RTTVAR is set to minimum TCP_RTO_MIN, which is 200 ms (Hz/5). The RTO never gets below $TCP\_RTO\_MIN + RTT$.

5.1 Removal of the exponential backoff mechanism

The Exponential Backoff mechanism is located at the bottom of the *tcp_retransmit_timer* method in the **tcp_timer.c** file, at a label marked *out_reset_timer*. The RTO (*icsk->icsk_rto*) is doubled with the code line:

```
icsk->icsk_rto = min(icsk->icsk_rto << 1, TCP_RTO_MAX);
```

If the new value exceeds the maximum RTO value *TCP_RTO_MAX*, the RTO is set to *TCP_RTO_MAX* instead, but this value is 120 s in the Linux 2.6.15 kernel, and should thus seldom be used.

```
1 if(tcp_packets_in_flight(tp) > 4)
    icsk->icsk_rto = min(icsk->icsk_rto << 1, TCP_RTO_MAX);
```

Figure 5.2: Thin stream No Exponential Backoff Modification

```
1 /* Not-A-Trick#2 : Classic rule... */
  if(tcp_fackets_out(tp) > tp->reordering)
     return 1;
```

Figure 5.3: Number of dupACKs test

The RTO timer is then reset with the line

```
inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS, icsk->icsk_rto,
                          TCP_RTO_MAX);
```

passing the new RTO value as a parameter to the method.

As the exponential backoff is done in 1 codeline, the removal of this mechanism is quite easy, and is just implemented as an if-test (shown in Figure 5.2).

Thus, the value is only doubled if the stream currently has more than 4 packets in flight. Otherwise, it is defined as thin (as specified in the thin stream definition earlier), and the RTO value remains the same (but is still reset, of course). Using this modification should help us avoid the high maximum RTD values, experienced in the thin stream tests (run in the original kernel).

However, there is one exception. The RTO value is initially set to 3 s, as mentioned earlier (see Section 3.2.2). If packet loss occurs before any RTO estimates can be made (requires RTT samples), the first RTD will be approximately 3 s. The connections unlucky enough to experience this will be off to a bad start.

5.2 Modified Fast Retransmit for thin stream use only

The check for the number of dupACKs is located in the *tcp_time_to_recover()* method in the **tcp_input.c** file. This method returns 1 if it finds the time ready to recover for a number of different reasons (implemented as if-tests). The test that checks for dupACKs is shown in Figure 5.3.

The Linux 2.6.15 kernel incorporates the Eifel response [38] and detection [15] algorithms to detect and avoid spurious retransmissions due to the reordering of packets by the network. The *tp->reordering* variable is used in this process. It holds the value of the number of packets allowed to be assumed reordered by the network before the packet is considered lost. Thus, the number of dupACKs the sender can receive before it must

```
if(tcp_fackets_out(tp) > 1 && tcp_packets_in_flight(tp) <= 4)
          return 1;
```

Figure 5.4: Thin stream 1 dupACK test

```
/* C. Process data loss notification, provided it is valid. */
if ((flag&FLAG_DATA_LOST) &&
    before(tp->snd_una, tp->high_seq) &&
    icsk->icsk_ca_state != TCP_CA_Open &&
    tp->fackets_out > tp->reordering) {
      tcp_mark_head_lost(sk, tp, tp->fackets_out-tp->reordering,
                         tp->high_seq);
      NET_INC_STATS_BH(LINUX_MIB_TCPLOSS);
}
```

Figure 5.5: Additional dupACK test in *tcp_fastretrans_alert()*

consider the packet to be lost, and initiate a retransmission. It is by default set to frexmt_thresh (thus 3), but can be adjusted when spurious retransmissions are detected. The *tcp_fackets_out()* method returns the number of dupACKs plus 1.

To incorporate the 1 dupACK invocation of the Fast Retransmit mechanism for thin streams, the additional test shown in Figure 5.4 is added at the bottom of the method.

Thus, it is time to recover if at least 1 dupACK has been received for this segment, provided that the number of packets in flight is at most 4.

Additionally, there is a test in the *tcp_fastretrans_alert()* method which checks the number of dupACKs, shown in Figure 5.5.

This test is seldom used, but it is still extended with the same thin stream test as the one in the *tcp_time_to_recover()* method, resulting in the following modification shown in Figure 5.6.

```
if ((flag&FLAG_DATA_LOST) &&
    before(tp->snd_una, tp->high_seq) &&
    icsk->icsk_ca_state != TCP_CA_Open &&
    (tp->fackets_out > tp->reordering ||
     (tp->fackets_out > 1 && tcp_packets_in_flight(tp) <= 4))) {
      tcp_mark_head_lost(sk, tp, tp->fackets_out-tp->reordering,
                         tp->high_seq);
      NET_INC_STATS_BH(LINUX_MIB_TCPLOSS);
}
```

Figure 5.6: Modified additional dupACK test in *tcp_fastretrans_alert()*

```
if(!(TCP_SKB_CB(skb)->flags & TCPCB_FLAG_SYN) &&
    (skb->len < (cur_mss >> 1)) &&
    (skb->next != sk->sk_send_head) &&
    (skb->next != (struct sk_buff *)&sk->sk_write_queue) &&
    (skb_shinfo(skb)->nr_frags == 0 &&
     skb_shinfo(skb->next)->nr_frags == 0) &&
    (tcp_skb_pcount(skb) == 1
     && tcp_skb_pcount(skb->next) == 1) &&
    (sysctl_tcp_retrans_collapse != 0))
         tcp_retrans_try_collapse(sk, skb, cur_mss);
```

Figure 5.7: Check if it is worthwhile and possible to collapse

5.3 Piggybacking in retransmitted packets (Fill every retransmitted packet)

The filling of retransmitted packets requires a more extensive implementation modification than the two previous enhancements. Additional unacknowledged data needs to be added to retransmitted packets, thus some tampering with the data to be sent is required. It is possible that this can be done in a number of different ways, depending on how one chooses to implement the appending of additional data, and where in the code this is done.

When going through the TCP code, I became aware of a method in the **tcp_output.c** file named *tcp_retrans_try_collapse()*. This method attempts to collapse two adjacent skbs during retransmission. That is, it tries to fit the data of the next skb into the current one, and then frees the next skb. Actually it is implemented as a proc variable, **tcp_retrans_collapse**, making it possible to turn it on and off. It is called from the *tcp_retransmit_skb()* method in the same file, provided it is worthwhile and possible. This is checked through an extensive if-test, which among other things checks if the proc variable is set (shown in Figure 5.7).

First it checks that the skb does not contain a SYN packet, next it requires that the data length is less than half of the current MSS. Then it is checked whether the next skb is the first unsent packet (*sk->sk_send_head*) or whether it is the head of the queue (*sk->sk_write_queue*). Additionally, it is checked whether any of the skbs contain fragments or more than 1 packet. Finally, it checks that tcp_retrans_collapse is enabled.

The *tcp_retransmit_skb()* method is called to retransmit 1 skb, which typically contains 1 packet. However, if *tcp_retrans_try_collapse()* is invoked, the skb can now contain 2 original packets. It is however not guaranteed that the collapse has taken place, even if the if-test is passed and *tcp_retrans_try_collapse()* invoked. The method performs some additional tests to make sure that it is worth collapsing. E.g. there must be enough

room left in the current skb to contain the next skb's data, and there is no need to collapse if the next skb is SACKed. However, provided that the collapse is carried out, the retransmission will consist of 2 packets, thus the packet is at least partially filled.

