Experimental Validation of the TCP-Friendly Formula

Bachelor Assignment

Ruud Klaver (0004901)
r.klaver@student.utwente.nl

Design and Analysis of Communication Systems
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente
Supervisor: Dr. ir. Pieter-Tjerk de Boer

April 6, 2005

Abstract

In this report I will attempt to verify the accuracy of the TCP-friendly formula, which describes the bandwidth used by a TCP connection based on the RTT and loss rate of this connection. First, a new and more accurate formula, specific to a particular TCP implementation, is derived through analysis. This new formula and the original one are then compared in terms of accuracy by doing both controlled and Internet TCP measurements.

Chapter 1

Introduction

The Transmission Control Protocol, or TCP [15], is ubiquitously used in both public and private computer networks. It operates at the Transport Layer of the TCP/IP network layer model. Usually working on top of IP, TCP is a connection-oriented protocol that supplies a bi-directional stream of data between two hosts and handles such responsibilities as order correcting in case of packet reordering, retransmissions in the case of lost or incorrect packets, and congestion control. Congestion control means that a sending TCP host detects packet loss on the connection to the receiving host and throttles its sending rate accordingly, to avoid sending more data than the connection can handle. Several congestion control algorithms exist, but this report will only focus on the algorithm found in the classic Reno TCP implementation [9].

When using a different protocol than TCP at the Transport Layer, which in most cases will be UDP, this protocol may not perform congestion control. This means that application programmers using the UDP protocol need to implement some form of congestion control themselves. If such applications make use of connections on which TCP is also used, which will frequently be the case, it is important that these connections apply the same amount of throttling that TCP does. This is because if UDP connections are either less or more aggressive than TCP this will result in an unfair distribution of bandwidth. To assess the amount of bandwidth used the so called TCP-friendly formula [17, 19] exists. This formula gives an indication of the maximum bandwidth used by a Reno TCP connection on a transmission path with certain latency and loss characteristics. If a UDP application monitors these characteristics and applies this formula to throttle its outgoing transfer speed, in theory it should acquire the same amount of bandwidth on that connection as TCP applications do. This prevents any application from gaining an unfair advantage over others in terms of used network resources or, in a worst case scenario, from transmitting too many packets and causing a congestion collapse of the link.

This report will attempt to assess the validity of this formula in a realistic situation through obtaining measurements of TCP connections. Chapter 2 will provide a detailed description of the TCP-friendly formula. In chapter 3 the approach to obtaining the measurements is discussed. After this in chapter 4 a new and more precise formula will be derived for one particular TCP implementation, which shall be compared to the original formula in terms of accuracy. In chapter 5 the results of the measurements will be presented and compared to both the original and the new formula. Finally in chapter 6 there will be some concluding remarks.

Chapter 2

The TCP-Friendly formula

The TCP-friendly formula attempts to describe the behaviour of a TCP connection that is in equilibrium in the Congestion Avoidance state. To understand the inner workings of this formula, we first need to understand the TCP congestion avoidance algorithm that is associated with TCP Reno, which is defined in [23, 9]. Congestion avoidance is a mechanism that throttles the sending data-rate of a TCP connection, so that this does not exceed the maximum attainable data-rate of the IP path from the sending host to the receiving host. This is done by having the sending TCP host keep a record of how many bytes it may send without receiving an acknowledgement for those bytes, i.e. the number of bytes "in flight". This record is called the congestion window, or *cwnd*.

During the initial phase, which is called Slow Start, *cwnd* shall be exactly the Maximum Segment Size, or MSS, and every time the sending side receives an ACK it will double *cwnd*. MSS defines the maximum number of payload bytes the sender can transmit within a single segment, limited by the TCP header size, the headers of lower network layers and finally the path MTU. It is readily apparent that in this phase *cwnd* may grow exponentially. This exponential growth continues until *cwnd* reaches a threshold value, called *ssthresh*. When this threshold value is reached, the TCP connection enters the Congestion Avoidance phase, in which *cwnd* may increase at a maximum pace of one MSS per Round Trip Time, or RTT. In contrast to the exponential increase in the Slow Start phase, the Congestion Avoidance phase prescribes a linear increase. These two different phases are illustrated in figure 2.1, where both the exponential and linear increase can clearly be seen. Of course this growth of the number of outstanding unacknowledged bytes cannot continue indefinitely, since the bytes need to be stored somewhere between the sending and receiving host. Some of the bytes will actually "fill the pipe", i.e. they will actually be in transit on a line somewhere between the sending host and the receiving host, while others will be stored in buffers of routers along this path. What will happen on the network layer is that, as the sending data-rate increases, a router along the path that is experiencing congestion on the interface to which the packets must be routed will drop at least one of these packets, since they simply cannot traverse the link at that time and the input buffers on the incoming interface are already filled. Note that packets will be dropped in some way or another in both classic tail drop and Random Early Detection (RED) [13] routers, as well as in routers that implement other queue management algorithms. The receiving host will notice that a packet is missing by means of the TCP sequence numbers, and send either a duplicate ACK with the sequence number of the packet it is missing, or it will send a selective acknowledgement (SACK) if both hosts implement this TCP option. [18] Once the sending host has detected enough (where enough is usually three) duplicate ACKs or SACKs to assume that the loss of the packet is not caused by packet reordering on a lower layer (in which case the problem will solve itself eventually), but because of a router

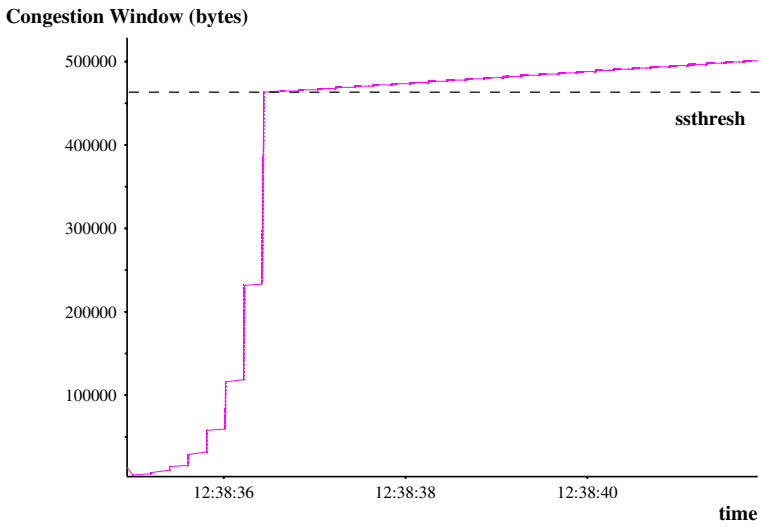

Figure 2.1: Graph describing the congestion window of a typical TCP connection, displaying both the exponential Slow Start phase and the linear Congestion Avoidance phase.

discarding a packet as previously described, it will go into what is known as Fast Retransmit / Fast Recovery.

When the sending TCP host enters this state it will immediately retransmit the segment that the receiving host is missing and set *ssthresh* to half the current *cwnd*. *cwnd* is set to the new value of *ssthresh* plus three segments, as three duplicate ACKs were already received to enter the state. For every consecutive duplicate ACK or SACK it then receives before it has received an ACK for the retransmitted segments, i.e. as long as the sending TCP host is not receiving "normal" ACKs, *cwnd* is increased by 1. This means that once this state is entered, the sender will have to wait until enough ACKs have arrived to raise *cwnd* above its old value before it can start transmitting new segments, because the number of segments that are in flight is equal to the old value of *cwnd*. Typically it would have to wait for about half of RTT, since it needs half the number of segments of this old value to start transmitting again. The other half of the RTT that it takes for the retransmitted segment to be acknowledged is used for sending segments at the normal transmission rate. The period that no segments are transmitted is used by the clogged router to free up its buffer, while the continual transmission of packets during this phase is performed because duplicate ACKs or SACKs indicate that the receiving TCP host is still actually receiving packets, and is thus an indication that the network can handle the transmission of more packets. After the ACK for the retransmitted segment has been received, *cwnd* is set to *ssthresh* (i.e. half its original value), and a new Congestion Avoidance cycle is started. This whole process is illustrated in figure 2.2. Note that, as it is a result that is obtained from measurements, the behaviour displayed in this graph does not exactly coincide with the theory presented here. The congestion window does not actually grow during Fast Retransmit and the slope of its growth is slightly curved. These are specifics of the TCP implementation used, which will be discussed in chapter 4.

Besides being limited by the congestion window, a TCP connection is also limited by the receiver window, at all time obeying the minimum of these two windows. The goal of this receiver window is to allow the receiving host to specify how much buffer space it has available to receive TCP segments. Although in this case the receiver window is only relevant pertaining data sent from the sender to the receiver, this window can be set in any TCP header. That means that in the case of

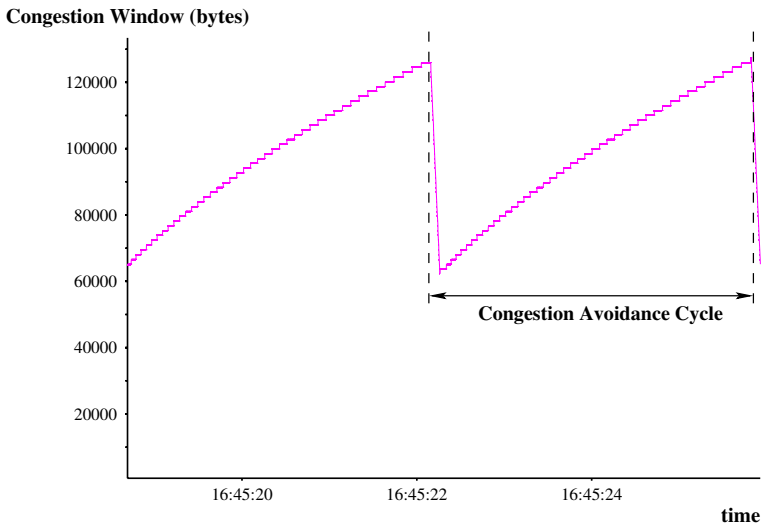

Figure 2.2: Graph describing the congestion window of a typical TCP connecting during two Congestion Avoidance cycles.

a sender-receiver scenario, the receiving host signals the size of this window through the headers of TCP ACK segments. The original TCP specification [15] reserves 16 bits in the header for the receiver window, allowing a maximum window size of 64kb. To prevent this receiver window from becoming the limiting factor on connections with a bandwidth-delay product in excess of 64kb, which is quite common nowadays, the TCP window scaling option was proposed [16]. This option allows a bitwise left-shift to be specified for the congestion window field of the TCP header, i.e. a multiplicative factor of 2^S, where S is the window scaling factor. This allows for receiver windows of up to 2^{30} (i.e. a gigabyte).

The observed congestion window of a certain TCP connection will, assuming that the characteristics of the transmission path of this connection remains constant, display a periodic pattern as illustrated in figure 2.2. The TCP-Friendly formula operates on exactly this premise. As stated before, it describes the behaviour of a TCP connection that is in equilibrium, i.e. one that has more or less constant characteristics and displays a certain pattern. A possible derivation is given in [19], which we will go through here. It simplifies the periodic sawtooth to make it a perfectly triangular shape, as can be seen in figure 2.3. In this graph, *cwnd* runs from $\frac{W}{2}$ to W, W being defined as the maximum *cwnd* size expressed in the number of windows of size MSS. Assuming that every segment is acknowledged by the receiving host and that the congestion window is opened at the maximum rate, the time taken to perform one Congestion Avoidance cycle is $\frac{W}{2} * RTT$. We can then count the total number of segments transferred by calculating the total surface area under the sawtooth for one cycle. Summing the rectangle and the triangle gives:

$$\left(\frac{W}{2}\right)^2 + \frac{1}{2}\left(\frac{W}{2}\right)^2 = \frac{3}{8}W^2 \tag{2.1}$$

If we define the probability that a segment is lost as p and that this event occurs once per Congestion Avoidance cycle, we can assume that the number of segments transferred also equals $\frac{1}{p}$. Using this equality we can express W as:

$$W = \sqrt{\frac{8}{3p}} \tag{2.2}$$

We can then express the bandwidth BW in terms of MSS, RTT and the maximum *cwnd* size

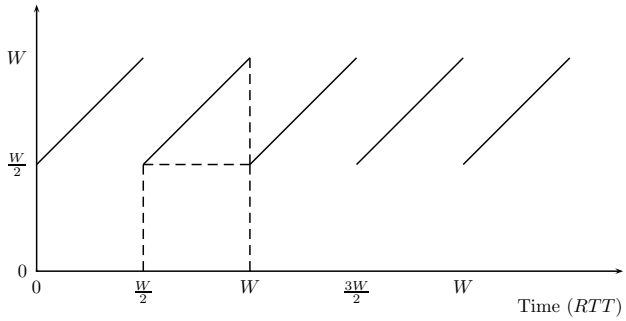

Figure 2.3: Simplification of the TCP congestion avoidance cycles as described in [19].

reached, W:

$$\begin{aligned} BW &= \frac{data\ per\ cycle}{time\ per\ cycle} \\ &= \frac{MSS * \frac{3}{8}W^2}{RTT * \frac{W}{2}} \\ &= \frac{\frac{MSS}{p}}{RTT\sqrt{\frac{2}{3p}}} \end{aligned} \tag{2.3}$$

This can be rewritten as:

$$BW = \frac{MSS}{RTT}\frac{C}{\sqrt{p}} \tag{2.4}$$

where $C = \sqrt{\frac{3}{2}} = 1.22$. Because the sawtooth pattern is grossly oversimplified, this formula is just a rough approximation of reality. The formula itself is therefore a rough approximation of the maximum bandwidth consumed by TCP connection with particular characteristics. Note that, as previously stated, it is assumed that the characteristics, i.e. the RTT, the MSS and the loss p, are constants.

