How to identify the speed limiting factor of a TCP flow

Bachelor's Thesis

Mark Timmer

Enschede, September 19, 2005

The August 23 version of this bachelor's thesis has been approved by

dr.ir. P.T. de Boer
dr.ir. A. Pras

Everything that is really great and inspiring is created by the individual who can labor in freedom.

- Albert Einstein

Abstract

This thesis develops a method for identifying the speed limiting factor of a TCP flow. Five factors are considered: the receive window, the send buffer, the network and two kinds of application layer factors. Criteria for recognizing each factor based on TCP header information are put forward. These criteria result in percentages that can be used as a measure of the impact of each of the factors on a flow. As we will see, different limitation percentages need a different interpretation, which is discussed at the end of this report.

Contents

1 Introduction **1**

1.1 Motivation for this research 1

1.2 Research questions 1

1.3 Approach 1

1.4 Structure of this thesis 1

1.5 Intended audience 2

2 Basics of TCP **3**

2.1 Introduction 3

2.2 Properties of TCP 3

2.3 Speed limitations of a TCP flow 4

2.4 Visualization of TCP flows 5

3 The Repository: General Aspects, Problems and Usage **7**

3.1 Introduction 7

3.2 General aspects of the repository 8

3.3 Identifying individual TCP connections 8

3.4 Problems with using the repository 9

3.5 Impact of the measurement location 12

4 Recognizing the Receive Window Limitation **15**

4.1 Introduction 15

4.2 A receive window limited flow 15

4.3 Calculating the number of outstanding bytes 16

4.4 Relating outstanding data to the receive window 19

4.5 Framework for performing receive window checks 22

4.6 Different kinds of receive window limitations 23

4.7 Identifying the cause of a receive window limitation 29

5 **Recognizing the Send Buffer Limitation** **31**
5.1 Introduction 31
5.2 A send buffer limited flow 31
5.3 Detecting the send buffer limitation 31
5.4 Estimating the send buffer size 32
5.5 Dealing with send buffer size estimation updates 33

6 **Recognizing the Network Limitation** **39**
6.1 Introduction 39
6.2 TCP Friendly formula 39
6.3 Estimating the loss rate 40
6.4 Estimating the round-trip time 42
6.5 Putting it all together 44

7 **Recognizing the Application Layer Limitation** **47**
7.1 Introduction 47
7.2 Lack of data 47
7.3 Application layer acknowledgments or requests 49

8 **Conclusions:**
Interpreting the Results **51**
8.1 Introduction 51
8.2 Assumptions 52
8.3 Interpretation of the limitation percentages 52
8.4 Prioritizing the limitations 54
8.5 Processing the repository 54
8.6 Further work 55

A **Proof for Section 3.4.2** **59**

Bibliography **61**

Acknowledgments **63**

List of Figures **65**

List of Abbreviations **67**

Index **69**

Chapter 1

Introduction

1.1 Motivation for this research

A few years ago computer networks were slow and only capable of transmitting small amounts of textual data. Technology has changed, and today entire movies can be sent over the internet. The available bandwidth continues to be upgraded, in order to be able to deliver even better services in the future. However, questions have been raised about whether or not increasing the bandwidth of a network will remain profitable. Some initial findings of the chair for *Design and Analysis of Communication Systems* at the University of Twente have indicated that the network is often *not* the speed limiting factor of a data flow. In order to support the answering of this question, we've performed research on how to identify the speed limiting factor of a TCP flow. This research can later on be used to address the network limitation issue mentioned, but isn't restricted to this; if the network is *not* the speed limiting factor, information about what *is* can also be gathered.

1.2 Research questions

Our main research question is: "How to identify the speed limiting factor of a TCP flow?". This question can be subcategorized into the following two questions:

- What factors can limit a TCP flow?
- How to recognize each of these factors?

1.3 Approach

This report will attempt to address these issues by examining a large amount of TCP (Transmission Control Protocol) data, collected from the Surfnet 5 network [SUR05]. Five limiting factors will be identified: the receive window, the send buffer, the network and two application layer factors. Criteria will be developed to support an automatic categorization of connections; the categories correspond with the limiting factors.

1.4 Structure of this thesis

Chapter 2 will give an overview of TCP, although some prior knowledge is advisable. Chapter 3 will continue with an explanation of the used repository and its flaws. Coverage of the criteria for detecting the receive window limitation, the send buffer limitation, the network limitation and the application limitation will be provided in Chapter 4, Chapter 5, Chapter 6 and Chapter 7, respectively.

Each limitation detection will result in a percentage, as defined in the Chapters 4 through 7. Finally, the interpretation of these percentages is discussed in Chapter 8.

1.5 Intended audience

This thesis has been written for an average university student who has had an introductory course in computer networking. Knowledge about the TCP protocol is required, but for completeness, some important properties of TCP are still mentioned in Section 2.2. All argumentation is performed in fairly small steps, such that someone without any knowledge about speed limiting factors of TCP flows should be able to follow the method of reasoning.