The approach taken here is to extend this mechanism so it will be able to collapse several packets, provided this is worthwhile and there is room in the packet. That is, just continue collapsing the next skb with the current one as long as it is possible and worth it, instead of just doing it once.

The *tcp_retrans_try_collapse()* method is doing the work of collapsing two skbs for us, we just have to decide how long it is worth collapsing. Thus, what we want to do is to perform the if-test with the subsequent call to *tcp_retrans_try_collapse()* as long as it is possible and profitable.

Performing the same task several times until some condition is not met usually means involving a while-loop. I have however earlier experienced that an implemented while-loop in the kernel resulted in the machine freezing. Even though this might not be the result here, I have chosen not to use a while-loop to avoid similar problems. However, this can easily be modified if this is found to be beneficial.

As this modification should only be performed in thin stream settings, the previously used thin stream test, checking the number of packets in flight, is applied here as well. Thus, if the stream is considered to be thick the usual approach is taken (Figure 5.7). Otherwise the approach shown in Figure 5.8 is taken.

The thing we wish to do is barely different from the usual approach. We still want to collapse 2 packets, the only difference is we want to do this several times and slacken the conditions for collapsing. Thus, we drop the requirements demanding that the data currently residing in the skb must be less than half of the current MSS, and that only 1 packet can be held in each skb. Additionally, the modification is not implemented as a proc variable, thus the condition for the proc variable being set is dropped.

The *sk_stream_for_retrans_queue_from()* for-definition is used for traversing the queue from a given skb, passed as a parameter to the definition. However, what we want to do is to remain at this element, and collapse the succeeding element with this one as long as this is possible and there are more elements in the queue. Thus, a little trick is used to provide this. We remember the first skb (*first*). When we have gone through one pass in the loop, the current skb will be the second element in the queue from where we started. Thus, the previous element is the one we actually want, and we therefore reset the current skb to this element.

Additionally, it is necessary to check that we have actually collapsed something (otherwise we will be stuck in an infinite loop). If we have passed the test and *tcp_retrans_try_collapse()* merged the next skb with the current one, the next skb (*skb->next*) should be different. Thus, when the test has been executed (possibly *tcp_retrans_try_collapse()* as well), we check

```
else{
  first = skb;
  sk_stream_for_retrans_queue_from(skb, sk) {
     if(skb->prev == first){
        skb = first;
     }
     changed = skb->next;
     if(!(TCP_SKB_CB(skb)->flags & TCPCB_FLAG_SYN) &&
        (skb->next != sk->sk_send_head) &&
        (skb->next != (struct sk_buff *)&sk->sk_write_queue) &&
        (skb_shinfo(skb)->nr_frags == 0
         && skb_shinfo(skb->next)->nr_frags == 0)){
           tcp_retrans_try_collapse(sk, skb, cur_mss);
     }
     if(changed == skb->next || skb->next == sk->sk_send_head ||
        skb->next == (struct sk_buff *)&sk->sk_write_queue){
        break;
     }
  }
  skb = first;
}
```

Figure 5.8: Thin stream collapse modification

```
if(tcp_packets_in_flight(tp) > 4){
   BUG_ON(tcp_skb_pcount(skb) != 1 ||
          tcp_skb_pcount(next_skb) != 1);
}
```

Figure 5.9: Conditional 1 packet bug test

if this is true. If it is the same, we break the for-loop as nothing more can be collapsed. Additionally, we check if the next skb is the first unsent packet or the head of the queue, as either of these mean that we are finished as well.

Since the modification is only used in a thin stream setting, the queue can at most contain 4 elements as each element represents at least 1 packet in flight. Additionally, the *tcp_retrans_try_collapse()* method is slightly changed as it contains a bug test checking if both skbs contains 1 packet. The thin stream test is just added here as well (shown in Figure 5.9).

5.4 Testing and evaluation of the implemented enhancements

As the modifications and their implementation have been described, we move on to see how they perform in practice, and if they provide the de-

sired enhancements when it comes to the RTD. However, as the earlier tests revealed a more or less identical behaviour for the TCP variations in thin stream settings, there is no need to test the modifications with all of these, as the modifications are made for thin stream use only. Thus, only some variations and SACK options are used in these tests. As New Reno is the default protocol it is naturally included in the tests. Based on the thick stream tests, Hybla was picked from the alternative variations, as it gave the best impression. Additionally, these 2 were tested with the SACK and FACK option, as these improved the RTD in the former tests.

5.4.1 Test layout

The tests run earlier were trying to simulate real traffic, and real stream traffic behaviour, but they might not have been able to capture all aspects of the nature of real traffic. However, it would be nice to see how the implemented modifications perform for real traffic as well, and especially game traffic. My supervisors were kind enough to provide an hour of real traffic from an Anarchy Online game server. Using one of their programs, the traffic flow is regenerated, providing about an hour of real game traffic.

To test the modifications with the Anarchy Online traffic, the original kernel must be tested with this traffic. Additionally, as we need dupACKs to test the Fast Retransmit modification, the number of packets in flight must be heightened. Thus, some tests with the original kernel must be run for comparison, resulting in some additional testing with the original kernel.

Thus, tests are run for 2 different kernels, the original and the modified. However, as the earlier tests showed (and was later discussed), the number of dupACKs are limited in thin streams, and even when we are utilizing them, their impact will be constrained. The Piggybacking modification might have a limited impact as well, as it requires additional outstanding data and enough room in the retransmitted packet. Additionally, if the piggybacked segment is not lost, this does not have any effect. Thus, the most influential enhancement should be the removal of the Exponential Backoff mechanism. To get a certain view of the effect of this modification compared to the others, the tests are run in a kernel with only this modification implemented as well.

As the Piggybacking modification is based on a mechanism already incorporated in the kernel as a proc variable, it would be interesting to see the difference in performance with and without this proc variable turned on. And additionally, to see the difference between the original and the modified mechanism. Thus, when testing the Anarchy Online traffic, the traffic is run in the original kernel both with and without the proc variable set. This is not done for the other test types as the piggybacking mechanism is not invoked there, due to the nature of these tests (explained under

the specific test type).

The Anarchy Online game server has a high number of connections during the provided period, 175 totally, varying in duration and in the amount of transmitted data. For the individual flows the transmission of data is not performed in specific time intervals, it is quite dynamic both in the interval between the sending of packets and in the number of packets transmitted within a given interval. Thus, the number of packets in flight might vary considerably for these flows, possibly even occasionally exceed the thin stream limit, temporarily disabling the thin stream modifications (if we are running a modified kernel). Thus, exponential backoff can potentially occur, if multiple retransmissions are needed during this period.

It is not worth evaluating the individual connections as they vary in the amount of data they are transmitting, and thus in the number of retransmitted packets. Several of the connections will typically only retransmit a few packets (possibly even none), which is not nearly enough to be able to draw any conclusions. Due to this nature of the traffic flow, the connections will be evaluated as one. That is, we summarize the connections, and look at the statistics for the entire hour. Thus, the RTD and packet statistics will be the average for all the connections. For the average and standard deviation calculations, the values are weighted corresponding to the number of retransmitted packets, meaning that the average for a connection with a couple of hundred retransmissions will have a bigger influence than a connection which only retransmits a few packets. Thus, even if the "abnormalities" experienced in the earlier tests occur occasionally, this should not have any considerable impact on the result, as they should only be a drop in the sea.