Additional to the assumptions already mentioned, there are a number of other assumptions and conditions that the TCP connection must adhere to for the TCP-Friendly formula to be valid to that connection. We shall give the complete list of requirements as (partly) presented in [19] that are to be imposed on a TCP connection. When possible we shall try to illustrate this by showing a graph where the requirement is not met.

1. There is only one segment lost during a Congestion Avoidance cycle, which is the segment that triggers the congestion window to be halved.
2. The ACK strategy on the receiving side will be to send one ACK for every segment received. If the ACK strategy is to send out an ACK for every 2 segments received, C is effectively halved. However the ACK strategy must be consistently one or the other, so that C can be a fixed value and different connections can be compared to each other.
3. The advertised receiver window must be large enough for the congestion window to grow sufficiently on the sender side. Usually this means that both parties need to implement the

TCP window scale option. [16] An example of a connection where this is not the case can be seen in figure 2.4.

4. The sender must always have data available to send. If this is not the case there isn't always enough data available to "fill the pipe" and make the congestion window grow at a steady rate. This can be seen in Figure 2.6.

5. Both the sender and the receiver must implement the TCP SACK option. [18] This is because, if several segments are lost during a Congestion Avoidance cycle (which often occurs if the bandwidth-delay product is sufficiently large), the sender and receiver may go into a request-response scenario to determine and retransmit the missing segments, while not transmitting any other segments. This is caused by the fact that the receiver can only signal that it is missing one segment at a time to the sender in the form of duplicate ACKs. This scenario prevents the congestion window from exhibiting the typical Congestion Avoidance sawtooth. SACKs prevent this by allowing the receiver to transmit more detailed information about which segments it already has and which segments it is missing. This requesting and responding of missing segments can also cause the connection going into a timeout and subsequent slow-start, instead of performing normal Congestion Avoidance. The effects of this can already be seen in figure 2.1.

6. Window opening strategies other than Congestion Avoidance, such as the ones used in TCP Vegas [11, 12] and TCP BIC [24], shall not be used.

7. As stated before, the TCP connection must have reached a state of equilibrium in the Congestion Avoidance phase and should display a perfectly periodic sawtooth. It will often take the connection a few packet loss events to reach this state. A TCP connection that experiences a lot of packet loss will also not reach this state of equilibrium, which can be seen in figure 2.5.

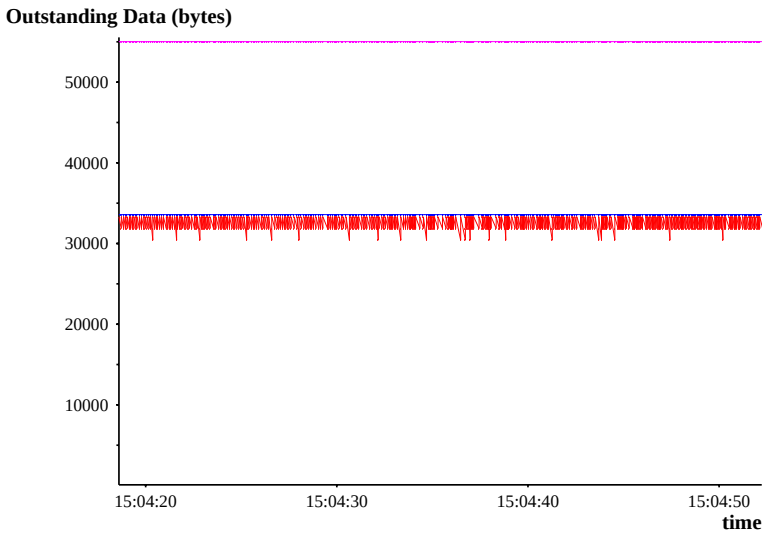

Figure 2.4: The red line in this connection, representing the number of bytes in-flight, is limited by the blue line on top of it, which is the receiver window. Also visible is that the congestion window, the purple line, is greater than the receiver window.

Besides this rather simplified derivation of the TCP-friendly formula, more complex ones exist such as [22]. This article assumes random loss characteristics instead of periodic loss, resulting in the same formula, but with $C = 1.3$. Because we are interested in the practical application of this

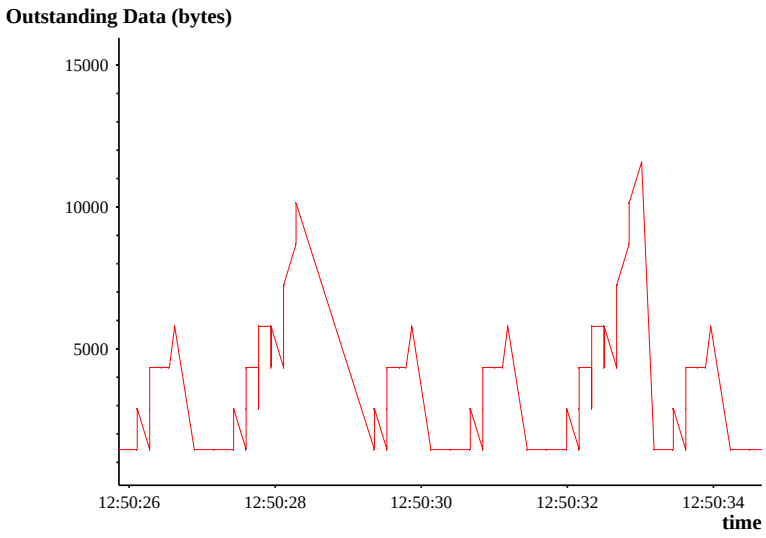

Figure 2.5: The number of bytes in flight on a typical high-loss TCP connection. Because of the high loss rate, a Congestion Avoidance equilibrium cannot be reached. Note that all values are multiples of MSS, which in this case equals 1448.

formula and not a purely theoretical one, I shall not elaborate further on the derivation of this factor.

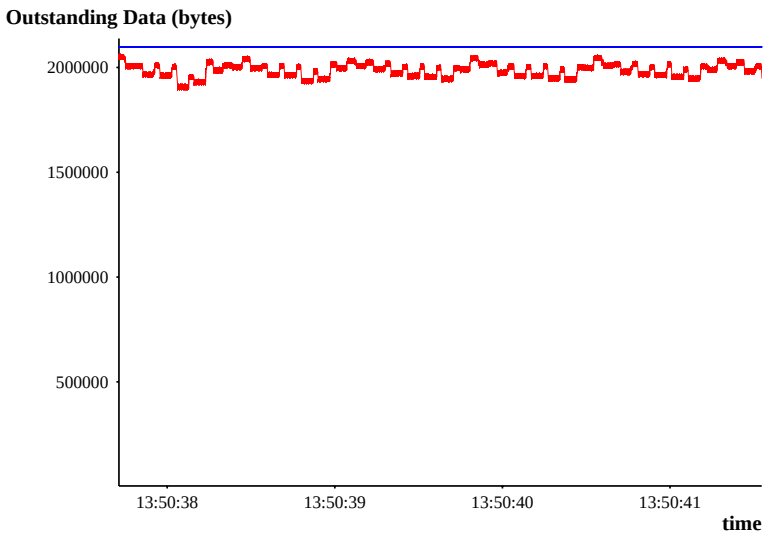

Figure 2.6: The red line represents the number of outgoing bytes on this TCP connection, the blue the receiver window. The sending site cannot generate enough segments to reach the receiver window. Amazingly this represents a connection from Western Europe to the US West coast.

Chapter 3

Approach

With all the conditions mentioned in the previous chapter, the question arises of where and how to measure TCP traffic to determine the exhibited behaviour. Although TCP is a bidirectional protocol by nature, the TCP-Friendly formula describes traffic going from a sending host and receiving host. To accurately perform RTT measurements, these measurements need to be done at the sending host of this scenario. This is because the relationship between a transmitted segment and its resulting received acknowledgement is the only direct causal one, and consequently the only time difference that can be measured as an indication for the RTT. The specifications for TCP timestamps [16] even state that a timestamp echo cannot be transmitted in a TCP packet that has no ACK flag set. In this sender-receiver scenario, the receiver will never get any ACK packets, besides the one for the connection establishment, so no reliable measurements can take place at the receiving end. This means that for the purposes of this research we need at least control over the sending host. Additionally, having control of the sending host has a number of other important advantages in satisfying the conditions presented in the previous chapter. In particular condition 4 (having enough data availlable to fill the pipe) and condition 6 (the window opening strategy must be classical Congestion Avoidance). If we do not have control over the sending party, making sure these conditions apply or even ascertaining that they do becomes difficult.

Ideally we should have control over both the sending and receiving host. If this is not the case, we should at least have control over the sending host, for the reasons stated above. In the case that we have no control over the receiving host, it is still difficult to satisfy some of the conditions in the previous chapter. Condition 2, which states that the ACK strategy should be to send one acknowledgement for every segment received, is one of these conditions. In the next sub-chapter we will discuss measures taken on the sending side to negate the effects of a varying ACK strategy, so that this condition need no longer be satisfied. Both condition 3 and 5 are impossible to remedy fully on the sending side, however. The former states that the TCP window scaling option should be used on both sides, and that this window scale should be sufficiently large to allow the receiver window to grow larger than the bandwidth-delay product of the pipe. From field trials it became apparent that for most hosts this was not the case by default. The latter states that both parties should use the TCP SACK option. This was also not found to be enabled for all hosts. As will be discussed later in this paper, satisfying these two conditions turned out to be one of the major problems in locating suitable testing hosts.

3.1 Linux 2.6

Because each TCP implementation specific to an operating system has subtle differences from implementations founds in others, a choice to study one of those particular implementations has to be made. For the purpose of this research, we will consider the TCP implementation as found in the 2.6.9 Linux kernel. As the source code for this operating system is readily available [3], the specifics of the implementation can be studied directly. Furthermore, as mentioned previously we will mostly be studying its sending side behaviour, meaning that, given the fact that the Linux operating system is widely used for server purposes nowadays, the research done will be representative for a large number of TCP connections. The peculiarities of the Linux kernel TCP implementation will be discussed in chapter 4, where a more accurate TCP-Friendly formula is presented. On the controlled systems used for measurements the *Gentoo Linux* distribution [1] was used.

One of the conditions that need to be satisfied for the TCP-Friendly formula to be accurate is that the acknowledgement strategy of the receiving host needs to be immediately sending one acknowledgement for every segment received. This means that the receiving side should not implement the so-called delayed ACK strategy [10]. From early field trials however it was clear that most, if not all TCP hosts do implement this. In fact, the specifications [9] dictate that this algorithm should be used and that implementing this means every second segment should be acknowledged. It also became clear that the receiving host is not always consistent in its choice. To negate the effects of any delayed ACK strategy, the congestion window growing algorithm of the Linux kernel was changed slightly to reflect the fact that an acknowledgement is received for several transmitted segments. The way that the Linux kernel performs Congestion Avoidance is to count the number of ACKs received. Once this counter reaches *cwnd*, *cwnd* is increased by one. By performing this counter increase the same number of times as the number of segments acknowledged, we essentially change the meaning of the counter from the number of ACKs received, to the number of segments acknowledged. This is perfectly compliant with the specifications [9], and is even suggested:

> Another acceptable way to increase *cwnd* during congestion avoidance is to count the number of bytes that have been acknowledged by ACKs for new data. (A drawback of this implementation is that it requires maintaining an additional state variable.) When the number of bytes acknowledged reaches *cwnd*, then *cwnd* can be incremented by up to $SMSS$ bytes.

The code for this slight change in the kernel can be found in appendix A. Note that strictly speaking we would have to remember the segment size of the segments that are acknowledged, since the acknowledgement just gives us a byte position and not a number of segments. Since condition 4 already requires there to always be data to send and because of the Nagle algorithm [20] that most hosts will implement, we can safely assume that all segments transmitted have the maximum segment size MSS.

Another adjustment made to the Linux kernel is writing the current *cwnd* somewhere in the TCP header so that it can be read out by a measuring application, the need for which will be discussed later. There were two options in doing this: either finding some unused bits in the TCP header that would not affect the normal operation of TCP when changed, or defining and adding a new TCP option. In terms of complexity, the latter would require more code to be added to the kernel than the first option, so the first option has preference. The urgent pointer of the TCP header [15] takes up 16 bits, by far sufficient to store the congestion window, and is rarely used, making it an ideal candidate. The change in kernel code for storing *cwnd* in the urgent pointer can be read in appendix A. Note that this code also makes sure that if the urgent pointer is used (i.e.

the URG flag is set), *cwnd* is not written in the urgent pointer field.