Chapter 2

Basics of TCP

2.1 Introduction

Since we look at ways to identify limiting factors for the speed of a TCP (Transmission Control Protocol) flow, it is important to understand the basics of TCP. Without proper knowledge of the static rules of the protocol, it is not possible to gain insight in the interesting dynamics.

TCP is a transport layer protocol, situated between the network layer and the application layer in the internet protocol stack, as illustrated in Figure 2.1. It relies on the network protocol IP.

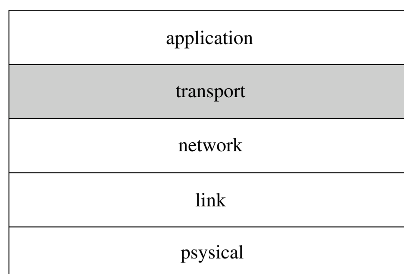


Figure 2.1: *Internet Protocol Stack*

Where IP is responsible for providing connectivity between two hosts, TCP provides transport functionality between applications. All the data coming from different application protocols in the application layer is gathered and put in TCP packets. Each packet receives a tuple with a source and destination port and is sent using IP (*multiplexing*). When a packet arrives at a TCP protocol entity, the port numbers are used to determine the application waiting for this packet and it is passed to the appropriate application at the application layer (*demultiplexing*).

Besides providing the multiplexing service, TCP also provides reliability, flow control and congestion control. Section 2.2 will address these three aspects of TCP, because they are very important for identifying packet flow speed limitations. This identification will be covered in Section 2.3. Finally, Section 2.4 will discuss the semantics of the TCP visualizations that will be used in the remainder of this report.

For more detailed information about the TCP protocol, see its RFC [Pos81].

2.2 Properties of TCP

2.2.1 Reliability

In order to be able to use the internet for fault-intolerant applications (FTP, e-mail, etcetera), TCP provides a mechanism for reliable data transfer. Before data is exchanged between two hosts, a connection is set up. Because of this connection-oriented property, state information can be saved.

The data to be sent is broken down into relatively small packets. Each packet has a sequence number and is transmitted over the network. After receiving a packet, the receiver sends back an acknowledgment packet to inform the sender of the correct arrival of the packet. This way, the sender can keep track of which bytes have arrived at the receiver and which have not. When an acknowledgment does not arrive in time (because of failure or possibly a very slow connection), a timer will fire and a retransmission will be performed.

When the receiver receives incorrect data (this is checked based on a checksum) or a packet that it did not expect (the sequence number is too high), it will send a duplicate acknowledgment for the last packet it correctly received. By these duplicate acknowledgments the sender can conclude

something is wrong and it can retransmit certain packets even before the timer would have fired (most implementations retransmit after receiving the third duplicate acknowledgment).

2.2.2 Flow control

Besides providing reliability, TCP also promises something to the receiver: flow control. Flow control means the sender will not overflow the receiver with amounts of data it cannot process. To achieve this, the receiver advertises a value called the *receive window* (*rwnd*) in its packets. The sender is obligated to restrict the number of outstanding bytes to this value. Therefore, by keeping the receive window low, the receiver can prevent the sender from sending a lot of data very quickly.

2.2.3 Congestion control

A third promise TCP makes is that it will not overflow the network. In October of 1986, the internet suffered from severe congestion for the first time [Jac88]. When there is congestion in a network, this means the routers are overburdened. One or more hosts send data with a speed the links cannot handle, so the buffers of the routers will overflow. This results in high delays and packet loss. Because of the reliability property of TCP, in that case retransmissions will be sent.

It seems clear that congestion is something we like to prevent from happening. The TCP mechanism for accomplishing this is called *congestion control*. A *congestion window* (*cwnd*) is maintained, indicating how many outstanding bytes are allowed. Because the receive window also indicates this, the minimum of *rwnd* and *cwnd* will be used as an upper bound on the number of outstanding bytes.

Initially, *cwnd* is equal to the maximum size of one packet (the MSS). After all, we do not know yet how many bytes the network can handle. When the connection starts, the slow start phase is entered. In this phase, *cwnd* will approximately double every round-trip time (RTT). The arrival of an acknowledgment for the first packet doubles *cwnd* to two times the MSS, the arrival of the two acknowledgments for these packets will make it four, and so on.

After a while, packet loss may be detected. The congestion window will be set back to the MSS and another quantity *threshold* will be set to $\frac{cwnd}{2}$. The process starts again, but will enter a new phase once $cwnd \geq threshold$: congestion avoidance. From now on *cwnd* will not double each RTT, but will grow linearly by one packet each RTT. Because congestion control directly relates the speed of sending packets into the network to packet loss and, therefore, to the speed at which the network can handle the packets, it reduces the amount of loss significantly. For a more in-depth coverage of TCP congestion control, see [Ste93].