Additionally, some extra statistics regarding dupACKs are included, as they are thought to be relevant for the evaluation. It would be interesting to see how many dupACKs are created in the Anarchy Online scenario, thus we additionally include statistics for the average number of dupACKS per connection, and the number of connections without dupACKs. It is important to keep in mind that the number of dupACKs does not reflect the exact number of possible Fast Retransmit invocations, as some dupACKs are ignored. An example, if a Fast Retransmit has just been triggered and a retransmission is under way, incoming dupACKs still indicating the loss of the retransmitted packet are ignored.

Additionally, we want to test the improvement of the Fast Retransmit modification, thus we need to create a thin stream environment which still contains a certain generation of dupACKs. This is done by using netperf with the same parameters as before, but changing the message size to 1448 B, which is the SMSS used in this network. Thus, as 4 messages are still sent in one burst, but the message size is changed to SMSS, this results in the transmission of 4 packets, which should provide a certain opportunity for receiving dupACKs (but by the thin stream definition, the stream is still

considered to be thin).

For the original kernel, the 2 tests just described are run. For the modified kernel and the kernel with the No Exponential Backoff mechanism only, the original tests are run in addition. As no differences could be seen in the thin stream settings, the tests are just run for one scenario, with a delay of 100 ms (gives RTT 200 ms) and a 5% loss rate.

Tracepump

This program is provided by the supervisors and regenerates the Anarchy Online traffic, making it possible to test this traffic with alternate RTT and loss scenarios, and see how this affect the traffic. This can then be monitored by using tcpdump, and subsequently tcptrace can be used to extract the relevant statistics. The program needs to be started at both sender and receiver.

The program is started at the receiver with the command:

```
./tracepump --recv
```

The only thing the receiver does, is to listen for TCP connections on a number of ports. If something connects, it reads data from the established connection and discards it.

The usage for the sender side is:

```
./tracepump --send [Options] <recvaddr> <file>
```

The sender reads a pcap file **<file>** and tries to recreate the packet streams as it is encountered in the file in real-time. To do that, it connects to a given IP address **<recvaddr>**, starting at a given port (default is 12000). For every further connection encountered in the trace file, it connects to the same IP address but increases the port number by one. Connection establishment for all ports is performed before the sending starts. Only TCP connections from the trace are considered, and only first time sending of packets is recreated, retransmissions are ignored.

Tcpdump (Section 3.1.2) is then used to listen and gather information about the traffic.

5.4.2 Test of the selected TCP variations with the earlier test configuration

The original kernel

Table 5.1 shows the earlier test results for the TCP variations and SACK options used in the tests performed here. These have been discussed earlier, so we will not go into details here. Shortly summarized, no differences

TCP Variation	Packet Statistics		DupACK Statistics	Retransmission Time Statistics (in ms)			
	Total	Rexmt	dupACKs	Min	Max	Avg	Stddev
RENO P	3929	414	1	412.1	1680.1	467.5	162.6
RENO S	3914	392	0	412.1	1680.1	466.4	178.4
RENO S F	3918	402	1	412.1	3328.2	468.0	217.2
HYBLA P	3912	403	0	412.1	1680.1	476.3	185.7
HYBLA S	3930	396	0	412.1	1680.1	454.6	141.6
HYBLA S F	3882	361	0	412.1	1680.1	469.4	184.4
Combined RTD Statistics				412.1	1954.8	467.0	178.3

Table 5.1: Netperf Statistics, Message Size 100 B, Original Kernel

TCP Variation	Packet Statistics		DupACK Statistics	Retransmission Time Statistics (in ms)			
	Total	Rexmt	dupACKs	Min	Max	Avg	Stddev
RENO P	3925	402	1	412.1	451.5	417.0	3.4
RENO S	3925	391	0	412.1	597.5	417.4	9.6
RENO S F	3923	381	0	412.1	464.0	417.3	3.9
HYBLA P	3921	380	0	412.1	444.0	416.9	3.3
HYBLA S	3896	352	0	412.1	930.9	418.9	27.5
HYBLA S F	3958	416	0	412.1	420.0	417.1	2.9
Combined RTD Statistics				412.1	551.3	417.4	8.4

Table 5.2: Netperf Statistics, Message Size 100 B, Modified Kernel

between the protocols, practically no dupACKs, thus RTO is the only detector of packet loss, and it's Exponential Backoff mechanism causes high maximum values for the RTD.

The modified kernel

As we know from the earlier tests, running the original test configuration provides practically no dupACKs, and this can be seen in the test results (Table 5.2). Additionally, as the second packet in a burst is not sent until the first has been acknowledged (see Section 3.2.2), there will usually not be anything for the Piggybacking modification to collapse. Thus, the No Exponential Backoff mechanism should more or less be the only modification invoked in these tests. However, this mechanism performs smoothly, resulting in an extreme declination of the maximum RTDs. The average of the maximum RTDs has dropped from 1954.8 ms to 551.3 ms, and 4 of the runs have maximum RTDs below or within a close proximity of the averages for the original kernel.

The averages for the modified kernel remain close to the minimum values (which is the best we can get with only No Expontial Backoff), and the standard deviation is very low as well, below 5 ms for 4 of the 6 runs. The

TCP Variation	Packet Statistics		DupACK Statistics	Retransmission Time Statistics (in ms)			
	Total	Rexmt	dupACKs	Min	Max	Avg	Stddev
RENO P	3914	361	0	412.1	420.0	417.0	3.0
RENO S	3907	355	0	412.1	664.0	418.0	15.1
RENO S F	3922	378	0	412.1	616.6	417.8	10.7
HYBLA P	3896	337	0	412.1	528.2	417.1	6.9
HYBLA S	3916	406	4	412.1	7720.3	459.9	453.8
HYBLA S F	3948	404	0	412.1	420.3	416.9	3.0
Combined RTD Statistics				412.1	1728.2	424.5	82.1
(excluding the Hybla S run)				412.1	529.8	417.4	7.7

Table 5.3: Netperf Statistics, Message Size 100 B, Kernel with only No Exponential Backoff

averages show a decrease of 40-60 ms from the runs with the original kernel. Thus, considering the stream nature in these tests (with no dupACKs), the results are satisfactory, as we keep the average close to the minimum.

Kernel with only the No Exponential Backoff modification implemented

The test results (Table 5.3) for this kernel are practically identical with the results from the previous section. The only exception is the Hybla run with the SACK options, which is unfortunate enough to get an abnormally high maximum value, but this is by pure chance and has nothing to do with the Hybla or SACK protocol. These results are however quite expected, as the No Exponential Backoff mechanism was the only modification invoked in the former tests (explained in the previous section). Thus, whether the additional enhancements are implemented or not does not make any difference (for this test configuration), and the test runs in both kernels will experience the same protocol behaviour. Removing the Hybla S run from the RTD statistics results in almost identical values for the 2 modified kernels.

5.4.3 Anarchy Online testing

The original kernel

The test results for the Anarchy Online traffic (Table 5.4) are quite interesting here, as a more dynamic behaviour might result in all of the modifications contributing to an improved performance. The differences between the TCP variations are limited, but as the thin stream nature is quite different for this traffic compared with the former tests (Section 5.4.2), a short comparison between the 2 will be presented.