There were a few other settings that needed to be changed for the Linux kernel to meet the conditions of chapter 2. All of these could be set using the Linux *proc* filesystem, or the *sysctl* utility. These settings include disabling TCP BIC [24], which seems to be enabled as the default window opening strategy for Linux 2.6, and increasing the buffer sizes assigned to TCP connections, which in turn allows the window scale option to be set at a sufficiently high value.

3.2 Obtaining Measurements

TCP measurements were obtained by using a slightly modified version of the *trafficsnd* utility included in the *testrig* [7] package, which sends a lot of zero bytes to a host on a specific port during a specific period. This application was found to be more efficient than using *netcat* and the Linux */dev/zero* device. This was the only reason for using it, the rest of the testrig package is not used. The receiving hosts used in lab setups and public hosts were running a simple *discard* service as found in any Linux distribution.

All measurements of TCP traffic were performed using *TCPDUMP* [8]. This application allows selectively snooping data on network interfaces and dumping this data to disc. The data was then analysed using *tcptrace* [21], a program that retrieves a host of statistics from raw *TCPDUMP* files. It can also produce graphs of amongst others the RTT and the number of bytes that are in flight at any point in time. These graphs can then be viewed in a specific application, called *xplot*. Since it can not directly measure the size of the congestion window, the number of bytes in flight serves as an indication of *cwnd*, assuming that the sending host is constantly transmitting bytes up to the number it is allowed to have in flight. This should already be the case because of condition 4. This indication however is not sufficient to analyse exactly what is happening inside the Linux kernel. For this reason the slight adjustments to the kernel mentioned previously were made. Of course this means some adjustments also had to be made to *tcptrace*, allowing the real *cwnd* to be drawn into the graph of the number of bytes in flight. The changes to the code can be found in appendix B.

When taking TCP measurements, we should also consider condition 1 of our list in chapter 2, which states that only one segment should be lost during a congestion avoidance cycle for the purposes of calculating the loss p. Frequently more than one segment is lost during the recovery phase of a cycle due to the inability of the sending TCP host to respond fast enough to the loss of a segment. In order to still gain measurements for which the TCP-Friendly formula should be valid, we can redefine the loss of a single segment when calculating p to the loss of several segments that are closely together in time, calling this a loss event. For TCP connections where more than one segment loss occurs per loss event this sacrifices only a small amount of accuracy, but this is preferable to discarding the measurement completely. This loss event is also used in [19]. The overall loss p can then be calculated as:

$$p = \frac{loss\ events}{total\ segments\ transmitted} \tag{3.1}$$

The number of loss events can be obtained manually by inspecting the graph of the number of outstanding bytes and "counting the peaks", but for long measurements this can be very tedious. To simplify the gathering of measurements, the counting of loss events was implemented in TCP, defining a loss events as a series of segments lost within twice the RTT of one another. Although this may seem like an arbitrary measure, in practice this is a very efficient way of separating different loss events. In [19] a range the length of one RTT is used, but this was found to be inaccurate. The changes in code to *tcptrace* for implementing this can also be found in appendix

B.

A lot of other small adjustments were made to the *tcptrace* code, mainly dealing with displaying debug information, so that the behaviour of the sending TCP host could be more precisely analysed. The debug information statements can also be seen in the source code in appendix B.

Two types of measurements were done, one in the controlled environment of the lab, where we had control of both the sending and receiving side and could influence a range of characteristics, the other on the Internet, where only the sending side was under our control. To emulate different situations and characteristics in the lab, the Linux traffic shaping features were used. Traffic shaping in the Linux kernel consists of a number of "queueing disciplines" or qdiscs that can be attached to an egress interface to control the queueing behaviour of traffic. To emulate different bandwidth sizes, a simple token bucket filter was attached to the Ethernet interface of the sending machine. As well as restricting the maximum sending rate, this filter allows its buffer size to be set, emulating different router buffer sizes. Latency that is not related to router buffers was emulated using the *netem* [5] qdisc, which allows a set delay in milliseconds to be added to an interface. All measurements performed ran for 5 minutes or more.

The problem with using these queueing disciplines is that the packet capturing in the Linux kernel actually takes place after traffic shaping. This means that any measurements done do not see the qdiscs as part of the line. To circumvent this, the Ethernet bridging feature of the Linux kernel was used, creating a bridge with only one interface attached. The behaviour of such a bridge is exactly the same as the normal interface, with the advantage that qdiscs can be attached to the actual interface, while measurements take place on the bridge, before passing through the queueing disciplines. The qdiscs are thus seen as part of the line, allowing us to emulate the line characteristics. This setup can be seen in figure 3.1.

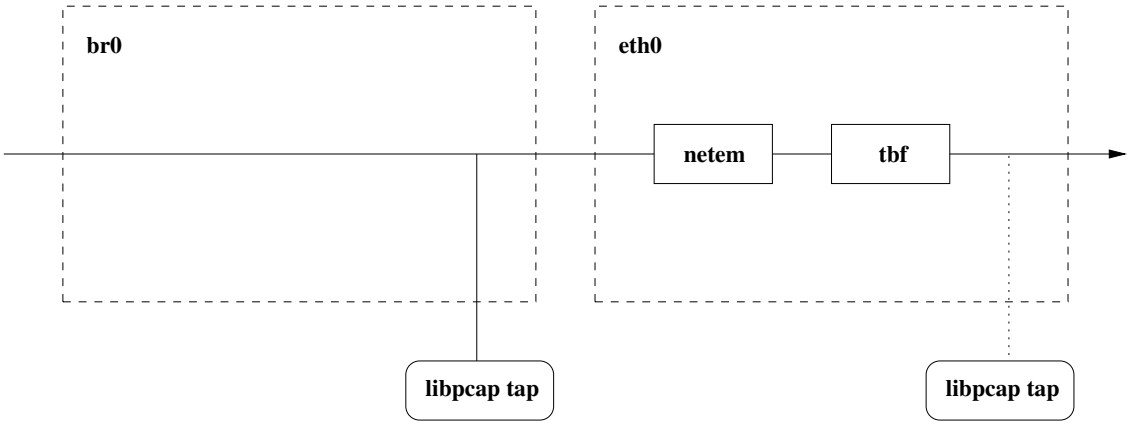

Figure 3.1: Schematic of the outgoing interface of the sending host with traffic going from left to right. Traffic is measured at the libpcap tap of the bridge interface br0, with traffic shaping being performed on the actual interface eth0. If measurements were performed on the libpcap tap of the eth0 interface, the traffic shaping would not be considered to be part of the line.

The Internet measurements required us to find receiving hosts on the Internet that had a discard service running and obeyed all the conditions of chapter 2. This was surprisingly hard to achieve and in practice this meant that all receiving test hosts had to have someone operating them locally.

Chapter 4

A New Formula

As [19] states, the model used to obtain the TCP-Friendly formula is only an approximation of the behaviour of TCP congestion avoidance. In this chapter I will attempt to define a more accurate formula that precisely describes the behaviour of TCP congestion avoidance as it is exhibited by the Linux 2.6 kernel. Comparing this new formula to the classic TCP-Friendly formula, we can then draw conclusions on its accuracy and applicability.

One simplification that is used as a basis for the TCP-friendly formula is that the value of the congestion window *cwnd* describes a "sawtooth" pattern in a perfectly straight line, see figure 2.3. In practice however this will always be a series of small steps instead of a sloped line, which can already be seen in figure 2.2. This is because TCP implementations will generally implement Nagle's algorithm [20], which dictates that when there are bytes in-flight, new segments may only be transmitted if they have the maximum possible size MSS. This, together with condition 4 of chapter 2, guarantees that the data actually sent is always a multiple of MSS, thus providing a step pattern. Secondly, in modern TCP implementations, such as the Linux kernel, *cwnd* is stored in units of segments with size MSS rather than bytes, again guaranteeing that the number of bytes sent is a multiple of MSS. Another thing that can be seen in figure 2.2 is that the general line of the *cwnd* steps is slightly curved. This is because, in practice, the bandwidth BW is a constant factor throughout, while RTT varies and not vice versa, as [19] would suggest. The IP connection bandwidth is often limited by a single bottleneck link between two routers, which determines the (constant) bandwidth of the TCP path. As more and more segments are allowed to be transmitted, the buffer in the router right before this link fills up, gradually increasing the RTT. This causes each step in the *cwnd* staircase to last slightly longer than the previous, causing a somewhat curved staircase.

We can use the first property to revise the formula that describes the relationship between the maximum congestion window W and the loss probability p. The second property will be discussed later. One should note that, depending on W being an odd or even number, the behaviour is slightly different. Instead of calculating the surface area under the *cwnd* line, we can deduce two formulae based on the summation of the areas under the individual steps. The derivations for the formulae will be illustrated using two example connections, one where W is odd ($W = 11$) in figure 4.1 and one where W is even ($W = 12$) in figure 4.2.

For these formulae to be as accurate as possible, we need to know exactly what is going on inside the Linux kernel. As stated before, the Linux kernel keeps a record of *cwnd* in terms of packets with size MSS. It is allowed to increase *cwnd* by one once it has received exactly *cwnd* acknowledgements from the receiving host. Since by definition it takes one RTT to transmit

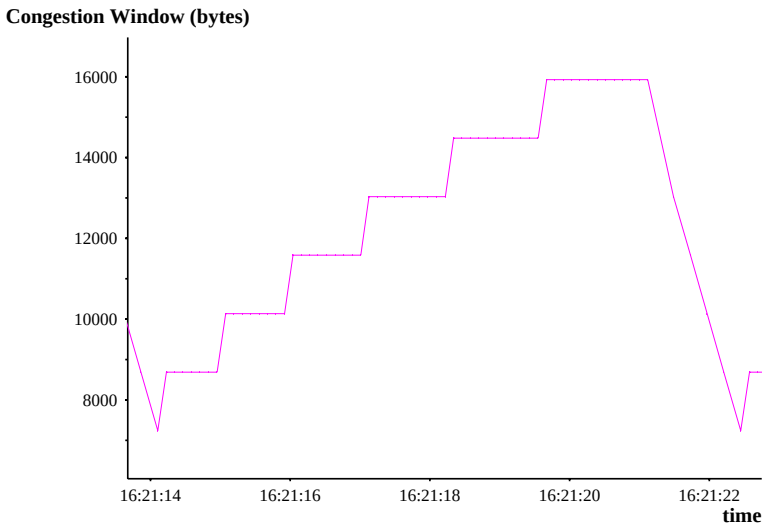

Figure 4.1: A full Congestion Avoidance cycle with $W = 11$. Note that in the staircase every first dot on a new plateau is actually two segments being sent right after eachother.

cwnd segments of MSS bytes, this theoretically should allow *cwnd* to grow by one every RTT, as is defined in [9]. There is one quirk in the Linux kernel however. The statement checking the ACK count to see whether it can increase *cwnd*, performed on reception of acknowledgements, is performed before incrementing the ACK counter. The code for this is included in appendix A. In practice this means that it will actually wait for $cwnd + 1$ ACKs to arrive before increasing *cwnd* by one. This is not strictly incorrect, since [9] only dictates a maximum speed of growth, but it does not allow *cwnd* to grow as aggressively as possible. To illustrate the complete behaviour, we shall now go through exactly one congestion avoidance cycle that occurs once the TCP connection is in equilibrium. We assume that all the conditions presented in chapter 2 apply.