2.3 Speed limitations of a TCP flow

Figure 2.2 gives an abstract overview of a packet being transmitted over TCP. When information is exchanged between two hosts, it looks as if the information is exchanged directly between the two application layers (indicated by the dashed arrow). However, in reality TCP adds a header and uses the underlying network for transportation. When we look at the level of TCP, several factors can be identified for limiting the speed of the connection.

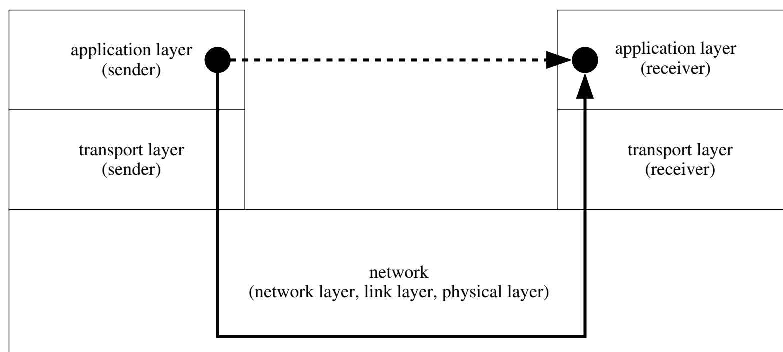


Figure 2.2: *A packet transmitted over TCP*

The receive window TCP exchanges data between the sending TCP entity and the receiving TCP entity. The receiving TCP entity has one mechanism to slow down a TCP connection: the receive window (see Subsection 2.2.2). When the receive window is small, only a few packets can be in flight before new packets are sent. When more packets could have been sent, the connection would probably be faster. The receive window could therefore limit the speed of a TCP connection.

It should be noted that there are two causes of a receive window limitation: either the application just cannot handle the data any faster (in that case the receive window is doing its job as it is supposed to) or the operating system uses a maximum value for the receive window that in practice is not large enough (in that case changing the implementation of the operating system would improve the speed of TCP connections).

The send buffer While a number of packets are in flight, the sender keeps a copy of these packets in its send buffer. After all, when a packet gets lost, it has to be able to perform a retransmission (see Subsection 2.2.1). The send buffer will of course have a limited amount of space, determined by the operating system. Once all the space is occupied by outstanding packets, the sending TCP entity has to stop accepting data from the application layer above it, until one or more acknowledgments have arrived. The send buffer could therefore limit the speed of a TCP connection.

The network TCP uses the underlying network for transportation, so when this transportation is slow, the connection will be slow. The network could therefore limit the speed of a TCP connection.

The application TCP relies on data it gets from the application layer. In case no new data is provided to TCP, even though it could send some, the connection cannot go any faster. The application could therefore limit the speed of a TCP connection. Two different application layer limitations can be identified: sometimes the application layer just does not provide data fast enough (resulting in periods with zero outstanding bytes) and sometimes the application appears to have its own kind of acknowledgment or request mechanism (resulting in a limited number of outstanding bytes the application layer will provide to the TCP layer).

It should be noted that only the application layer entity on the sending side is considered a limiting factor, because the receiving application layer entity does not have any direct influence on TCP (however, it can have influence on the sending application layer entity and thereby influence TCP indirectly).

2.4 Visualization of TCP flows

In this report several diagrams of TCP flows will be used, to illustrate certain scenarios. Each diagram will have the same semantics and contains several elements, which are all illustrated in the example of Figure 2.3.

An arrow from A to B indicates a packet with source address A and destination address B, and vice versa. The caption of the arrows indicates the type of packet (data or ack). All the examples are about half-duplex packet flows from A to B (see Section 3.3), so there will be no data contained in the acknowledgment packets from B to A.

In case of a data packet, the quantities *seq* and *length* can be displayed. The value of *seq* indicates the sequence number of the data packet and *length* gives its length (excluding the headers). Sometimes, when this is more appropriate for an example, the quantity *nextseq* is displayed instead. This value describes the sequence number of the next data byte that is to be sent, so in fact it is the sum of *seq* and *length*.

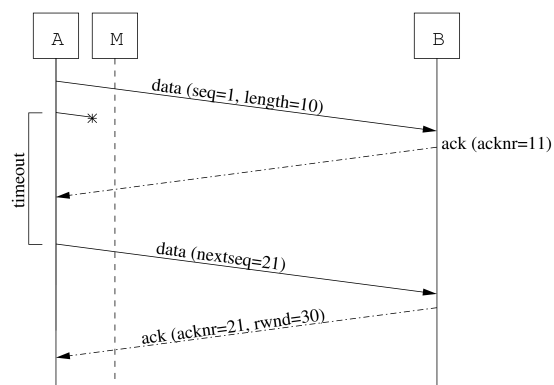


Figure 2.3: *Example TCP flow*

In case of an acknowledgment packet, the acknowledgment number (*acknr*) is shown. This number indicates the next byte to be received, so it acknowledges all bytes with sequence numbers up to the

acknowledgment number minus one. Sometimes the value of the receive window (*rwnd*) will be shown as well, as is the case for the second acknowledgment packet.