TCP Variation	Packet Statistics		DupACK Statistics		Retransmission Time Statistics (in ms)			
	Total	Rexmt	dupACKs	No dupACKs	Min	Max	Avg	Stddev
RENO P	2209.7	207.1	25.5	37	413.7	1954.0	470.1	216.5
RENO S	2232.6	199.9	55.2	9	396.4	1622.4	463.1	175.2
RENO S F	2221.3	201.5	54.9	18	401.5	1617.6	465.0	177.6
HYBLA P	2266.0	201.7	67.6	7	392.0	1585.5	461.5	176.1
HYBLA S	2232.4	196.2	74.5	7	396.9	1695.6	462.9	175.0
HYBLA S F	2257.4	197.7	75.1	8	412.3	1818.7	463.6	173.7
Combined RTD Statistics					402.1	1715.6	464.4	182.4

Table 5.4: Anarchy Online Statistics, Original Kernel

The average RTDs roughly correspond to the values gotten in the earlier tests. There are however several other interesting differences between the tests. The most welcome one might be the higher number of dupACKs present in the Anarchy tests. Especially Hybla and the enabling of the SACK options provide a considerably increased number, making things more promising for the Fast Retransmit modification. However, their presence are still quite limited, as several connections contain no dupACKs, and thus their effect might be limited as well.

As the packet transmission frequency is different and more dynamic, the minimum and maximum values are a little different. The maximum RTDs are still high, while the minimum RTDs reflect the fact that at least some retransmissions are not triggered by the RTO, but as the minimum values still remain close to the minimum RTO value (which should be approximately 400 ms, 200 $ms + RTT$), this indicates that these retransmissions are rare.

The original kernel without tcp_retrans_collapse enabled

The interesting part for these results (Table 5.5) are the slight increase in the number of retransmitted packets, even though the number of transmitted packets has decreased, indicating that the collapse mechanism indeed has a certain effect. However, the differences are not great. The combined number of packets transmitted has decreased from 2236.6 to 2222.3, while the combined number of retransmissions has increased from 200.7 to 206.2. This gives a retransmission rate of 0.0897 for the original kernel with tcp_retrans_collapse enabled, while the rate without it is 0.0928. The combined average RTD is slightly affected as well, increasing from 464.4 to 467.5 when the proc variable is disabled.

Some interesting side effects is the considerable decrease in the minimum RTD values, as well as the increase in the number of connections without dupACKs for the Reno protocol, resulting in a lower number of

TCP Variation	Packet Statistics		DupACK Statistics		Retransmission Time Statistics (in ms)			
	Total	Rexmt	dupACKs	No dupACKs	Min	Max	Avg	Stddev
RENO P	2201.0	209.1	25.5	27	372.6	1637.0	467.0	197.0
RENO S	2231.2	211.8	31.2	31	361.5	1792.5	468.9	194.7
RENO S F	2193.7	207.7	30.6	29	393.5	1895.5	471.1	203.5
HYBLA P	2226.5	202.2	65.1	16	334.9	1645.6	468.1	200.9
HYBLA S	2246.8	203.6	72.3	9	331.7	1625.6	463.1	177.9
HYBLA S F	2234.5	202.8	72.5	7	350.9	1938.9	466.5	188.5
Combined RTD Statistics					357.5	1755.9	467.5	193.8

Table 5.5: Anarchy Online Statistics, Original Kernel without tcp_retrans_collapse

TCP Variation	Packet Statistics		DupACK Statistics		Retransmission Time Statistics (in ms)			
	Total	Rexmt	dupACKs	No dupACKs	Min	Max	Avg	Stddev
RENO P	2232.4	210.7	24.5	27	365.8	739.2	418.4	25.9
RENO S	2235.0	202.2	51.8	16	324.2	794.1	419.3	36.4
RENO S F	2236.4	203.3	52.1	10	342.5	703.2	418.4	36.9
HYBLA P	2256.4	202.6	64.3	8	318.2	914.5	419.8	44.6
HYBLA S	2247.1	199.1	71.2	9	316.4	624.3	417.8	32.6
HYBLA S F	2244.0	196.6	70.4	8	324.3	860.8	421.5	44.8
Combined RTD Statistics					331.9	772.7	419.2	36.9

Table 5.6: Anarchy Online Statistics, Modified Kernel

dupACKs per connection.

The modified kernel

The results (Table 5.6) show considerable improvements in the RTD statistics for the modified kernel. The comparison will here be against the original kernel with tcp_retrans_collapse enabled, as this produced the best results. The RTD averages have a decrease of 40-50 ms. The maximum RTDs show a dramatic drop in their values, reducing their combined average from 1715.6 to 772.7 ms. This drop affect the standard deviations, which experience heavy reductions as well, going from values in the region 175-215 ms to 25-45 ms. An interesting fact is the considerable drop in the minimum RTD values, now well below the minimum RTO value, and thus indicating a certain effect of the Fast Retransmit modification. The combined average of the minimum RTDs has decreased from 402.1 to 331.9 ms, an improvement of approximately 70 ms.

In these tests, packets are often collapsed during retransmissions, thus

```
# Connection (192.168.1.2:53746-192.168.2.2:12000)
# Retransmission region for packet 3053
#  3048     PKT t 748.453 seq 146762 len  35
#  3049     ACK t  748.66 ack 146797
#  3050     PKT t 748.773 seq 146797 len  73
#  3051     PKT t 749.045 seq 146870 len 294
#  3052     PKT t 749.173 seq 147164 len 282
#  3053 RTR PKT t 749.189 seq 146797 len 649
#  3054     ACK t 749.252 ack 147164
#  3055     ACK t 749.396 ack 147446
#   Retransmission count: 1
#   Time between first trans and last retrans: 415.895 ms
```

Figure 5.10: The modified collapse mechanism in action, extracted with **loss_details**

potentially reducing the need for retransmissions of succeeding packets, but it is only a couple of times there are more than 2 packets to collapse. Thus, the modified part of the collapse mechanism is only invoked a couple of times (see Figure 5.10), and it's effect compared to the original mechanism is thus minimal. To be precise, the modified mechanism itself is always invoked, but unless there are more than 2 packets to collapse, it's behaviour is not any different from the original mechanism. But the collapse mechanism itself is frequently invoked, which should be beneficial. However, as the original kernel has access to the original collapse mechanism as well, the modified kernel will only have an advantage when the modified part is used. As this rarely happens here, the difference between the kernels when it comes to this mechanism will be minimal.

If we compare the results here with those in Table 5.2, the latter's are marginally better, but we are only talking about a difference of a few ms. Some differences are not totally unexpected as the stream behaviour is quite different. As the time intervals between transmissions and the amount of data and packets transmitted are different, the results for the tests are not fully comparable. The most important fact is that both present considerable improvements in almost all aspects of the RTD statistics.

Kernel with only the No Exponential Backoff modification implemented

The results for this kernel (Table 5.7) are quite similar to the ones gotten in the former section, but some minor differences exist. The tendency is that the different RTD statistics are generally slightly higher in this kernel than in the completely modified version. The combined averages of the RTD statistics all give values a little higher than in the totally modified kernel (see Table 5.8).