The starting point for our cycle shall be the moment that the sending TCP host is allowed to increase *cwnd* to W, attaining its maximum value. This permission is granted by means of receiving the Wth (N.B.: not $(W-1)$th) acknowledgement from the receiving host. At this instance it is allowed to have two more segment in flight (i.e. unacknowledged) than it currently has, i.e. a segment that may be sent as a result of the received acknowledgement and an "extra" segment as a result of the increased *cwnd*. Both of these segments will then be transmitted by the sending host. Now, somewhere along the connection path will be the IP hop that is the bottleneck for the entire connection. The buffer in the router directly before this bottleneck link (from the point of view of the sending TCP host) will have been gradually filling up as *cwnd* was increased. The segment that is caused by the reception of the acknowledgement will neatly fit in the buffer, but the "extra" segment transmitted will not be able to fit in the already full buffer and will be discarded by the router. The sending TCP host however will not know this (we assume that ECN is not used). Blissfully unaware, it will continue transmitting segments as acknowledgements for segments sent before the lost segment come in. At some point of course the receiving host will receive a segment that has an inconsistent sequence number and from that it can see that a segment is missing. As a response for this inconsistent segment it will send a SACK, stating that it has received this segment, but that the segment before this one is missing. This SACK will arrive at the sending side one RTT after it has sent the segment after the one that got discarded, which as you may recall equates to $W-1$ packets later, because all the links and buffers along the transmission path can cumulatively accommodate that many packets. At this point the sending TCP host starts paying attention, but it does not know if the SACK was caused by packet

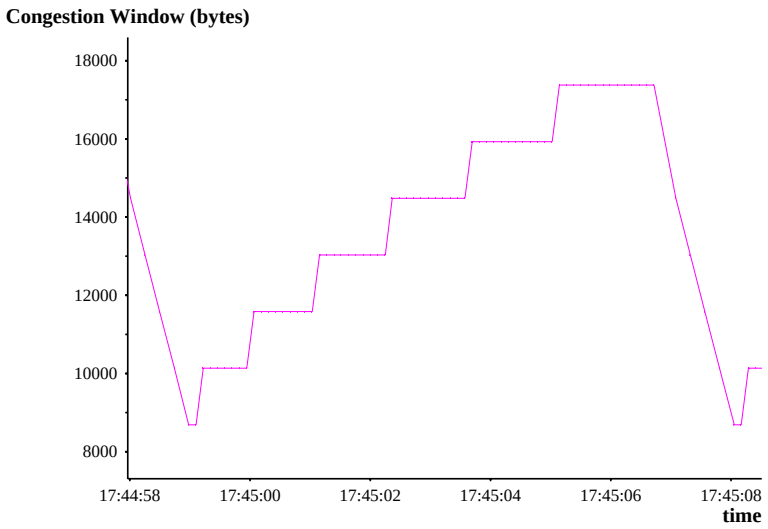

Figure 4.2: A full Congestion Avoidance cycle with $W = 12$. Note that in the staircase every first dot on a new plateau is actually two segments being sent right after eachother.

reordering along the route to the receiving host or by actual packet loss. In case it was just packet reordering the problem should resolve itself eventually and the sending host continues transmitting a segment as response to the SACK, because it actually does indicate that the segment sent after the lost one has left the network and according to *cwnd* it is allowed to "replace" that packet in the pipe. The SACK however does not increase the ACK counter, so *cwnd* will not be increased as a result of it. This continues until it receives the third SACK, when it decides that there actually was no reordering and the segment is lost, akin to the three-duplicate-ACKs rule in normal Fast Retransmit. As a response to this it directly retransmits the lost segment, without transmitting a new segment, and also resets the ACK counter to 0. As was said in chapter 2, *ssthresh* is now set to half of *cwnd*. The number of segments transmitted during this period can be expressed as $1 + 0 + (W - 1) + 2 + 1$, or $W + 3$.

We have now entered what is known as the Fast Recovery phase. Now instead of waiting for half a RTT and sending nothing and then sending again at the normal rate after that period, as is described in [9] and chapter 2 of this report, Linux implements a form of "rate-halving", as first described in [14]. This means that once the sending host has decided that the segment was lost and has retransmitted this lost segment, it sets *cwnd* to the number of outstanding segments, which should be $W-1$. On every SACK it subsequently receives, it will alternate between lowering *cwnd* by one and transmitting a segment, effectively halving the rate at which it transmits. This allows the bottleneck buffer in the transmission path to gain some free space. Because the retransmitted segment is acknowledged by the $(W - 1)$th incoming ACK, counted after the retransmission, a total of $\frac{W-2}{2}$ segments will be transmitted during this period, rounded down. This means $\frac{W}{2} - 1$ in case W is even and $\frac{W}{2} - 1\frac{1}{2}$ in case W is odd. We can also conclude that *cwnd* will have been reduced by one $(\frac{W-2}{2})$ times, rounded up. Since directly after the retransmission we know that $cwnd = W - 1$, this means that *cwnd* ends up at $W - 1 - (\frac{W}{2} - 1) = \frac{W}{2}$ if W is even, and $W - 1 - (\frac{W}{2} - \frac{1}{2}) = \frac{W}{2} - \frac{1}{2}$ if it is odd. Or in short, the new *cwnd* is half of what it was before, rounded down.

We should note that *cwnd* now equals *ssthresh*. In this situation, [9] states that the implementation is free to choose whether it wants to perform Slow Start or Congestion Avoidance. Linux

actually chooses to do Slow Start in this situation, so *cwnd* is immediately increased by one when it receives the next ACK. So now we start the typical Congestion Avoidance step pattern with $cwnd = ssthresh+1$. There is only one snag in the Linux kernel. During the Fast Recovery phase, it uses the ACK counter register to indicate its behaviour on the next received ACK, storing a 1 when *cwnd* should be reduced on the next ACK and storing a 0 when a new segment should be sent. This means that when W is even, this counter is left at a value of 1 when the ACK for the retransmitted segment is received. It is then not reset by the kernel when it enters Slow Start for one segment and stays 1 when it enters Congestion Avoidance. The effect of this is that for the first step of this phase, it sends one segment less than it really should. This step pattern then represents the bulk of the segments sent and behaves quite rationally. As stated before, during each step the sending host will wait for $cwnd + 1$ ACKs to arrive. Additionally, it is allowed to send one extra segment when it is allowed to increase *cwnd* by one. The number of segments sent during this phase when W is odd can be summed up by the following formula:

$$1 + \sum_{x=\frac{W}{2}+\frac{1}{2}}^{W-1} (x+2) \tag{4.1}$$

In case W is even, this should be the following formula:

$$1 + \sum_{x=\frac{W}{2}+1}^{W-1} (x+2) - 1 = \sum_{x=\frac{W}{2}+1}^{W-1} (x+2) \tag{4.2}$$

We can now add the formulae for the different phases together to form the formula describing the total number of segments sent during one complete Congestion Avoidance cycle, which equates to $\frac{1}{p}$. For the case where W is odd this results in:

$$\begin{aligned}
\frac{1}{p} &= W + 3 + \frac{W}{2} - 1\frac{1}{2} + 1 + \sum_{x=\frac{W}{2}+\frac{1}{2}}^{W-1} (x+2) \\
&= \sum_{x=\frac{W}{2}+\frac{1}{2}}^{W-1} (x+2) + 1\frac{1}{2}W + 2\frac{1}{2} \\
&= \cdots \\
&= \frac{3}{8}W^2 + 2W + \frac{13}{8}
\end{aligned} \tag{4.3}$$

And if W is even this results in:

$$\begin{aligned}
\frac{1}{p} &= W + 3 + \frac{W}{2} - 1 + \sum_{x=\frac{W}{2}+1}^{W-1} (x+2) \\
&= \sum_{x=\frac{W}{2}+1}^{W-1} (x+2) + 1\frac{1}{2}W + 2 \\
&= \cdots \\
&= \frac{3}{8}W^2 + \frac{7}{4}W
\end{aligned} \tag{4.4}$$

Note the resemblance to (2.1). If we solve the odd formula for W using the ABC formula we get:

$$\begin{aligned}
W &= -\frac{8p - \sqrt{25p^2 + 24p}}{3p} \\
&= \sqrt{\frac{25}{9} + \frac{24}{9p}} - \frac{8}{3}
\end{aligned} \tag{4.5}$$

And the even formula:

$$\begin{aligned} W &= -\frac{7p - \sqrt{49p^2 + 24p}}{3p} \\ &= \sqrt{\frac{49}{9} + \frac{24}{9p}} - \frac{7}{3} \end{aligned} \tag{4.6}$$

In (2.3), RTT and W are used to express the total length of a single Congestion Avoidance cycle, which in turn is used to calculate the total bandwidth. Because in practice RTT varies with he number of segments that are in-flight however, we cannot provide an accurate formula for the length of a cycle. Instead we will use a different approach to calculate the bandwidth BW. Let us make a model of the transmission of a single segment, which we can safely assume to be of size MSS. At the time of transmission we know how many segments there are in-flight including this segment, as this should be equal to *cwnd* (assuming *cwnd* is kept in segments, not bytes). Let us call the number segments that are in flight for segment N F_N. Once the ACK for the segment has arrived we also know the RTT for this particular segment and thus we know the time it takes to fully transmit and ACK F_N segments. Let us call this RTT measurement RTT_N. We can then express the the bandwidth measurement obtained from this segment as:

$$BW = \frac{MSS * F_N}{RTT_N} \tag{4.7}$$

Because on a connection we will generally measure RTT as an average round trip time of all unambiguous segments, we can generalise this equation. To do this we define F_{avg}, which is the average number of segments in-flight obtained from all segments that provide valid (i.e. unambiguous) RTT measurements. Generalising (4.7) gives:

$$BW = \frac{MSS * F_{avg}}{RTT_{avg}} \tag{4.8}$$

Note that in all formulae from [19] this RTT_{avg} is written as RTT, as shall be the case in the rest of this report. As this equation provides the average bandwidth over all measured segments, this should be a very accurate estimate of BW, using as many data as possible. When we solve this equation for F_{avg}:

$$F_{avg} = BW * RTT/MSS \tag{4.9}$$

When we look at [19], we can see that this is actually a familiar equation. It is used in most of the graphs in this article as a "performance based estimate of the average window size".

At this point we should explain something about the ambiguous ACKs mentioned before. An application measuring a TCP connection cannot blindly obtain a RTT measurement from every segment it sees. Some segments generate so-called "ambiguous ACKs", i.e. an ACK for which cannot be determined which transmitted segment caused it. Because, as explained earlier, a RTT measurement needs a direct causal relationship between one transmitted segment and one received segment, a valid RTT measurement cannot be obtained from these ambiguous ACKs. As all RTT measurements in this article were done using TCPTRACE, we will use its definition of an ambiguous ACK [21]. Its manual specifies two cases of an ambiguous ACK. The first case is the acknowledgement of a segment that was retransmitted. This ACK could be caused by the original transmission of the segment or any of its subsequent retransmissions. The second case is the ACK for a segment that has segments before it that are not acknowledged yet. An ACK for this segment could be received because the actual segment was received on the other side of the connection, or because a missing segment before it was received. This has to do with the fact that TCP does not acknowledge individual segments, but instead notifies the sender which segment it is expecting to receive next. In practice this means that no measurements can be done for SACKs. The result of all this is that RTT is not the average round trip time of all segments, discarding

a small number of segments each cycle. Consequently when we will be calculating the F_{avg} we will also discard the number of segments that are in-flight for those segments that do not generate valid RTT measurements. This provides for a more accurate indication of the bandwidth BW. The end result however is that some segments each cycle are completely ignored for the bandwidth calculation, but as in this model we assume BW to be constant this does not matter. In practice BW should stay relatively the same during this period and because we are using as much data as possible we should be able to accurately extrapolate the missing points.

By inspecting the RTT graph of a Congestion Avoidance cycle we can see which segments should be taken into consideration, i.e. which ones generate valid RTT measurements, and also what *cwnd* values are used at this point. This is because the RTT values will generally also describe a staircase, increasing by one step every time one segment more is added to the buffer. The size of the step is then the time it takes to transmit one segment in milliseconds. This can be seen for both $W = 11$ and $W = 12$ in figures 4.3 and 4.4 respectively. The derivation of F_{avg} in the next

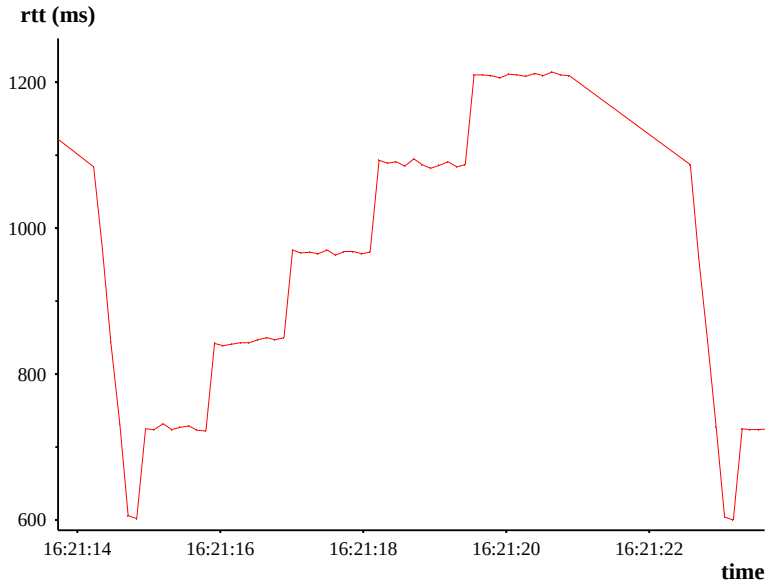

Figure 4.3: Round Trip Time graph for a full Congestion Avoidance cycle where $W = 11$.

paragraph corresponds to these graphs.