A lost packet is indicated by an asterisk on the arrow, as is the case for the first data packet. The figure also shows how a timeout is visualized.

Sometimes the caption of an arrow will be placed above the arrow (as is the case for most arrows in the example) and sometimes — for lay-out purposes — it will be shown at the beginning of the arrow (as is the case for the first acknowledgment packet of the example).

When relevant, the measurement location is indicated by a third square (labeled M) and a dotted line.

Chapter 3

The Repository: General Aspects, Problems and Usage

3.1 Introduction

To gain insight in the limiting factors of a TCP flow and to test the methods developed in this study, an amount of network data was necessary. Instead of collecting the data ourselves, we have used the repository created by Remco van de Meert in 2003 [vdM03]. The repository was created as part of the Measuring, Modelling and Cost Allocation (M2C) project, aimed at understanding the characteristics of network traffic in the internet. Figure 3.1 gives an architectural overview of the measurement setting [vdM03]. The repository has been created in parts of fifteen minutes which are not adjacent. Therefore, multiple flows have only been recorded partially.

Data was gathered at four locations, which are not named explicitly. We have chosen to examine the first one, which is described in [vdM03] as follows;

> *On location #1 the 300 Mbit/s (a trunk of 3 x 100 Mbit/s) Ethernet link has been measured, which connects a residential network of a university to the core network of this university. On the residential network, about 2000 students are connected, each having a 100 Mbit/s Ethernet access link. The residential network itself consists of 100 and 300 Mbit/s links to the various switches, depending on the aggregation level. The measured link has an average load of about 60%.*

The other repository files could just as well be processed using the criteria developed in this report. The choice for this part of the repository was made because its link has the highest load of the available repository parts. Therefore, we assume it will exhibit the most interesting behaviour.

Section 3.2 will cover the privacy aspects of using the repository and will discuss its data format. A method for identifying individual TCP connections will be described in Section 3.3. Unfortunately, the repository turned out to have some problems. Section 3.4 will identify these problems, assess the severity by indicating the consequences and give solutions for working with a faulty repository. Finally, Section 3.5 will discuss the impact of the measurement location and describe a method for determining the location of the measurement point with respect to the flow (either close to the sender or close to the receiver).

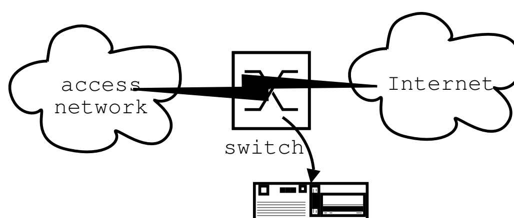
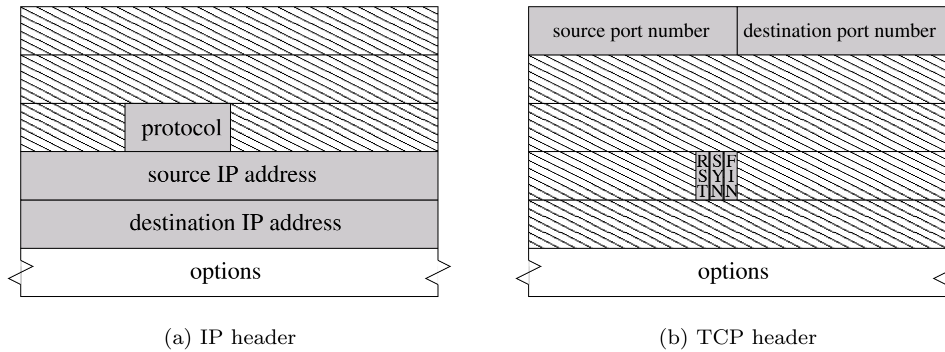


Figure 3.1: *Repository measurement setting*



(a) IP header (b) TCP header

Figure 3.2: *Headers of IP and TCP packets*

3.2 General aspects of the repository

Privacy

Using this repository will not cause any of the violations stated in RFC 1262: "Guidelines for Internet Measurement Activities" [Cer91]. Only the packet headers have been recorded and the source and destination IP addresses have been scrambled. Fortunately, this has been done in such a way that the possibility to trace distinct packets back to a single packet flow is preserved. This is accomplished by assuring that when two packets have the same addresses before the scrambling, they also have the same addresses afterwards (although of course these are other addresses than before the scrambling). Also, when only the first n bits of two original IP addresses were equal, these bits will still be equal to each other after scrambling.

Furthermore, the data has been collected by passive monitoring instead of sending out extra messages. Therefore, no burden has been placed on the network.

Data format

The repository has been structured in several files, each containing fifteen minutes worth of headers. Because the standard utility 'tcpdump' [Res03] was used, the files can be processed using the C programming language in combination with the pcap library. Pcap makes it possible to extract individual packets from the large data files, so that they can be processed.