However, the major improvement that can be seen, when comparing the results with the original kernel, must be subscribed to the implementation of the No Exponential Backoff mechanism, as the kernel with just this

TCP Variation	Packet Statistics		DupACK Statistics		Retransmission Time Statistics (in ms)			
	Total	Rexmt	dupACKs	No dupACKs	Min	Max	Avg	Stddev
RENO P	2208.1	207.3	25.6	36	448.5	761.7	420.2	29.9
RENO S	2249.1	203.1	54.7	14	397.9	902.7	423.1	49.5
RENO S F	2263.1	206.4	54.4	11	402.3	824.1	421.7	38.3
HYBLA P	2229.8	198.8	64.6	10	388.2	851.3	422.2	42.2
HYBLA S	2261.5	201.9	74.3	12	403.9	897.9	422.9	39.3
HYBLA S F	2246.8	198.6	73.3	10	394.1	804.1	422.4	36.2
Combined RTD Statistics					405.8	840.3	422.1	39.2

Table 5.7: Anarchy Online Statistics, Kernel with only No Exponential Backoff

Kernel	Combined RTD Statistics			
	Min	Max	Avg	Stddev
Original Kernel	402.1	1715.6	464.4	182.4
Modified Kernel	331.9	772.7	419.2	36.9
Kernel with No Exponential Backoff only	405.8	840.3	422.1	39.2

Table 5.8: Comparison of the different kernels

modification implemented, perform almost as well as the totally modified kernel. Thus, it is the most influential of the implemented modifications, but the others contribute as well. The Fast Retransmit contribution can be seen through the reduced minimum values, while for the Piggybacking modification, the modified part is seldom invoked, thus this will have a minimal effect. However, the collapse mechanism used in this modification is quite beneficial, and the modification has made it a general mechanism in thin streams, instead of a proc variable. Thus, the mechanism itself is quite useful.

5.4.4 Thin stream testing with dupACKs

As described in the previous section, the influence of the Fast Retransmit mechanism was limited due to the sparse generation of dupACKs, necessary for it's invocation. It seemed to have a certain effect, but we were not really able to test how well this enhancement could perform. Thus, we perform some additional testing with a different configuration to see how the modification performs when it is given better conditions.

TCP Variation	Packet Statistics		DupACK Statistics	Retransmission Time Statistics (in ms)			
	Total	Rexmt	dupACKs	Min	Max	Avg	Stddev
RENO P	10430	687	594	412.4	2381.1	557.2	237.4
RENO S	7843	540	473	331.2	2511.3	607.8	272.9
RENO S F	7406	468	407	413.6	1598.1	579.2	189.2
HYBLA P	7740	536	516	204.7	4416.2	618.5	380.8
HYBLA S	7713	510	484	204.7	2384.1	618.2	290.8
HYBLA S F	7725	521	504	204.8	4160.3	612.1	291.5
Combined RTD Statistics				295.2	2908.5	598.8	277.1

Table 5.9: Netperf Statistics, Message Size 1448 B, Original Kernel

The original kernel

For some reason, running the new test configuration in the original kernel caused the runs with Reno protocol considerable problems. Hybla's runs finished without any trouble, but the Reno runs stopped short, finishing well before the time requested for the run. The best attempt stopped about 14 minutes into the 30 min run. This is not optimal for the comparison between the different kernels, but the values gotten for the Reno runs roughly corresponds to the values gotten for the Hybla variation. Thus, I make the assumption that the values are representable for the Reno protocol, and adopt the same approach as was done in the Anarchy Online tests, running the tests several times and thus getting values for several connections and then calculate their combined average. Although this is unfortunate, it should not have that much of an influence as the Hybla results provide a certain impression of the performance in this kernel, and none of the previous (extensive) thin stream tests have shown any significant differences between the TCP protocol variations. Additionally, combining the average of several connections should still provide a quite representable impression of the Reno variation.

Apart from the problem just described, the increase in the amount of data requested to be sent results in a considerable jump in the RTD averages, compared with the results in Section 5.4.2. The average RTD has gone from somewhere in the region of 450-475 ms to values above 600 ms (see Table 5.9). The maximum RTDs have increased as well, while the only benefit seems to come from the lower minimum values, which should be due to the invocation of Fast Retransmit. The most promising aspect in these tests is that there are now plenty of dupACKs being created.

The modified kernel

The results (Table 5.10) show huge improvement in the RTD statistics compared to the results for the original kernel. The average RTDs are not that

TCP Variation	Packet Statistics		DupACK Statistics	Retransmission Time Statistics (in ms)			
	Total	Rexmt	dupACKs	Min	Max	Avg	Stddev
RENO P	7703	499	341	204.4	1727.9	442.5	186.5
RENO S	7687	483	357	204.4	1187.9	428.6	183.1
RENO S F	7728	526	373	204.4	1319.9	450.0	198.8
HYBLA P	7708	504	401	202.8	1688.0	413.4	194.0
HYBLA S	7739	535	397	204.5	2065.7	446.2	228.7
HYBLA S F	7745	542	406	202.3	1671.3	444.6	219.6
Combined RTD Statistics				203.8	1610.1	437.6	201.8

Table 5.10: Netperf Statistics, Message Size 1448 B, Modified Kernel

TCP Variation	Packet Statistics		DupACK Statistics	Retransmission Time Statistics (in ms)			
	Total	Rexmt	dupACKs	Min	Max	Avg	Stddev
RENO P	7707	503	443	207.6	1464.1	575.1	153.2
RENO S	7677	473	425	205.9	2999.4	505.3	164.2
RENO S F	7707	503	469	208.6	1208.1	507.8	129.9
HYBLA P	7745	541	481	204.9	2649.8	528.0	192.8
HYBLA S	7709	505	486	204.7	1344.5	546.1	189.8
HYBLA S F	7701	497	485	203.4	1536.0	564.0	200.8
Combined RTD Statistics				205.9	1867.0	537.7	171.8

Table 5.11: Netperf Statistics, Message Size 1448 B, Kernel with only No Exponential Backoff

far from the value level for the other tests run in the modified kernel, and show an improvement of 150-200 ms. Actually, the lowest average RTD, regardless of the test type, can be found in these results (413.4 ms). The maximum RTD show dramatic reductions as well, and the minimum values are still as low as they should be (close to the RTT).

The modification of the Fast Retransmit mechanism does not lead to a higher retransmission rate for these tests. The retransmission frequency for the original and the modified kernel is practically identical, with a rate of 0.0668 and 0.0667, respectively.

As the messages requested to be transmitted equal the SMSS, the Piggybacking modification cannot be invoked here, as there will never be any room in the packet for it to fill. However, it is uncertain how much of the improvement are due to the Fast Retransmit modification, and how much should be subscribed to No Exponential Backoff. To get a certain idea, we run the same test configuration in the kernel with only the No Exponential Backoff mechanism implemented.

Kernel with only the No Exponential Backoff modification implemented

The results for this kernel (Table 5.11) show considerable improvements compared to the original kernel, but the average RTDs are nowhere near the values gotten in the previous section. Thus, the Fast Retransmit modification should get a lot of credit for the improvements in this test configuration, while the No Exponential Backoff mechanism plays a smaller (but still considerable) part. The maximum RTDs show a considerable reduction from the original kernel tests, but they are still a little higher than for the completely modified kernel. The minimum RTD values still suggest a certain influence by original Fast Retransmit (3 dupACKs).