We can calculate the exact value of F_{avg} by adding up the number of segments that are in-flight for each segment that generates a valid RTT measurement and dividing that by the amount of these measurements. We will start counting at the same point we did when we were counting the number of segments transmitted, i.e. the moment that the sending TCP host receives the ACK that brings *cwnd* to its highest value W. The segment it then transmits as a result of the ACK generates one valid *cwnd* measurement of $W - 1$. The "extra" segment is of course lost, so it will never generate an ACK and accordingly no RTT measurement. Any segment sent after this, up to and including the retransmission of the lost segment, will not generate valid RTT measurements and should be discarded. Valid measurements start once again when the sending TCP host receives acknowledgements from the segments it sent after the retransmission. We had already determined that the number of segments sent during the Fast Recovery period is $\frac{W}{2} - 1$ in case W is even and $\frac{W}{2} - 1\frac{1}{2}$ in case W is odd. Because during this time the sending host alternates between sending a new segment and reducing *cwnd* by one in response to incoming ACKs, *cwnd* for these segments will be sequentially lower by one for each segment transmitted. In case W

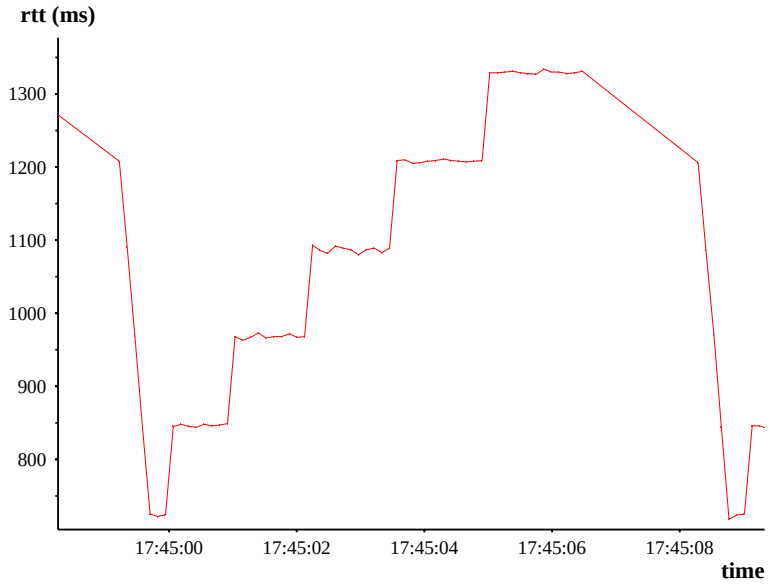

Figure 4.4: Round Trip Time graph for a full Congestion Avoidance cycle where $W = 12$.

is even this means that the congestion window runs from $W - 2$ (remember that it was set to $W - 1$ for the retransmitted segment and that *cwnd* lowering is performed first) to, in $\frac{W}{2} - 1$ measurements, $W - 2 - (\frac{W}{2} - 1 - 1) = \frac{W}{2}$. For the case that W is odd the measurements run from $W - 2$ to $W - 2 - (\frac{W}{2} - 1\frac{1}{2} - 1) = \frac{W}{2} + \frac{1}{2}$ We should keep in mind here that the last event occurring as the result of a SACK, before the retransmission is acknowledged, in case W is odd is a *cwnd* reduction, but when W is even this is the transmission of a new segment. This means that, as the next ACK comes in that is a direct result of the single slow start step, the buffer has not been allowed to empty out further in case W is even, but for the case where W is odd it is one lower. This means that the number of bytes in-flight at the time of transmission of this segment was $\frac{W}{2}$ in the even case and $\frac{W}{2} - \frac{1}{2}$ in the odd case. For the next segment, the sending TCP host was allowed to transmit two segments in short succession, because *cwnd* was increased due to the single slow start step. One of these segments will result in a measurement of $\frac{W}{2}$ if W is even and $\frac{W}{2} - \frac{1}{2}$ if W is odd, the other segment will result in measurements of one segment higher in both cases. For the next few segments the number in flight will stay the same as *cwnd* follows its typical Congestion Avoidance stair step pattern. This means that it will result in $cwnd + 2$ measurements of *cwnd* segments being in the buffer, but one of these measurements is from the step where *cwnd* is actually one higher, i.e. the first of the two segments that are sent in short succession of each other. There is one exception to this, and that is the first step in case W is even. Because the *cwnd* counter is still set to 1, it will do one measurement less. This step pattern continues on to $W - 1$ segments being in flight, of which we can see $W - 1 + 1$ measurements. The next measurement is the result of the ACK that allows *cwnd* to increase, which is the first measurement of our new cycle. We can now express F_{avg} in the way we previously described. For the case where W is odd this leads to:

$$F_{avg} = \frac{W - 1 + \sum_{x=\frac{W}{2}+\frac{1}{2}}^{W-2} x + \frac{W}{2} - \frac{1}{2} + \frac{W}{2} - \frac{1}{2} + \sum_{\frac{W}{2}+\frac{1}{2}}^{W-1} (x * (x+2)) - (W-1)}{1 + \frac{W}{2} - 1\frac{1}{2} + 1 + 1 + \sum_{\frac{W}{2}+\frac{1}{2}}^{W-1} (x+2) - 1}$$

$$
\begin{aligned}
&= \frac{\sum_{\frac{W}{2}+\frac{1}{2}}^{W-1}(x^2+3x)}{\sum_{\frac{W}{2}+\frac{1}{2}}^{W-1}(x+2)+\frac{W}{2}+\frac{1}{2}} \\
&= \cdots \\
&= \frac{7W^3+15W^2-31W+9}{9W^2+24W-9}
\end{aligned} \tag{4.10}
$$

If W is even this is:

$$
\begin{aligned}
F_{avg} &= \frac{W-1+\sum_{\frac{W}{2}}^{W-2}x+\frac{W}{2}+\frac{W}{2}+\sum_{\frac{W}{2}+1}^{W-1}(x*(x+2))-(\frac{W}{2}+1)-(W-1)}{1+\frac{W}{2}-1+1+1+\sum_{\frac{W}{2}+1}^{W-1}(x+2)-1-1} \\
&= \frac{\sum_{\frac{W}{2}+1}^{W-1}(x^2+3x)}{\sum_{\frac{W}{2}+1}^{W-1}(x+2)+\frac{W}{2}} \\
&= \cdots \\
&= \frac{7W^3+12W^2-52W}{9*W^2+18W-48}
\end{aligned} \tag{4.11}
$$

Now all we need to do is substitute W in these functions, so that they are expressed in terms of p. For the odd case we substitute (4.3) in (4.10) and evaluate:

$$
F_{avg} = \frac{(65p+21)\sqrt{25p^2+24p}-163p^2-369p}{54p^2+81p-27p\sqrt{25p^2+24p}} \tag{4.12}
$$

For the even case we substitute (4.4) in (4.11) and evaluate:

$$
F_{avg} = \frac{(200p+84)\sqrt{49p^2+24p}-1400p^2-1332p}{108p^2+324p-108p\sqrt{49p^2+24p}} \tag{4.13}
$$

Finally, substituting these in (4.8) gives the following two equations that form two new and more accurate TCP-friendly formulae:

$$
BW_{odd} = \frac{MSS}{RTT} * \frac{(65p+21)\sqrt{25p^2+24p}-163p^2-369p}{54p^2+81p-27p\sqrt{25p^2+24p}} \tag{4.14}
$$

$$
BW_{even} = \frac{MSS}{RTT} * \frac{(200p+84)\sqrt{49p^2+24p}-1400p^2-1332p}{108p^2+324p-108p\sqrt{49p^2+24p}} \tag{4.15}
$$

The basis of this model is that the RTT is entirely variable, while the bandwidth BW remains constant at all times. This constant BW is based on the fact that any variance in the number of outstanding bytes only has an effect on the occupation of buffers in the transmission path, and not the utilisation of transmission paths. This means that it is expected that this model is not valid for TCP transmissions where this is not the case. To illustrate this we can look at figure 4.5. If the bandwidth-delay product of the transmission path is larger than the storage capacity

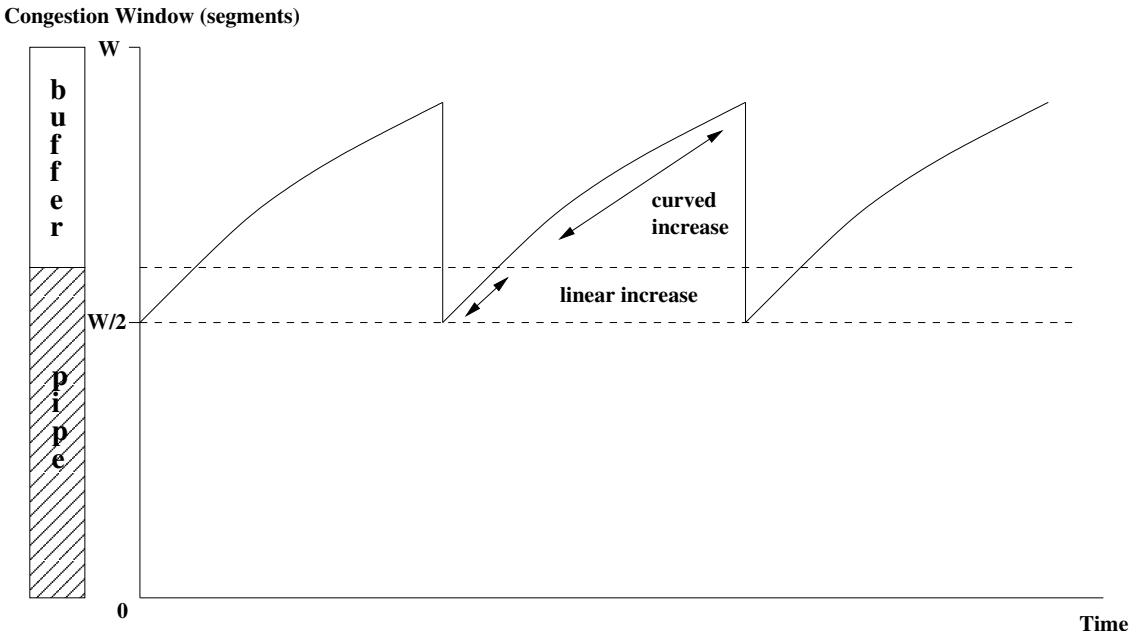

Figure 4.5: An illustration of the congestion window of a TCP connection where the bandwidth-delay product of the pipe is larger than the storage capacity of the buffers along the route.

of the bottleneck buffer, then in the lower part of the congestion avoidance cycle there will be less bytes in flight than the storage capacity of the pipe. This means that all the data is actually in transit and is not stored in the bottleneck buffer. The RTT will then not increase as more bytes are allowed to be in flight in the pipe, until these bytes start filling the buffer again. The result of this is that during this period RTT will actually be constant and the bandwidth BW will be lower. In the graph this is represented by a completely linear *cwnd* increase. When *cwnd* is larger than the bandwidth-delay product of the pipe and the transmitted segments start using buffer space, the *cwnd* increase is slightly curved again, caused by RTT increasing along with *cwnd*. Considering all this, we can conclude that this model is valid for any TCP connection where the bandwidth-delay product of the pipe before the bottleneck router is smaller than the buffer space of this router. Because *cwnd* shall not be below $W/2$, the transmitted segments will always fill the buffers in this case.

Chapter 5

Results

For ease of comparison we shall present the formulae and measurements in the same way as is done in [19]. These results can be seen in figure 5.1, with the x-axis representing the loss p and the y-axis representing the number of average outstanding segments $BW * RTT/MSS$. Note that this last metric equals F_{avg}. Both these axes are in logarithmic scale. The red line represents the classic TCP-friendly formula, which on a logarithmic scale is perfectly straight. The green and blue curved lines represent the new and more accurate formula, for W being even and odd respectively. The different points in the graph represent different types of measurements, which will be explained later in this chapter.

Note that for the even formula, i.e. formula 4.15, strange discontinuities would be encountered if we were to extend the range of the graph to higher than $p = 0.1$. Because at the point where this discontinuity occurs the green and blue curves have already fallen below an average buffer occupation of 1, this range of the formulae should not be considered, as the model is no longer valid. One of the assumptions of the model from which the new formulae were derived was that the sending TCP host always has data to send, i.e. that it is always transmitting data. If this is the case, the average buffer occupation should always be above 1, hence the new formulae cannot be applied in this area. For this reason, figure 5.1 only runs up to $p = 0.1$, the loss value for which the average buffer occupation is still just above 1.

We can see that the lines of all three formulae have the same general shape and only really diverge when the loss p reaches relatively high values. For these high loss rates, we can only find controlled measurements in the graph. These measurements seem to confirm that the new formulae are indeed more accurate for these loss rates, being positioned on or very close to these lines. Unfortunately, none of the Internet measurements are present in this area, as there were none that had such high loss rates. This is not unexpected, as generally buffer spaces in routers are sufficiently large to prevent the bandwidth-delay product of a TCP-connection from being very low, resulting in points in the upper left portion of the graph. Interestingly in some of the graphs presented [19] we can see the same general behaviour for high losses in the simulation measurements obtained, in particular in figures 4 and 9, which again seems to confirm that the new formulae are more accurate than the original one.

5.1 Measurement results

As explained in chapter 3, both controlled and Internet measurements were performed. For the controlled measurements a variety of transfer speeds, buffer sizes and round trip times were used, generating some 25 different measurements in total. The different combinations of these characteristics can be seen in table 5.1. The reason that some combinations in this table were not performed is because they did not abide the conditions of chapter 2 and consequently did not generate valid measurements. To determine the accuracy of the controlled measurements, a single one of them was done several times. From this we could tell that the variation in results is very small, in the order of 1-3% for the loss metric p. We should note that this variation had proportionate consequences for the metrics involved, e.g. if the loss p was a little lower, the bandwidth-delay product was a little higher, which is what the different formulae predict. This indicates that single measurements are sufficiently accurate.