3.3 Identifying individual TCP connections

Before any examination of TCP packets can begin, we first have to make sure a packet *is* a TCP packet. Therefore, the type field of the Ethernet header is checked for the value 8 (IP) and the protocol field of the IP header is checked for the value 6 (TCP).

In order to make it possible to acquire some useful files for manual exploration, we first divided one file with a large mixture of packet flows into different files; one for each TCP connection. The term connection is used to refer to a half-duplex two-way packet flow. Full-duplex flows (connections where data is sent both ways simultaneously) will not be considered, because those flows are much less easy to analyse. For example, duplicate acknowledgments seem to occur when two data packets are sent without receiving a packet in between. Also, identifying application layer acknowledgments (see Section 7.3) will not be possible anymore. A flow is identified as being half-duplex when one of the directions is transmitting more than ten times as much bytes as the other direction. This choice has been made because even half-duplex flows sometimes contain data in the form of application level acknowledgments of requests (as will be covered in Chapter 7).

Fortunately, not a lot of connections are full-duplex. The rest of this report will use both the terms *connection* and *flow* when a half-duplex two-way packet flow is meant.

The criteria for splitting the packets have mainly been derived from [Mog92]. This paper mentions that although the TCP address tuple (a receiver address and a sender address, both consisting of an IP address and a port number) uniquely identifies a connection at a certain moment, the same tuple can be used later on for a new connections. So, besides dividing the packets based on the TCP address

```
139.103.145.129     219.90.77.117       TCP    fj-hdnet > xrl [ACK, CWR] Seq=29302 Ack=32796
139.103.145.129     219.90.77.117       TCP    fj-hdnet > xrl [PSH, ACK, CWR] Seq=29302 Ack=3
219.90.77.117       139.103.145.129     TCP    [TCP ACKed lost segment] xrl > fj-hdnet [ACK,
139.103.145.129     219.90.77.117       TCP    fj-hdnet > xrl [PSH, ACK, CWR] Seq=29448 Ack=3
219.90.77.117       139.103.145.129     TCP    [TCP Previous segment lost] xrl > fj-hdnet [AC
```

Figure 3.3: *Acknowledgment of "lost" packet*

tuple, also some other considerations should be made. After a FIN segment has been sent and the connection has been closed properly, it is possible to start a new connection between the same IP hosts, using the same source and destination ports [Pos81]. Therefore a SYN segment (identifying the start of a new connection) that belongs to a TCP address tuple where previously a FIN segment has already been processed, should be considered as belonging to a new connection. Moreover, we also interpret a packet with the RST flag set as a FIN segment. After all — as manually observed in real flows — a RST flag can end a TCP flow.

The relevant fields in the TCP and IP headers mentioned so far are highlighted (by means of a grey background) in Figure 3.2.

3.4 Problems with using the repository

3.4.1 Incomplete headers

```
Packet Length: 78 bytes
Capture Length: 66 bytes
```

Figure 3.4: *Captured bytes*

While [vdM03] states that only the first 64 bytes of each Ethernet frame have been captured, this is not the case. Instead, 66 bytes were captured, as can be read from a part of an Ethereal [Eth05] screenshot, displayed in Figure 3.4. Still, 66 bytes is not enough for some packets. The Ethernet header contains 14 bytes, the IP header without options 20 bytes and the TCP header without options 20 bytes; together 54 bytes. So, 12 bytes remain for header options. When a timestamp is provided, 10 bytes and in most cases 2 NOP bytes are included. A second header option, for example a window scaling option, would then not be recorded. This could lead to incorrect results of the receive window detection.

As a solution ambiguous flows are dropped. A flow is ambiguous when all three of the following criteria apply:

- The header of the SYN packet was larger than 66 bytes (otherwise the window scaling options cannot have been lost).
- The window scaling option was not present in the part of the header that was visible (otherwise we do know its value).
- The flow would without the window scaling option be categorized as a receive window limited flow (otherwise it would definitely not be categorized as a receive window limited flow with an even larger receive window).

As it turns out, practically no flows are dropped based on these criteria (so the incompleteness is not much of a problem).

3.4.2 Fake gaps

Because of the unreliability of IP, sometimes TCP packets get lost. If this happens and another packet is sent before the lost packet has been retransmitted, a "gap" occurs. In this report only the missing of a packet not seen before is called a gap. For example, the packet sequence $A\ B\ C\ D\ B\ E$ does not contain a gap, even though E normally does not follow B. However, the sequence $A\ B\ E\ C\ D\ F$ (packet reordering) does contain a gap, because the packets missing between B and E have not been seen before. The reason for this definition is that it makes it possible to detect fake gaps (explained further on). Normally, a gap is filled later on, as is the case with $A\ B\ E\ C\ D\ F$.