5.4.5 Evaluation of the implemented enhancements

Now that we have tested that the implemented enhancements work, are invoked, and seem to have an influence on the performance, it is time to reflect on what kind of impression they conveyed.

The original tests showed that the performance was far from optimal in thin stream settings with respect to the RTD. The preferred packet loss detection mechanism, Fast Retransmit, was hardly invoked at all, and this left the drastic RTO mechanism as the only detector of loss. The downfall of the Fast Retransmit influence was due to the almost complete non-existence of the dupACKs needed for it's invocation. Additionally, a certain aspect of the RTO mechanism did not help things. The Exponential Backoff mechanism doubles the RTO for each retransmission of the same segment, causing even higher RTD values. Even though it is not invoked very often, it's influence on game impression can still be unfortunate. Thus, as the RTO was practically the sole detector of loss, the RTD values gotten were quite high.

This impression was upheld for the additional testing performed here, with general high values for the RTD statistics, clearly indicating that the TCP protocol and it's variations are not optimized for this kind of traffic. Thus, leaving considerable room for improvements.

As the RTO mechanism seems to assume a considerably more important position in thin streams, it is essential to limit it's unfortunate effects. Thus, the Exponential Backoff mechanism was removed as this resulted in some extremely high RTD values, most unfortunate in game scenarios. This modification proved to be the most important enhancement for the tests performed here. For the tests with the original configuration and the Anarchy Online tests, this modification was the dominant factor for the improvement of the performance. Additionally, it had a significant influence in the last test configuration as well.

Thus, this modification is a most welcome and efficient enhancement for the thin stream performance. However, this modification only optimizes the performance of the least preferred loss detection mechanism. Even,

though it improves the RTD considerably, the RTD is still quite high. The less penalizing and potentially faster packet loss detection mechanism, Fast Retransmit, fails to become active in the thin stream settings, which is most unfortunate. It is to demanding when it comes to the number of dupACKs it requires before it's invocation. DupACKs are considerably more of a luxury in thin streams than they are in it's usual thick stream scenario.

Thus, dupACKs should be utilized when they occasionally occur, and Fast Retransmit was modified in an attempt to increase it's invocation frequency, reducing it's frexmt_thresh from 3 to 1. This should allow the Fast Retransmit mechanism to be triggered when the first dupACK is received, instead of having to wait for 2 additional ones. As this modification is entirely dependent on the generation of dupACKs, it's influence was limited in the tests where dupACKs were rare. However, when dupACKs were steadily created it showed it's potential, providing a huge improvement of the RTD values, without increasing the retransmission frequency.

As the RTO and the Fast Retransmit mechanism are the incorporated means to detect packet loss, and both have been modified here to provide lower RTDs, little more can be done here. However, the only thing better than having a low RTD is to avoid the retransmission altogether. Thus, a piggybacking mechanism was incorporated as well in an attempt to reduce the number of retransmissions. The mechanism utilized an already incorporated mechanism (tcp_retrans_collapse), implemented as a proc variable, which tried to collapse 2 packets during a retransmission, thus adding an extra packet to the retransmitted one. This mechanism was extended to add packets to the retransmitted one as long as there was room. The collapse mechanism itself performed smoothly, but the extension, collapsing 3 or 4 packets, was seldom used. This was due to the lack of additional outstanding data, as there rarely were more than 2 packets in flight. Thus, the main benefit provided by this modification was the discovery of the tcp_retrans_collapse mechanism itself, and additionally making this mechanism independent of the proc variable for thin streams. However, transmitting several original packets in one retransmission is still a considerable benefit.

Thus, the modifications all seem to improve certain aspects of the performance. Now it is time to move on to evaluate the different kernel versions' performance.

Figure 5.11 show the results for the different test types for each kernel. The following abbreviations are used in the figure:

`orig. conf. - The tests run with the original test configuration`

`anarchy - the Anarchy Online tests`

`dupACK - the thin stream tests with dupACK generation`

The modified kernel can present the best results, and show a consider-

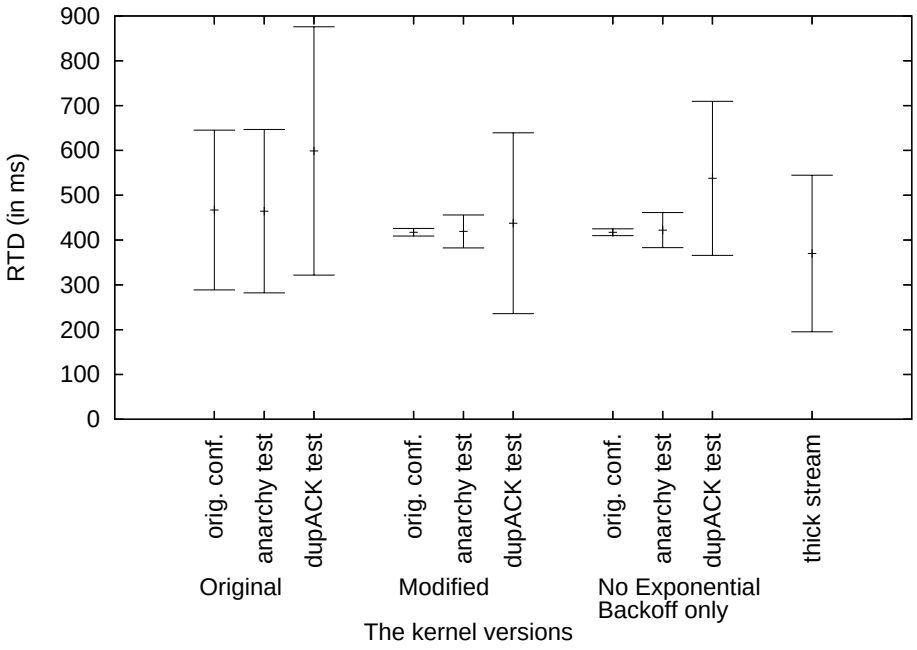

Figure 5.11: The combined RTD statistics for each test type in each kernel

able improvement compared to the original kernel, while the last version show the No Exponential Backoff mechanism's impressive influence.

Thus, the enhancements has resulted in a desired improvement of the performance, but how do their RTD values compare to the values gotten in the corresponding thick stream scenario? Table 5.12 show the values for the selected variations in the thick stream scenario, and the combined average RTD has been included in Figure 5.11. The results in this scenario are closely related to the throughput rate, and additionally to the enabling of SACK and FACK. The Hybla P run is not included in the following discussion due to it's abnormal behaviour.

The values for the modified kernel are not that far from the values gotten for the Reno protocol in the thick stream scenario. They get below the value gotten for the plain Reno protocol, but the enabling of SACK and FACK provide considerable improvement in the thick stream scenario, while they fail to produce the same effect in the thin stream scenario. Hybla's advantage is presumably related to it's superior throughput rate, which should result in a faster generation of dupACKs, and thus a faster and more frequent invocation of Fast Retransmit. As the most important characteristic of thin streams is their limited throughput, they logically do not possess the same opportunity.