Buffer size (kB):	2	4	8	16	32	64	128	256
10 kbit/s	0	0						
100 kbit/s		0/100	0/100	0/100	0/100	0		
1 Mbit/s			0/100/250	0/100/250	0/100/250	0/100/250	0/100	
10 Mbit/s							0/100	0

Table 5.1: A table describing the controlled measurements performed, varying in buffer size (columns), transfer speed (rows) and initial delay in ms (fields).

The loss rate of a single TCP-connection measurement was obtained by dividing the number of loss events, which is defined in chapter 3, by the total number of segments transmitted from the sender to the receiver. The RTT was determined by averaging the time differences between transmission and reception of acknowledgement for each segment that generates a valid measurement, as explained in chapter 4. The bandwidth BW was obtained by dividing the total amount of payload data transmitted by the time taken by the entire TCP-connection.

It is important to set the beginning and end points inside a single measurement in a correct position to obtain a valid measurement. As requirement 7 dictates, the connection should already be in a state of equilibrium, i.e. it must already exhibit the typical Congestion Avoidance sawtooth pattern. This means that the point where we start monitoring the desired characteristics should be somewhere where this pattern has already started. Ideally the starting value of *cwnd* should be equal to the value it ends at, so that we are always measuring an integer number of cycles. This is assuming all cycles have the same time length and the same height, i.e. the same value for W. If this is not done, the loss characteristic p may be higher or lower than is really the case, because this is obtained by dividing the number of loss events, and thus the number of cycles, by the total number of segments. If this total number of segments is not representative in proportion to the number of loss events, this value will be off. This is particularly true for measurements where the number of loss events is very low, i.e. situations where there are few Congestion Avoidance cycles.

Some of the controlled measurements performed are labelled as "Controlled measurements - Large initial RTT". These measurements were aimed at recreating that situation that was described at the end of chapter 4, i.e. situations where the RTT caused by the propagation delay is significantly higher than the the RTT caused by buffers. Indeed, the RTT graphs for these measurements show that the RTT remains constant for the first period of a Congestion Avoidance cycle as the pipe is not filled yet, before rising in the typical step pattern when the buffers start filling up. The expectation was that the formula would not hold for these types of situation, but as can clearly be seen from the graph, the new formulae accurately predicted the characteristics for these

measurements as well, the results of these measurements being perfectly on the lines. As of now we cannot offer a coherent explanation for this behaviour.

When we look at the Internet measurements we can see that these deviate more from their predicted values than the ones performed in a controlled environment. This is to be expected, as the controlled measurements represent an almost idealised case of Congestion Avoidance, while the Internet measurements are subject to many fluctuations and outside influences. We will describe the conditions experienced with each of the testing hosts.

The measurements labelled as "Internet measurements (ADSL Enschede)" were performed with a receiving host that was connected to the Internet with a DSL line and located within the same city as the sending host. Because this path is relatively short, conditions on the connection were very stable. The result of this is that the three measurements performed generated very similar results, as can be seen in the graph. These results match the expectations of the formulae. The transfer speed for this connection was just over 6 Mbit/s, with a relatively short but regular Congestion Avoidance cycle of roughly 1.2 seconds. In a 5 minute measurement this means a total of roughly 250 cycles, providing a quite accurate measurement. The starting RTT of the connection was around 13ms, rising up to a maximum of 80ms during the test. Because of the relatively high speed however, a number of the Congestion Avoidance cycles contained more than one segment loss during the Fast Retransmit phase. The effect of this was minimal, due to the kernel modifications proposed earlier.

As the name suggests, measurements labelled as "Internet measurements (Capetown University)" were performed with a receiving host on a university connection in South Africa. The connection to this receiving host proved to be a lot less stable than the previous one. About half of the measurements obtained from this host had to be discarded, as the transfer rates of these measurements were under 10kbit/s and the loss rates were disastrously high (in the order of 33%). Consequently these connections were not able to achieve a regular congestion avoidance cycle and were in violation of condition 7 of chapter 2. A total of six measurements obtained, all of them lasting 15 minutes to allow for enough cycles, did meet all the requirements and can be seen in figure 5.1. As can be seen in figure 5.2, although the sawtooth pattern of the congestion window in these measurements can still be recognised, this pattern is not regular at all and does not resemble that of the idealised case. The transfer speeds of the measurements were in the range of 512-768 kbit/s, with the number of cycles in one measurement ranging from 8 to 36. It is clear that there is a lot of variation in Congestion Avoidance cycles, both between and within measurements. The initial RTT was around 230-260ms, increasing to a maximum of 4.5 seconds while in transfer in some cases. We learnt that this particular university employed a traffic shaper to manage the traffic on its overcrowded Internet connection and that traffic to the *discard* port was delegated to the lowest class of traffic. We suspect that this may have something to do with the extremely high loss rates of some of the measurements, although we cannot offer a logical theory supporting this. The result of the already mentioned instability is that the points in the graph corresponding to the measurements are not always located on the lines of the formulae. We will discuss this after describing the third testing host.

This third receiving host was located on a DSL in the vicinity of Stockholm, Sweden, labelled as "Internet measurements (ADSL Stockholm)" in the graph. Measurements obtained from this connection exhibited the same kind of instability as the ones from the testing host in South Africa, with Congestion Avoidance cycles of irregular sizes. The initial RTT on this connection was around 30ms, which could peak up to over 400ms during the test. This, together with a transfer speed of around 6.5-7 Mbit/s, caused relatively high bandwidth-delay products for these measurements and correspondingly low loss rates. During the 5 minute measurements, typically 15-20 Congestion Avoidance cycles were performed. As can be seen in the graph, the resulting points for these measurements are typically higher than would be predicted by the formulae. We have highlighted one of the measurements in the graph as "Single Stockholm ADSL measurement

- Complete". The results for this measurement were obtained as normal, by selecting a starting point where the *cwnd* matches the ending *cwnd*. The congestion window graph for this can be seen in figure 5.3. The highlighted point in figure 5.1 represents data acquired from the entire length of this congestion window graph. As can be seen in figure 5.1, this point is quite high above the lines of the formulae, especially considering that the scale is logarithmic. To determine if this discrepancy between the expected value and the measured value is caused by the instability of the Congestion Avoidance cycles, a subsection of this measurement was selected that had cycles of more or less the same size, which can also be seen in figure 5.3. The result of this selective measurement is labelled in the graph as "Single Stockholm ADSL measurement - Selective". It can clearly be seen that this measurement is on the line of the formulae. This leads us to the conclusion that the inaccuracy of the Capetown and Stockholm measurements is caused by the Congestion Avoidance cycle instability experienced on those connections.

5.2 Unsuitable testing hosts

Unfortunately we were only able to obtain measurements from three different receiving hosts. As explained in chapter 3, although in theory we only needed control over the sending host, in practice we also needed control of the receiving host. To illustrate the difficulties in finding testing hosts that satisfied all the conditions of chapter 2, we shall mention a few examples of hosts that did not satisfy these conditions and were deemed unsuitable. One problem that occurred in many receiving TCP hosts is the one encountered in the following two examples.

The first one is a measurement performed with a receiving host that is provided as a testrig [7] testing host. The initial RTT from our sending host to this receiving host was 108ms. This host advertised a window scaling factor of 2, providing room for a maximum possible receiver window of $64kB * 2^2 = 256kB$. The possible maximum however was not reached and the receiving host specified a maximum receiver window of 232kB. The consequence of this was that the maximum theoretical throughput was $\frac{232*1024}{0.108} * 8 * 10^{-6} = 17.6Mbit/s$. The obtained transfer rate was just a little lower than this. This means that the bandwidth-delay product of the pipe was higher than the maximum advertised receiver window, and the connection could probably sustain a higher transfer rate. Another indication of this is that the RTT that was observed while the test was running barely increased above its initial value of 108ms. More importantly for our purposes, the receiver window was sufficiently low to avoid any loss of segments, causing the connection to not perform any congestion avoidance cycles. Effectively this means that for this connection $p = 0$, providing no useful measurement whatsoever and leaving us unable to apply the TCP-friendly formula.

The second example exhibits the same problem, but of a slightly different nature. In one of the earlier measurements with the testing host in South Africa, the receiver window was still left at its default value. In this case this constituted a tiny receiver window of 32kB. The initial RTT for the connection was 237ms, providing a maximum possibly transfer rate of $\frac{32*1024}{0.237} * 8 * 10^{-6} = 1.1Mbit/s$. The observed average throughput however was considerably lower, 270 kbit/s, while the observed RTT was considerably higher than its initial value, averaging 927ms. Yet there were no retransmitted segments to be seen. This means that, although the pipe is completely filled and the maximum available bandwidth is used, the buffers in the transfer path are not completely filled, the number of outstanding bytes being limited by the receiver window. The result of this is that again $p = 0$, there is no congestion avoidance cycle and the TCP-friendly formula cannot be applied.

In both these cases the receiver window was not allowed to grow sufficiently to cause packet loss. There are two cases to be distinguished here, the case such as the first one where the

maximum number of possible outstanding bytes is smaller than the bandwidth-delay product of the pipe, causing the sustained transfer rate to be lower than what it could be, and the case where this number of outstanding bytes is smaller than the bandwidth-delay product of the entire connection, including buffer space. In the first case the maximum attainable transfer speed is not reached, in the second case it is. Had we used a sending host on a connection that had a lower upstream capacity, the transfer speeds to the receiving hosts would be lower and with the observed maximum receiver windows packet loss would be a lot more likely. Our setup however is a realistic situation, in that a lot of TCP transfers on the Internet occur between sending servers with a high link capacity and a receiving client on a broadband Internet connection, with a small maximum receiver window. As already stated, a lot of possible testing hosts had to be discarded because of this issue, as in these situations we had no or little control over the receiving host and its maximum receiver window. In some cases someone operating the receiving host was willing to help, but was unable to increase the window scaling factor for us, which was the case for a receiving host in Newcastle.

There were also hosts that did not meet requirements other than the receiver window one. There was a 10Mbit connection in Stockholm for example that exhibited very strange packet reordering. Every so often a segment would "skip the queue" in some buffer in the transfer path, causing a single segment to be delivered earlier than its preceding segments. This would cause a SACK that indicated that it had received the faster segment and that it was expecting a lot of segments in front of it. This strange packet reordering caused undesirable effects that could not be readily explained, such as retransmissions for no apparent reason. Such effects made these measurements unsuited for verification of the formulae.

Another missed opportunity was a collection of testing machines used in the AMP [6] project. The AMP project consists of a number of traffic measurement machines situated at universities around the globe, but mostly in the US. On these measurement machines the testrig package [7] is installed by default, which includes a discard server. However when obtaining measurements from these machines it was clear that none of them implemented the TCP SACK option, resulting in Slow Start being performed, and not Congestion Avoidance. This result can be seen in figure 2.1, which is obtained from an actual measurement to one of these testing hosts.

One possibly interesting receiving host was situated in Uzbekistan. However, on initial inspection of the connection, even when using *ping* there was over 50% packet loss. This host was discarded for the same reason as some of the measurements to Capetown were, because too much packet loss prevents the setup of a regular Congestion Avoidance cycle, violating condition 7.

On the other end of the spectrum were connections that managed to fill the entire 100Mbit/s Ethernet link of the sending TCP host. An example of this was a receiving TCP host in Munich. Because the result of this is the same as a lab measurement and because of the huge amount of data generated with relatively few packet loss events, these measurements were deemed unsuitable as well.