Figure	File	Size	Relevant packets	Fake gaps
3.6a	*loc1-20020625-0415*	841 MB	2,194,078	90
3.6b	*loc1-20020526-1115*	1081 MB	2,591,397	300
3.6c	*loc1-20020528-1115*	1849 MB	4,194,771	2,221

Table 3.1: *Repository files used for fake gap analysis*

While examining some traces, it appeared that gaps occur that are not filled at all. Schematic: *A B C E F G...* While *D* is missing, it is never retransmitted. Moreover, sometimes packets that have not been sent yet are acknowledged, as can be seen from the Ethereal screenshot displayed in Figure 3.3. The only logical explanation is that the missing packets *were* in fact correctly transmitted and received, but were just not recorded by the measurement device. These kind of gaps — that are not gaps in reality — will be called *fake gaps*. Automatic detection of these fake gaps has been developed, to get a better overview of the seriousness of the situation. To detect fake gaps, however, we first have to detect gaps in general.

Detecting gaps

As indicated in Chapter 2, the quantity 'next sequence number' is defined as the sequence number of a packet plus its length, so basically it indicates the sequence number of the next expected packet. For detecting gaps, we keep track of the highest expected next sequence number, which is simply the maximum of all next sequence number values observed thus far. When a packet with a higher sequence number than the highest expected next sequence number arrives, this indicates a gap.

Detecting fake gaps

In order to detect a fake gap, we look at whether or not the gaps seen thus far will be filled later on or not. After all, when a gap is fake, the real TCP flow has not experienced any problems. For each data packet observed, check if it is filling some gap we have seen in the past. Because some gaps are larger than one packet, a single packet will not always fill the whole gap. For implementation simplicity we consider a gap filled when at least one packet has filled at least one byte of it. After all, for fake gaps even this will not happen.

Manual examination of some flows resulted in the knowledge that sometimes the border of a gap is crossed. Therefore, a packet that only partially fits in a gap should also be considered a gap filling packet (for example packet X and packet Y in the example of Figure 3.5). More formally: a packet with a sequence number equal to or lower than the last sequence number of a gap (12, in case of the example) and a next sequence number larger than the first sequence number of a gap (9, in case of the example) is considered to fill the corresponding gap.

Now that we know when a gap has been filled, a fake gap is simply defined as a gap that has *not* been filled. In order to give some conclusions about the number of fake gaps in a repository, it is advisable to only consider connections for which a FIN packet has been seen, because otherwise gaps that would have been filled after the measurement period would incorrectly be categorized as being fake gaps.

Fake gaps in the repository

For three fifteen-minutes intervals of repository data a graph of the fake gaps has been made, illustrated in Figure 3.6. Table 3.1 gives information about the files used.

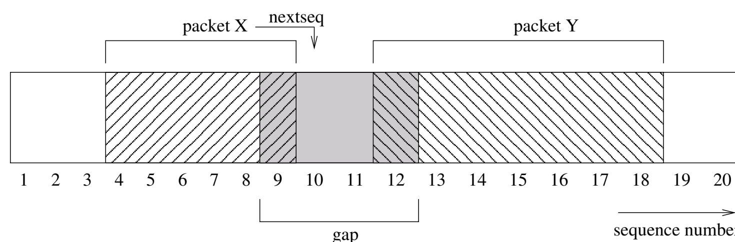
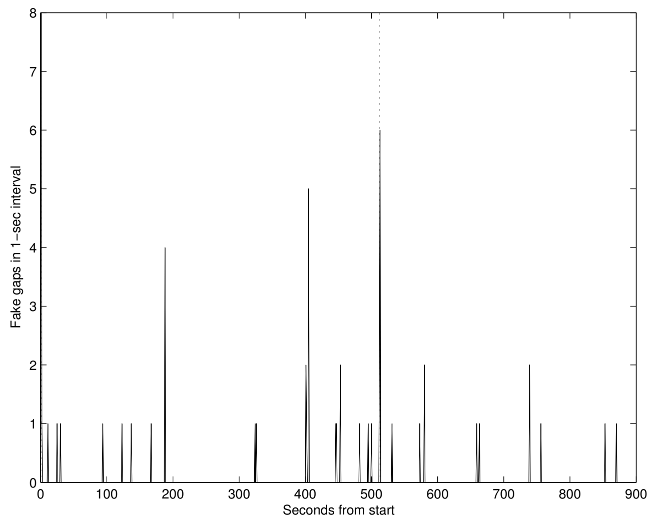
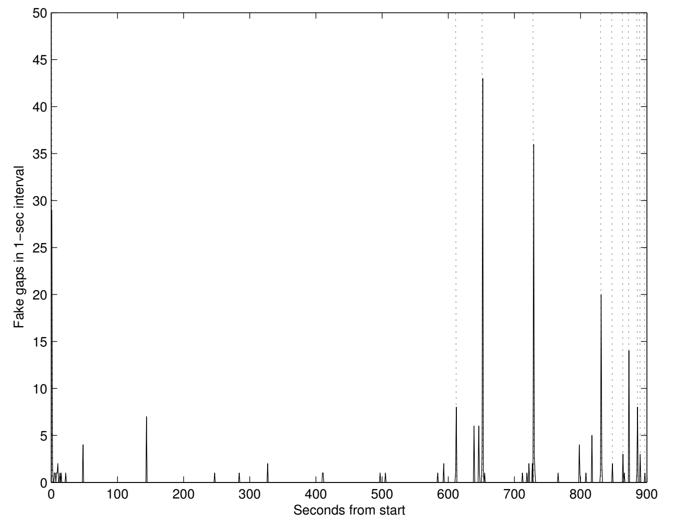