Still, if we compare with the results gotten in the original kernel, the

TCP Variation	Packet Statistics		DupACK Statistics	Retransmission Time Statistics (in ms)			
	Total	Rexmt	dupACKs	Min	Max	Avg	Stddev
RENO P	11235	616		208.0	2088.1	443.4	227.7
RENO S	12643	655		208.0	1880.1	395.1	164.8
RENO S F	13003	649		208.0	1680.1	375.1	160.8
HYBLA P	25933	1459		206.4	5052.0	678.5	500.4
HYBLA S	38962	1985		204.2	3036.2	360.2	189.6
HYBLA S F	42821	2188		204.4	1888.1	275.9	130.1
Combined RTD Statistics				206.5	2604.1	421.4	228.9
(excluding Hybla P)				206.5	2114.5	369.9	174.6

Table 5.12: The thick stream RTD statistics for the selected variations, delay 100 ms, loss 5%

modifications have approximately halved the distance to the thick stream average, but they are still a long way from approaching the top thick stream value (Hybla S F 275.9 ms), which is almost a 100 ms better than the thick stream average. The difference is that in the thick stream scenario, differences between the individual variations exist, and there is usually always some variation(s) that lives up to the expectation and performs better than the others. In this scenario represented by Hybla with FACK. The thin stream scenarios seems to lack this quality, and thus they might not be able to compete with the thick stream top values.

However, when it comes to the maximum RTDs, the values gotten for thin stream scenarios, in the modified kernels, are better than in the thick stream scenario. This might be more important than the average RTD for certain thin stream applications, e.g. game settings, as long as the average RTD is still fairly low. It is usually the high maximum values that are detectable by the user.

5.5 Summary

In this chapter, we have described the implementation of the proposed enhancements, and then tested these enhancements for different test types to see if they indeed provided any improvements in the performance.

The implementation of the proposed modifications was done without any extensive tampering with the implemented TCP protocol. The removal of exponential backoff and the modification of Fast Retransmit were incorporated through a couple of additional if-tests, or an extension of existing ones. Neither the Piggybacking modification required any comprehensive alteration.

The tests showed that the modifications indeed resulted in a considerably improved performance, but some contributed more than others. The

No Exponential Backoff modification proved to be the most influential of the modifications, almost solely responsible for the improvement in 2 of the test scenarios, and additionally playing a considerable part in the last scenario. The Fast Retransmit modification depended on the number of dupACKs created, and thus had a limited influence in the test scenarios where their presence was sparse. However, it provided a huge improvement for the RTDs in the last scenario with a steady dupACK rate. The Piggybacking modification had the least influence, but this was due to the traffic flow nature in these tests. For 2 of the test scenarios, it was not invoked due to the test configuration, and even in the Anarchy Online scenario, there were seldom more than 2 packets in flight. Thus, the collapse mechanism itself was invoked fairly often, but there were seldom more than one packet that could be collapsed, and hence the modified part could not be triggered in most of the collapse scenarios.

Compared with the original kernel, the modifications approximately halved the distance to the average RTDs in the corresponding thick stream scenario, a significant improvement. Thus, we have tested the TCP variations, discovered serious weaknesses in their thin stream performance, proposed, implemented and tested several enhancements, it is time to wrap it up. This is done in the next and final chapter.

Chapter 6

Conclusion, future work

6.1 Conclusion

The objective of this thesis was to test and evaluate the TCP variations present in the Linux 2.6.15 kernel for thick and thin streams, in different RTT and loss scenarios, and improve TCP's performance for thin streams, as several of it's incorporated mechanisms were not optimized for this sort of stream.

The initial tests showed that most of the TCP variations performed satisfactory in the thick stream scenarios, with respect to the RTD. They differed considerably in the throughput rate, but this did only have a small impact on the RTD. The SACK and FACK options provided considerable improvements in the RTDs and are thus strongly recommended whenever available. If they are both available, FACK is the recommended option. Even though the performance was quite different among the protocols in an arbitrary thick stream scenario, it was always possible to find some variation that performed satisfactory.

For the thin stream settings, it was hard to find any differences between the variations. Some differences existed, but these were not consistent at all, and seemed to be arbitrary for the settings. The tests revealed an overall poor performance by the TCP variations. This was mainly due to the almost total absence of Fast Retransmit, which failed to get invoked due to the lack of dupACKs, thus leaving the RTO as the only means to detect packet loss. It's exponential backoff part lead to a number of extremely high RTDs, most unfortunate to experience in game scenarios.

To improve performance in the thin stream settings, several enhancements were presented. The removal of the exponential backoff mechanism was implemented to avoid the high maximum RTDs, while the number of dupACKs required before invoking the Fast Retransmit mechanism was reduced from 3 to 1, to utilize the few dupACKs that occasionally occur in thin streams. Piggybacking outstanding data in a retransmitted packet

was implemented to reduce the possibility of needing to retransmit several successive segments.

The testing of these enhancements showed considerable improvement in the performance, but the enhancements varied in their contribution. The removal of the exponential backoff mechanism proved to be the most influential enhancement for 2 of the 3 test types, and had a considerable effect in the last test type as well, providing an extreme decline in the maximum RTDs. The Fast Retransmit modification was suffering from the limited number of dupACKs present in the first 2 test types, but provided with a steady generation of dupACKs in the last test type, it showed it's potential and provided a huge improvement in the RTDs.

The influence of the Piggybacking mechanism was limited in these tests, but this was largely due to the nature of the streams tested. The implementation of the Piggybacking mechanism extended an existing feature in the TCP protocol, which was able to collapse 2 packets to 1 during a retransmission, thus adding an additional packet to the retransmitted packet. The mechanism was extended to continue collapsing, as long as there was room in the packet and additional data to fill it with. Due to the thin stream nature of these tests, it was limited how often and how much additional data were outstanding. The collapse mechanism could only be invoked in the Anarchy Online tests. Quite frequently, the retransmitted packet was the only one in flight, and thus there was nothing to add to the retransmission. When additional data was outstanding, it mostly consisted of 1 packet, thus the extented part of the collapse mechanism was not used. This part was only occasionally used in the connections. Even when the collapse mechanism was used, there was a considerable chance that the original transmission of the added segment(s) had successfully arrived, in which case the addition of the segment(s) did not have any effect.

Both of the loss detection mechanisms have been modified to provide a reduced RTD. If we have several packets in flight, there is a chance that we receive a dupACK, and in that case the modified Fast Retransmit mechanism should ensure a fast retransmission. If we have to wait for the RTO to expire, we have removed the exponential backoff mechanism to minimize the unfortunate effect of this mechanism, thus resulting in faster retransmissions if we need more than one attempt. Additionally, we can append additional outstanding data to the retransmission, reducing the possibility of needing to retransmit several successive segments. Combined, these mechanisms should provide a considerably improved and possibly quite satisfactory performance for thin streams as well.

6.2 Future work

To really be able to test the Piggybacking modification properly, a test scenario with a steady rate of 3 or 4 collapsable packets in flight should be used. In addition to the extended testing of the Piggybacking mechanism for retransmitted packets, the implementation of piggybacking for every transmitted packet is strongly recommended. Incorporating a full piggybacking mechanism should avoid a considerable part of the retransmissions altogether, which should have a significant impact. Additionally, reducing the initial 3 s RTO value should be considered.

If the piggybacking of every transmitted packet cannot be implemented, an alternative could be to repeat the transmission of a packet. That is, sending several copies of the same packet when it is requested to be transmitted. This will improve the chances of a successful reception of the first transmission. Additionally, if a previous packet was lost, this increases the chances of receiving dupACKs, and thus the triggering of the modified Fast Retransmit mechanism, the preferred loss detection mechanism. This should provide a similar effect, and should reduce the number of retransmissions considerably. When retransmissions do occur, the thin streams are now considerably better equipped to handle these, through the enhancements presented in this thesis.