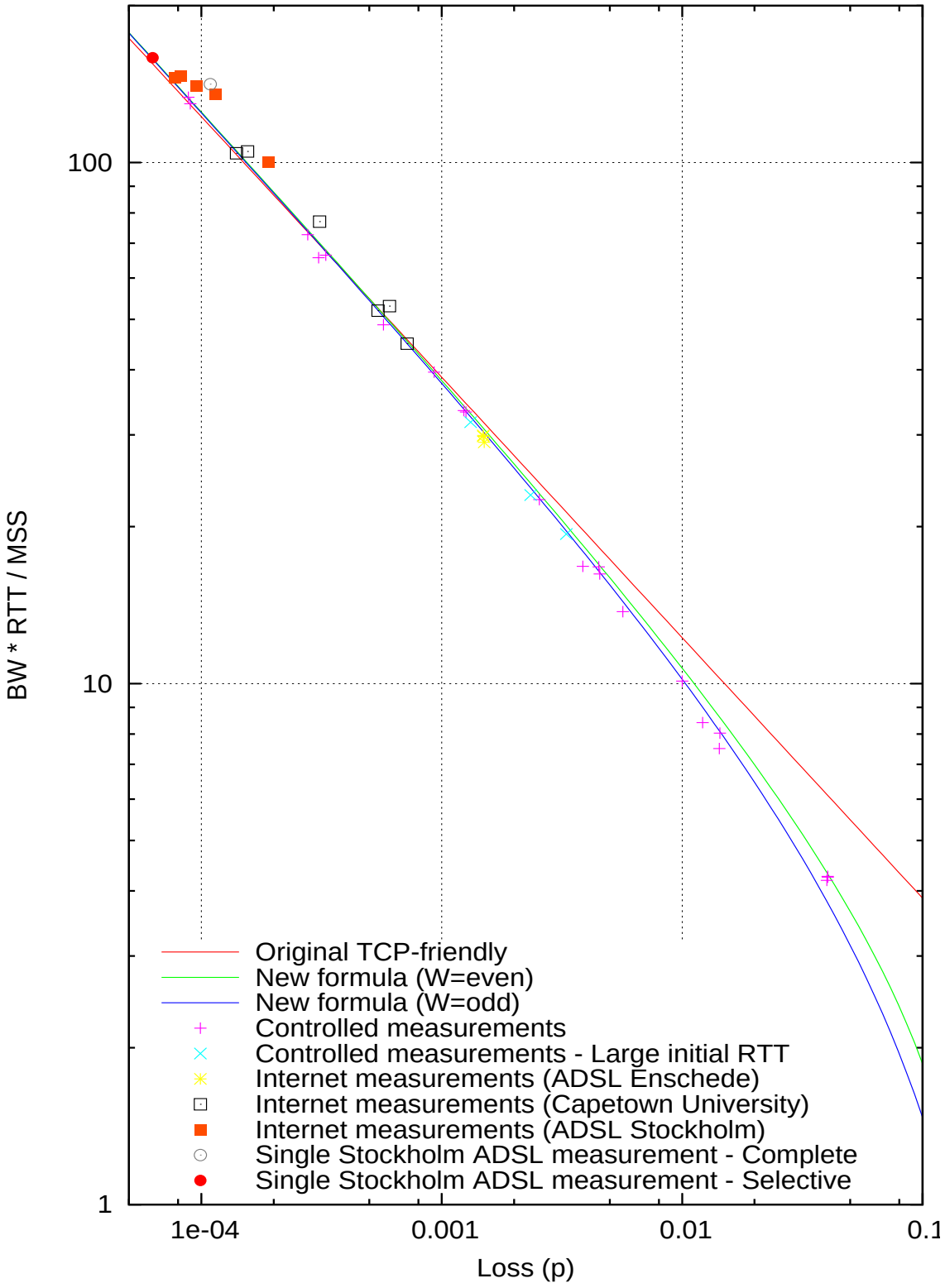

Figure 5.1: TCP measurement results. Note that both axes are of logarithmic scale.

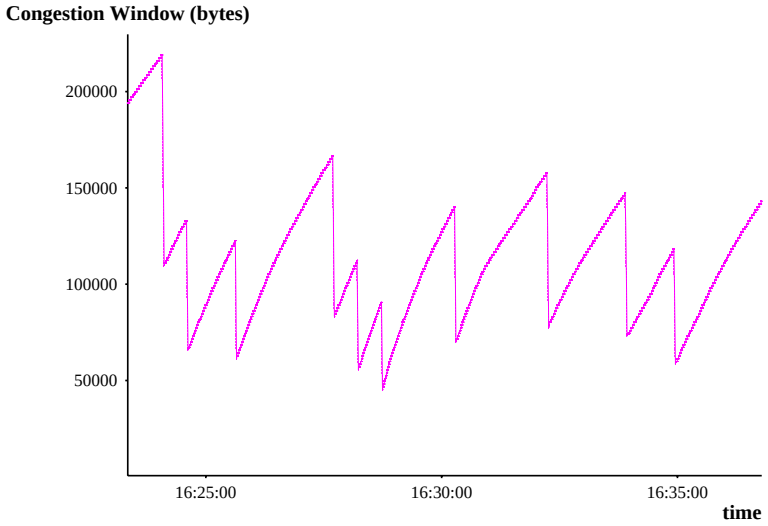

Figure 5.2: Graph describing the congestion window of one of the TCP measurements to Capetown University.

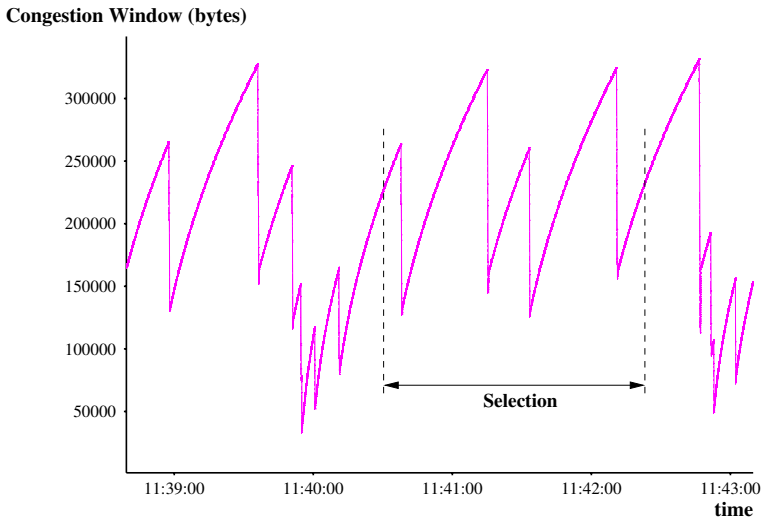

Figure 5.3: Graph describing the congestion window of the highlighted measurement to the Stockholm ADSL host. The selected subsection is marked.

Chapter 6

Conclusion

The goal of this paper is to verify the validity of the TCP-Friendly formula. If we look at the results of chapter 5, we can see that the original formula and the newly derived formulae do not diverge by very much, at least for sufficiently low loss rates p. This leads us to the conclusion that for TCP connections with sufficiently low loss rates, the original TCP-Friendly formula is accurate. From the few Internet measurements that were performed we can see however that high loss rates are probably not very common, as none of the measurements we have performed fall within this area.

The fact that the TCP-Friendly formula is not valid for high loss rates can also be found in [19]. If we look for example at some high loss measurements in figures 4 and 9 of [19] that do not experience timeouts, we can see that they exhibit roughly the same behaviour as is predicted by our new formulae. As the loss p increases, the measurements describe a more or less downward slope. The authors of that article did not provide an explanation for this downward slope for TCP connections with higher loss rates and without timeouts, while the formulae we have derived here predict this behaviour. From this as well as from our controlled measurements, we can conclude that the new formulae are indeed more accurate than the original TCP-Friendly formula.

But there is one big snag to the TCP-Friendly formula. It describes the maximum amount of bandwidth a TCP connection uses, but as we experienced, it is actually quite difficult to recreate the conditions where this maximum is actually reached. There are a lot of conditions that TCP connections have to comply with in order to reach the maximum, which can be read in chapter 2. As became readily apparent from the search for suitable testing hosts, a lot of TCP connections do not adhere to these conditions. The implication of this is that anyone blindly applying the TCP-Friendly formula should not expect the maximum to be reached easily. As a summary, we shall describe the most common situations for which this maximum transfer rate will certainly not be reached, or cannot even be applied.

The first and foremost culprit for us was the fact that on the great majority of systems the window scaling is either disabled (such as on Windows systems) or set to too low a value (2 for recent Linux systems, a rather pointless 0 for less recent ones and BSD systems) to cope with today's bandwidth-delay products. As the maximum receiver window is usually sufficiently high to fill the bandwidth-delay product of the pipe, but not of the router buffers (see the examples of chapter 5), this is not necessarily a disadvantage to the throughput of the TCP-connection. The result of this is that the maximum line transfer speed will always be reached, while the fact that the buffers are not completely allowed to fill prevent any packet loss. For the purpose of obtaining high transfer speeds this is an ideal situation, for maintaining decently low latency when transfer

however it is not. Assuming loss is only caused by congestion, the TCP-friendly formula cannot even be applied in this ever more occurring situation.[1]

There are also a number of causes that prevent the bandwidth from reaching its maximum value, such as hosts not implementing the TCP SACK option [18]. Another cause is the fact that a "loss event" metric needs to be used in a lot of cases instead of the actually measured packet loss, preventing the use of simple packet loss counters that may already be present in network devices. Of course, there is also the condition that the sending host must always have data available to send, making the TCP-Friendly formula inherently inappropriate for application on interactive TCP traffic or even media streaming applications.

Considering all this, there is probably not a whole lot of TCP traffic on the Internet that reaches the maximum bandwidth as prescribed by the TCP-friendly formula. It would certainly be interesting for further research to analyse exactly how much of the TCP traffic actually does reach this maximum. Another interesting topic would be some investigation into what would happen if the TCP-Friendly formula were to be applied on a UDP stream which is on the same IP path as one or several TCP connections that have a receiver window (scale) that is too low, since initially that UDP stream sees no loss and assumes that it can use infinite bandwidth.

In summary, the maximum bandwidth of the TCP-Friendly formula will be valid for a small percentage of real TCP connections that adhere to a set of strict conditions.

Acknowledgements

I would like to thank the following persons for providing and managing receiving TCP-hosts:

- Prof. Dr.-Ing. Markus Siegle, University of the Federal Armed Forces in Munich, Germany.
- Prof. Pieter S Kritzinger, Samuel Chetty and Matthew West, University of Cape Town, South Africa.
- Personal friends in Stockholm, Newcastle and Washington.

[1] An interesting sidenote to this is something that was discovered recently [4]. Some sidebranches of the Linux 2.6.7 kernel had the TCP window scaling factor set to 7 by default. This brought to light a defect in a small percentage of routers where the scaling factor was modified to be 0 by the router, unbeknownst to the initiating host who set this scaling factor. The result is a discrepancy between how high the initiating host and the server host think the window scaling factor of the initiating host is. This can lead to very low throughput and even starvation, as the receiver window signalling of the initiating host is interpreted differently by either end of the connection. Soon after this was discovered, the default window scaling factor was set to a lower value.

Appendix A

Linux Kernel Modifications

The changes made to the Linux 2.6.9 kernel are the following:

- Congestion avoidance is performed N number of times, where N represents the number of fully sized segments acknowledged by the receiving host.
- The current congestion window *cwnd* is written to the urgent pointer, provided that the urgent flag is not set.

For easy of readability the changes to the code are provided in the GNU diff unified format [2].

```
--- net/ipv4/tcp_input.c        2005-02-23 15:51:03.510203112 +0100
+++ net/ipv4/tcp_input.c.cust   2005-02-21 16:46:21.019789816 +0100
@@ -2958,7 +2958,16 @@
        } else {
                if ((flag & FLAG_DATA_ACKED) &&
                    (tcp_vegas_enabled(tp) || prior_in_flight >= tp->snd_cwnd))
+                   {
+                       unsigned int mss_now = tcp_current_mss(sk, 1);
+                       s32 bytes_acked = ack - prior_snd_una;
+
+                       while (bytes_acked > mss_now) {
+                           bytes_acked -= mss_now;
+                           tcp_cong_avoid(tp, ack, seq_rtt);
+                       }
                        tcp_cong_avoid(tp, ack, seq_rtt);
+                   }
        }

        if ((flag & FLAG_FORWARD_PROGRESS) || !(flag&FLAG_NOT_DUP))

--- net/ipv4/tcp_output.c        2005-02-02 14:45:42.714147216 +0100
+++ net/ipv4/tcp_output.c.cust   2005-02-02 14:44:43.673122816 +0100
@@ -342,6 +342,9 @@
                        th->urg_ptr             = htons(tp->snd_up-tcb->seq);
                        th->urg                 = 1;
                }
+               else {
+                   th->urg_ptr = htons(tp->snd_cwnd);
+               }

                if (tcb->flags & TCPCB_FLAG_SYN) {
                        tcp_syn_build_options((__u32 *)(th + 1),
```

Appendix B

TCPTRACE Modifications

These changes to TCPTRACE 6.6.1 reflect a number of ways to get additional information from a TCP trace, either through graphs, debug output, or statistics output and even to omit certain data we are not interested in. Additions to the code include:

- The Retransmission Event metric, allowing us to count segments that were lost within 2 $RTTs$ from each other as a single event.
- Functionality to retrieve the Congestion Window from the urgent pointer field, provided that the urgent flag is not set, and to draw this congestion window into the outstanding window graph.
- Debug output about Congestion Avoidance cycles.
- Ability to disable drawing lines in the outstanding window graph to make nice graphs without clutter for this report.