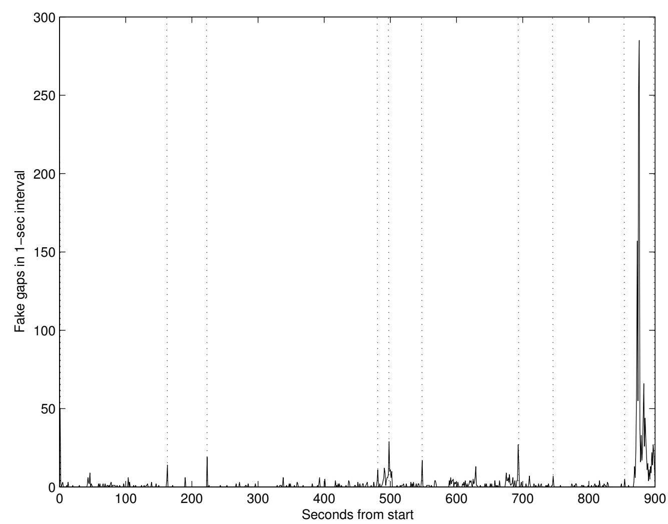
Figure 3.5: *Filling of gaps*



(a)

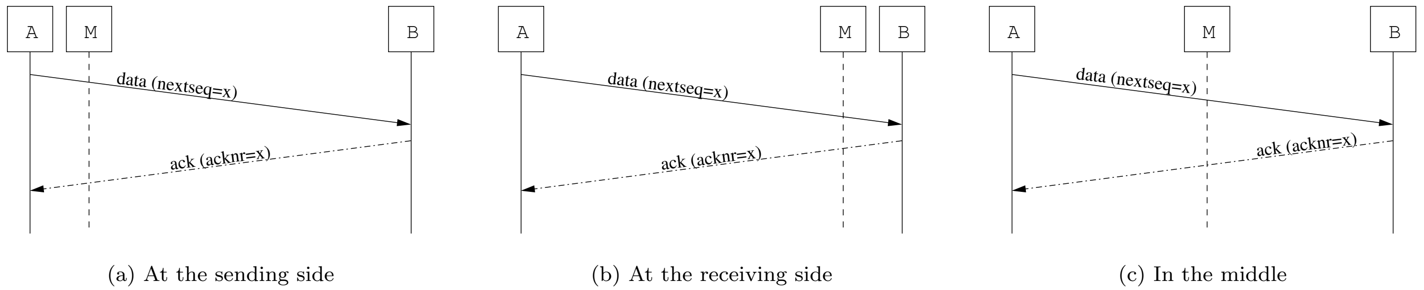


(b)



(c)

Figure 3.6: *Fake gaps in the repository*



(a) At the sending side

(b) At the receiving side

(c) In the middle

Figure 3.7: *Location of the measurement point*

Only connections of which the measurement point is situated at the sending side are considered, because the rest of the analyses also only consider these connections (as will be explained in Section 3.5). Furthermore — as advised in the previous paragraph — only finished connections have been considered. Note that the scale on the vertical axis is different for each graph. For each 1-second interval, the number of fake gaps in the entire repository is shown. The time of occurrence of a gap is defined as the time of the last packet before the gap.

Besides counting fake gaps of the individual connections, an analysis of the recording behaviour of the repository has been made. A dotted line in the figures means that the repository did not record any data for at least 5 milliseconds, which is a very good indication for problems with recording data. After all, even the smallest of the three files contains 12,553,221 packets in 900,000 milliseconds, giving a chance of approximately $6 \cdot 10^{-12}$ of seeing at least one of such voids (see Appendix A for a proof of this statement). Of course the chances for a void occurrence in the other files is even smaller, as they are larger. The time of occurrence of such a void has been defined as the time of the last packet observed before the void.

The two analyses (counting gaps in TCP sequences and counting voids in the repository) were developed and implemented independently and still show peaks at the same times. This clearly supports the correctness of these analyses.

Further research could be performed to investigate why fake gaps occur. For our research, the knowledge that fake gaps occur is enough information. Unfortunately it causes some problems, as we will see in the next chapters.

3.5 Impact of the measurement location

The measurement point is directly connected to an access network (see Figure 3.1), so most of the time it can by approximation be considered as situated at the same location as one of the hosts of a flow (the one situated in the access network).