Bibliography

[1] BIC TCP Home Page. http://www.csc.ncsu.edu/faculty/rhee/export/bitcp/index.htm.

[2] Forward acknowledgment: Refining tcp congestion control. http://web.it.kth.se/~haritak/project/details/FACK.html.

[3] H-TCP: TCP Congestion Control for High Bandwidth-Delay Product Paths. Internet Draft, obtained from: http://www.hamilton.ie/net/draft-leith-tcp-htcp-00.txt.

[4] How the Linux TCP output engine works. http://vger.kernel.org/~davem/tcp_output.html.

[5] LARTC Howto. http://lartc.org/howto/.

[6] Netem. http://linux-net.osdl.org/index.php/Netem.

[7] Netperf Homepage. http://www.netperf.org/netperf/NetperfPage.html.

[8] RFC1072: Tcp extensions for long-delay paths. http://rfc.net/rfc1072.html.

[9] RFC1122: Requirements for Internet Hosts – Communication Layers. http://rfc.net/rfc1122.html.

[10] RFC2018: Selective acknowledgment options. http://rfc.net/rfc2018.html.

[11] RFC2581: TCP Congestion Control. http://rfc.net/rfc2581.html.

[12] RFC2883: An extension to the selective acknowledgement (sack) option for tcp. http://rfc.net/rfc2883.html.

[13] RFC2988: Computing TCP's Retransmission Timer. http://rfc.net/rfc2988.html.

[14] RFC3168: The addition of explicit congestion notification (ecn) to ip. http://rfc.net/rfc3168.html.

[15] RFC3522: The Eifel Detection Algorithm for TCP. http://rfc.net/rfc3522.html.

[16] RFC3782: The NewReno Modification to TCP's Fast Recovery Algorithm. http://rfc.net/rfc3782.html.

[17] Sally Floyd's Homepage: HighSpeed TCP (HSTCP). http://www.icir.org/floyd/hstcp.html.

[18] Tcp westwood home page. http://www.cs.ucla.edu/NRL/hpi/tcpw/.

[19] Tcpdump/libpcap. http://www.tcpdump.org/.

[20] Tcptrace Official Homepage. http://jarok.cs.ohiou.edu/software/tcptrace/.

[21] The TC manual in Linux.

[22] Tibet introduction. http://www.elet.polimi.it/upload/martigno/tcp/node1.html.

[23] Yee's Homepage: HighSpeed TCP. http://www.hep.ucl.ac.uk/~ytl/tcpip/highspeedtcp/hstcp/index.html.

[24] Yee's Homepage: TCP Variants. http://www.hep.ucl.ac.uk/~ytl/tcpip/background/tahoe-reno.html.

[25] E. Blanton, R. Dimond, and M. Allman. Practices for TCP Senders in the Face of Segment Reordering. http://folk.uio.no/espensp/draft-blanton-tcp-reordering-00.txt.

[26] L. S. Brakkmo and L. L. Peterson. Tcp vegas: End to end congestion avoidance on a global internet. *IEEE Journal on selected areas in communications*, 13(8):16, 1995.

[27] C. Caini and R. Firrincieli. Packet spreading techniques to avoid bursty traffic in long RTT TCP connections. Technical report, 2004. http://folk.uio.no/espensp/01391456.pdf.

[28] C. Caini and R. Firrincieli. TCP Hybla: a TCP enhancement for heterogeneous networks. *International Journal of Satellite Communications and Networking*, 22(5):547–566, 2004. http://www3.interscience.wiley.com/cgi-bin/fulltext/109604907/ABSTRACT also available from: http://folk.uio.no/espensp/E018669126.pdf.

[29] N. Cardwell and B. Bak. A tcp vegas implementation for linux. http://flophouse.com/~neal/uw/linux-vegas/.

[30] S. Floyd. RFC3649: HighSpeed TCP for Large Congestion Windows. http://rfc.net/rfc3649.html.

[31] S. Floyd and K. Fall. Promoting the Use of End-to-End Congestion Control in the Internet. *IEEE/ACM Transactions on Networking*, (May 3), 1999. http://www.icir.org/floyd/papers/collapse.may99.pdf also available from: http://folk.uio.no/espensp/collapse.may99.pdf.

[32] M. Gerla, M. Y. Sanadini, R. Wang, A. Zanella, C. Casetti, and S. Mascelo. Tcp westwood: Congestion window control using bandwidth estimation. Technical report. http://www.cs.ucla.edu/NRL/hpi/tcpw/tcpw_papers/2001-mobicom-0.pdf.

[33] C. Griwodz. Lecture INF5070: Protocols without QoS Support, 2005. Available from: http://www.uio.no/studier/emner/matnat/ifi/INF5070/h05/undervisningsmat%eriale/04-nonqos-protocols.pdf or http://folk.uio.no/espensp/04-nonqos-protocols.pdf.

[34] C. Griwodz and P. Halvorsen. The Fun of using TCP for an MMORPG. Technical report. http://folk.uio.no/espensp/funtrace.pdf.

[35] T. Kelly. Scalable TCP: Improving Performance in Highspeed Wide Area Networks. Technical report, December 2002. Retrieved from: http://datatag.web.cern.ch/datatag/pfldnet2003/papers/kelly.pdf also available from: http://folk.uio.no/espensp/kelly.pdf.

[36] K. Kurata, G. Hasegawa, and M. Murata. Fairness Comparisons Between TCP Reno and TCP Vegas for Future Deployment of TCP Vegas. http://www.isoc.org/inet2000/cdproceedings/2d/2d_2.htm.

[37] D. Leith, R. Shorten, and Y.Lee. H-TCP: A framework for congestion control in high-speed and long-distance networks. Technical report, August 2005. http://www.hamilton.ie/net/htcp2005.pdf.

[38] R. Ludwig and A. Gurtov. The Eifel Response Algorithm for TCP. http://folk.uio.no/espensp/draft-ietf-tsvwg-tcp-eifel-response-03.txt.

[39] M. Mathis and J. Mahdavi. Forward acknowledgement: Refining tcp congestion control. *Association for Computer Machinery (ACM)*, 26(4):11, 1996.

[40] L. Peterson. Advanced protocol design. `http://www.cs.arizona.edu/protocols/`.

[41] R.N.Shorten and D.J.Leith. H-TCP: TCP for high-speed and long-distance networks. *Proc. 2nd Workshop on Protocols for Fast Long Distance Networks, Argonne, Canada*, 2004. `http://www.hamilton.ie/net/htcp3.pdf`.

[42] R. Wang, K. Yamada, M. Y. Sanadidi, and M. Gerla. TCP With Sender-Side Intelligence to Handle Dynamic, Large, Leaky Pipes. *IEEE Journal on selected areas in communications*, 23(2):14, 2005. `http://www.cs.ucla.edu/NRL/hpi/tcpw/tcpw_papers/WYSG05.pdf`.

[43] L. Xu, K. Harfoush, and I. Rhee. Binary Increase Congestion Control for Fast, Long Distance Networks. Technical report. `http://www.csc.ncsu.edu/faculty/rhee/export/bitcp.pdf` also available from: `http://folk.uio.no/espensp/bitcp.pdf`.