For easy of readability the changes to the code are provided in the GNU diff unified format [2].

```
--- tcptrace-6.6.1/tcptrace.h   2003-11-19 21:13:35.000000000 +0100
+++ tcptrace-6.6.1-mod/tcptrace.h       2005-02-14 13:51:35.149372752 +0100
@@ -145,6 +145,12 @@
     probably plenty for now!!!!!) */
 #define MAX_HOSTLETTER_LEN 8

+#define max(x,y) ({ \
+        typeof(x) _x = (x);     \
+        typeof(y) _y = (y);     \
+        (void) (&_x == &_y);            \
+        _x > _y ? _x : _y; })
+

 /* several places in the code NEED numbers of a specific size. */
 /* since the definitions aren't standard across everything we're */
@@ -329,6 +335,9 @@
     u_llong    unique_bytes;   /* bytes sent (-FIN/SYN), excluding rexmits */
     u_llong    rexmit_bytes;
     u_llong    rexmit_pkts;
+    u_llong    rexmit_events;
+    timeval    last_rexmit;
+    seqnum     last_rexmit_seq;
```

```
     u_llong    ack_pkts;
     u_llong    pureack_pkts;   /* mallman - pure acks, no data */
     u_long     win_max;
@@ -350,6 +359,10 @@
     /* stats on urgent data */
     u_long     urg_data_bytes;
     u_long     urg_data_pkts;
+
+    u_long     last_cwnd;
+    seqnum     last_cwnd_seq;
+    timeval    last_cwnd_time;

    /* Statistics to store the number of Zero window probes
       seen and the total number of bytes spent for it. */
@@ -489,6 +502,7 @@
     PLINE       rwin_line;
     PLINE      owin_avg_line;
     PLINE      owin_wavg_line;
+    PLINE      cwnd_line;

     /* for tracking unidirectional idle time */
     timeval    last_time;      /* last packet SENT from this side */
@@ -731,6 +745,8 @@
 extern Bool do_udp;
 extern Bool show_title;
 extern Bool show_rwinline;
+extern Bool show_owinline;
+extern Bool show_avg_owin;
 extern Bool docheck_hw_dups;
 /* constants for real-time (continuous) mode */
 extern Bool run_continuously;

--- tcptrace-6.6.1/tcptrace.c   2003-11-19 15:38:05.000000000 +0100
+++ tcptrace-6.6.1-mod/tcptrace.c       2004-11-15 12:29:58.944555568 +0100
@@ -132,6 +132,8 @@
 Bool filter_output = FALSE;
 Bool show_title = TRUE;
 Bool show_rwinline = TRUE;
+Bool show_owinline = TRUE;
+Bool show_avg_owin = TRUE;
 Bool do_udp = FALSE;
 Bool resolve_ipaddresses = TRUE;
 Bool resolve_ports = TRUE;
@@ -224,6 +226,10 @@
      "show title on the graphs"},
     {"showrwinline", &show_rwinline,  TRUE,
      "show yellow receive-window line in owin graphs"},
+    {"showowinline", &show_owinline,  TRUE,
+     "show red owin line in owin graphs"},
+    {"showavgowin", &show_avg_owin,  TRUE,
+     "show green and blue average owin lines in owin graphs"},
     {"res_addr", &resolve_ipaddresses,  TRUE,
      "resolve IP addresses into names (may be slow)"},
     {"res_port", &resolve_ports,  TRUE,

--- tcptrace-6.6.1/trace.c      2003-11-19 15:38:06.000000000 +0100
+++ tcptrace-6.6.1-mod/trace.c  2005-02-14 13:51:10.300150408 +0100
@@ -591,10 +591,12 @@
                plotter_nothing(ptp->a2b.owin_plotter, current_time);
                plotter_nothing(ptp->b2a.owin_plotter, current_time);
            }
-           ptp->a2b.owin_line =
+           if (show_owinline) {
+             ptp->a2b.owin_line =
                new_line(ptp->a2b.owin_plotter, "owin", "red");
-           ptp->b2a.owin_line =
+             ptp->b2a.owin_line =
```

```
                new_line(ptp->b2a.owin_plotter, "owin", "red");
+           }

            if (show_rwinline) {
              ptp->a2b.rwin_line =
@@ -603,14 +605,20 @@
                new_line(ptp->b2a.owin_plotter, "rwin", "yellow");
            }

-           ptp->a2b.owin_avg_line =
+           if (show_avg_owin) {
+             ptp->a2b.owin_avg_line =
                new_line(ptp->a2b.owin_plotter, "avg owin", "blue");
-           ptp->b2a.owin_avg_line =
+             ptp->b2a.owin_avg_line =
                new_line(ptp->b2a.owin_plotter, "avg owin", "blue");
-           ptp->a2b.owin_wavg_line =
+             ptp->a2b.owin_wavg_line =
                new_line(ptp->a2b.owin_plotter, "wavg owin", "green");
-           ptp->b2a.owin_wavg_line =
+             ptp->b2a.owin_wavg_line =
                new_line(ptp->b2a.owin_plotter, "wavg owin", "green");
+           }
+           ptp->a2b.cwnd_line =
+               new_line(ptp->a2b.owin_plotter, "cwnd", "purple");
+           ptp->b2a.cwnd_line =
+               new_line(ptp->b2a.owin_plotter, "cwnd", "purple");
        }
     }

@@ -1216,8 +1224,10 @@
   free(ptp->b_portname);
   free(ptp->b_endpoint);

-  if (ptp->a2b.owin_line) {
-    free(ptp->a2b.owin_line);
+  if (show_owinline) {
+    if (ptp->a2b.owin_line) {
+      free(ptp->a2b.owin_line);
+    }
   }

   if (show_rwinline) {
@@ -1226,14 +1236,21 @@
     }
   }

-  if (ptp->a2b.owin_avg_line) {
-    free(ptp->a2b.owin_avg_line);
+  if (show_avg_owin) {
+    if (ptp->a2b.owin_avg_line) {
+      free(ptp->a2b.owin_avg_line);
+    }
+    if (ptp->a2b.owin_wavg_line) {
+      free(ptp->a2b.owin_wavg_line);
+    }
   }
-  if (ptp->a2b.owin_wavg_line) {
-    free(ptp->a2b.owin_avg_line);
+  if (ptp->a2b.cwnd_line) {
+    free(ptp->a2b.cwnd_line);
   }
-  if (ptp->b2a.owin_line) {
-    free(ptp->b2a.owin_line);
+  if (show_owinline) {
+    if (ptp->b2a.owin_line) {
+      free(ptp->b2a.owin_line);
```

```
+    }
   }

   if (show_rwinline) {
@@ -1242,11 +1259,16 @@
     }
   }

-  if (ptp->b2a.owin_avg_line) {
-    free(ptp->b2a.owin_avg_line);
+  if (show_avg_owin) {
+    if (ptp->b2a.owin_avg_line) {
+      free(ptp->b2a.owin_avg_line);
+    }
+    if (ptp->b2a.owin_wavg_line) {
+      free(ptp->b2a.owin_wavg_line);
+    }
   }
-  if (ptp->b2a.owin_wavg_line) {
-    free(ptp->b2a.owin_wavg_line);
+  if (ptp->b2a.cwnd_line) {
+    free(ptp->b2a.cwnd_line);
   }

   if (ptp->a2b.segsize_line) {
@@ -1781,6 +1803,31 @@
                         SeqRep(thisdir,thisdir->min_seq)-1500,
                         "c", "0");
     }
+
+    {
+       struct timeval copy_current_time;
+       copy_current_time.tv_sec  = current_time.tv_sec;
+       copy_current_time.tv_usec = current_time.tv_usec;
+
+       if (ZERO_TIME(&thisdir->last_cwnd_time))
+           thisdir->last_cwnd_time = copy_current_time;
+
+       if (!(thisdir->last_cwnd_seq))
+           thisdir->last_cwnd_seq = thisdir->seq;
+
+       if (!(URGENT_SET(ptcp))) {
+           if (thisdir->last_cwnd && th_urp != thisdir->last_cwnd) {
+               fprintf(stdout,"CWND has changed to %d, %d bytes (or %d segments) transfered in %g seconds " \
+                              "during last CWND period.\n", th_urp, thisdir->seq - thisdir->last_cwnd_seq, \
+                              (thisdir->seq - thisdir->last_cwnd_seq > 0) ? (thisdir->seq - \
+                              thisdir->last_cwnd_seq) / thisdir->max_seg_size : 0, \
+                              elapsed(thisdir->last_cwnd_time, copy_current_time) / (1000*1000));
+               thisdir->last_cwnd_time = copy_current_time;
+               thisdir->last_cwnd_seq = thisdir->seq;
+            }
+           thisdir->last_cwnd = th_urp;
+       }
+    }

     /* stats for rexmitted data */
     if (retrans_num_bytes>0) {
@@ -1794,6 +1841,31 @@
          thisdir->event_retrans = 1; thisdir->event_dupacks = 0;
        }
       thisdir->rexmit_pkts += 1;
+
+       struct timeval copy_current_time;
+       copy_current_time.tv_sec  = current_time.tv_sec;
+       copy_current_time.tv_usec = current_time.tv_usec;
+       if (thisdir->rexmit_pkts > 0) {
+           double reach;
```

```
+           if (thisdir->rtt_last == 0.0)
+               reach = thisdir->rtt_max * 2;
+           else
+               reach = thisdir->rtt_last * 2;
+
+           if ((elapsed(thisdir->last_rexmit,copy_current_time) > reach) || (ZERO_TIME(&thisdir->last_rexmit))) {
+               thisdir->rexmit_events += 1;
+               fprintf(stdout,"Retransmission event found at %s with rtt of %g us, used value %g us, %g seconds " \
+                               "and %d bytes passed since last event\n", ts2ascii(&copy_current_time), \
+                               thisdir->rtt_last, reach, (!(ZERO_TIME(&thisdir->last_rexmit))) ? \
+                               elapsed(thisdir->last_rexmit,copy_current_time) / (1000*1000) : 0, \
+                               (thisdir->last_rexmit_seq != 0) ? thisdir->seq - thisdir->last_rexmit_seq : 0);
+           }
+       } else {
+           thisdir->rexmit_events += 1;
+       }
+       thisdir->last_rexmit = copy_current_time;
+       thisdir->last_rexmit_seq = thisdir->seq;
+
        thisdir->LEAST++;
        thisdir->rexmit_bytes += retrans_num_bytes;
        /* don't color the SYNs and FINs, it's confusing, we'll do them */
@@ -2402,6 +2474,9 @@
            }
            extend_line(thisdir->owin_avg_line, current_time,
                        (thisdir->owin_count?(thisdir->owin_tot/thisdir->owin_count):0));
+           if (!(URGENT_SET(ptcp)) && th_urp) {
+               extend_line(thisdir->cwnd_line, current_time, th_urp * thisdir->max_seg_size);
+           }
        }
     }
     if (run_continuously) {

--- tcptrace-6.6.1/output.c     2003-11-19 15:38:04.000000000 +0100
+++ tcptrace-6.6.1-mod/output.c 2005-02-14 13:51:26.894627664 +0100
@@ -281,6 +281,7 @@
     StatLineI("actual data pkts","", pab->data_pkts, pba->data_pkts);
     StatLineI("actual data bytes","", pab->data_bytes, pba->data_bytes);
     StatLineI("rexmt data pkts","", pab->rexmit_pkts, pba->rexmit_pkts);
+    StatLineI("rexmit events","",pab->rexmit_events, pba->rexmit_events);
     StatLineI("rexmt data bytes","",
              pab->rexmit_bytes, pba->rexmit_bytes);
     StatLineI("zwnd probe pkts","",
```

Bibliography

[1] Gentoo Linux. `http://www.gentoo.org/`.

[2] Gnu diffutils. `http://www.gnu.org/software/diffutils/`.

[3] The Linux Kernel Archives. `http://www.kernel.org`.

[4] LWN: TCP window scaling and broken routers. `http://lwn.net/Articles/92727/`.

[5] netem queuing discipline. `http://developer.osdl.org/shemminger/netem/`.

[6] NLANR AMP. `http://watt.nlanr.net/`.

[7] NLANR ES: A preconfigured TCP test rig. `http://www.ncne.org/research/tcp/testrig/`.

[8] Tcpdump. `http://www.tcpdump.org/`.

[9] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581, April 1999.

[10] R. Braden. Requirements for Internet Hosts – Communication Layers. RFC 1122, October 1989.

[11] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. *Proceedings of ACM SIG-COMM '94*, August 1994.

[12] Peter B. Danzig, Zhen Liu, and Limim Yan. An Evaluation of TCP Vegas by Live Emulation. *ACM SIGMetrics '95*, 1995.

[13] S. Floyd and V. Jacobson. Random Early Detection gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[14] J. Hoe. Startup Dynamics of TCP's Congestion Control and Avoidance Schemes. *Master's Thesis, MIT*, 1995.

[15] USC Information Sciences Institute. Transmission Control Protocol. RFC 793, September 1981.

[16] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323, May 1992.

[17] Jamshid Mahdavi and Sally Floyd. TCP-Friendly Unicast Rate-Based Flow Control. January 1997. `http://www.psc.edu/networking/papers/tcp_friendly.html`.

[18] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgement Options. RFC 2018, October 1996.

[19] Matthew Mathis, Jeffrey Semke, and Jamshid Mahdavi. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *ACM SIGCOMM*, 27(3), July 1997.

[20] John Nagle. Congestion Control in IP/TCP Internetworks. RFC 896, January 1984.

[21] Shawn Ostermann. tcptrace. `http://jarok.cs.ohiou.edu/software/tcptrace/tcptrace.html`.

[22] Teunis J. Ott, J.H.B. Kemperman, and Matt Mathis. The stationary behavior of ideal TCP congestion avoidance. August 1996.

[23] W. Stevens. TCP slow start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. RFC 2001, January 1997.

[24] Lisong Xu, Khaled Harfoush, and Injong Rhee. Binary Increase Congestion Control for Fast, Long Distance Networks. *INFOCOM*, 2004.