Because all traffic passing the measurement point has been recorded, for some of the connections the *sending* host will be situated in the access network of the measurement point, and for some of the connections the *receiving* host will. Figure 3.7a and Figure 3.7b illustrate these situations. For some flows, one host is located in the access network illustrated in Figure 3.1, while the other is situated in another access network that is also directly connected to the same switch as the measurement device. A visualization of these kind of flows in given in Figure 3.7c.

At first we attempted to develop general criteria that can be applied for all three scenarios, but this proved to be very difficult if not impossible. Fortunately, problems each time arose with the measurement point situated at the receiving side or in the middle. Therefore, only connections with the measurement point situated at the sending side will be considered. To give an idea of why this choice has been made, in each relevant chapter the problems of examining flows with the measurement point at the receiving side (or in the middle) will be explained.

Determining the location of the measurement point

In order to determine the location of the measurement point, the IP range of the access network located near to the measurement device has to be known. Two facts now come in handy:

- When the first n bits of two original IP addresses were equal, these bits will still be equal to each other after scrambling (although of course they generally will be different from the values before scrambling).
- Each flow contains at least one host that is situated in the access network near the measurement point.

As it turns out (by manual examination of some flows), the access network is identified by the first 16 bits of the IP address. For an automatic establishment of this prefix, just look at the prefixes of the source and destination address of the first packet of the repository (let us call them P and Q). If P and Q are equal, that is the wanted prefix and we are finished. If not, we continue by examining the next packet of the repository (this packet does not have to belong to the same flow as the first one). We check whether or not P is equal to at least one of the prefixes of this packet. If it is *not*, then P could not have been the access network prefix, because that one has to occur in every packet. Therefore, Q has to be the wanted prefix. If P *is* equal to at least one prefix of the packet under examination, perform the same check for Q. If Q is not equal to at least one of the prefixes of the packet, then P has to be the wanted prefix. If both checks fail to establish the right prefix (which will happen only when the prefixes of the examined packet are equal to P and Q), continue with the next packets until the right prefix has been found.

To determine the location of the measurement point for a specific flow after the prefixes are known, look at the IP address of the sending host and the IP address of the receiving host. If the prefix of the IP address of the sending host *is* equal to the prefix of the access network of the measurement device and the prefix of the IP address of the receiving host *is not*, the scenario of Figure 3.7a applies. If it is the other way around, Figure 3.7b applies. Finally, when both prefixes are equal to the prefix of the access network, Figure 3.7c applies.

Chapter 4

Recognizing the Receive Window Limitation

4.1 Introduction

As mentioned in Section 2.3, one of the speed limitations for a TCP flow is the receive window: a quantity advertised by the receiver, to indicate the maximum number of bytes it can handle without having to drop anything because of a buffer overflow. In this chapter we will look at ways to automatically examine TCP flows to identify the receive window limitation. As we will see, an exact conclusion is not always achievable.

First, Section 4.2 will present two diagrams of receive window limited flows, to show what the limitation is about. The remainder of this chapter will then develop a method for detecting the receive window limitation. Roughly, this detection comes down to checking whether or not most of the time the receive window is filled completely by the sender. In order to be able to perform this check, we have to calculate the number of outstanding bytes and we have to relate this value to the receive window. Section 4.3 will discuss the first issue, whereupon Section 4.4 will discuss the second.

Of course the receive window will not be filled all the time, even though it *is* the limiting factor. After all, after receiving an acknowledgment some packets have to be transmitted first, before the receive window is filled again completely. The number of outstanding bytes will therefore increase and decrease every round-trip time. The receive window check should always be performed just before a decrease starts, during what we will call a *peak*. Section 4.5 will provide a framework for receive window checks, that will take care of performing checks exactly on these peaks. The number of times that on such a moment the receive window *was* indeed the limiting factor can be expressed as a percentage. A high percentage would of course then indicate a receive window limitation.

It appears that the receive window can sometimes be the limiting factor of a TCP flow, even though it is never completely filled. Section 4.6 will discuss the scenarios where this occurs and will provide detection criteria that can be applied on the peaks provided by the framework of Section 4.5.

Finally, we will discuss the different causes of a receive window limitation in Section 4.7

4.2 A receive window limited flow

To illustrate the process of recognizing the receive window limitations and to describe the problems that arise, the two diagrams of Figure 4.1 will be used. In these diagrams a TCP flow between two hosts (A and B) is shown. For the semantics of the figures, see Section 2.4. Additionally, in this chapter a data packet with $nextseq = x$ will be referred to as 'data packet x'. Symmetrically, an acknowledgment packet with $acknr = x$ will be referred to as 'ack packet x'.

The illustrated flow is restricted by the receive window, so the sender (host A) can only send a limited number of data packets. At some point (in this example after sending data packet x_3) the number of outstanding bytes is equal to the receive window that is advertised by the receiver, so the data flow will halt. Subsequently, after a while the receiver receives the data and sends acknowledgments. The sender receives these and can start sending new data, until the difference between the acknowledgment number of the last received ack packet and the sequence number of the last sent data

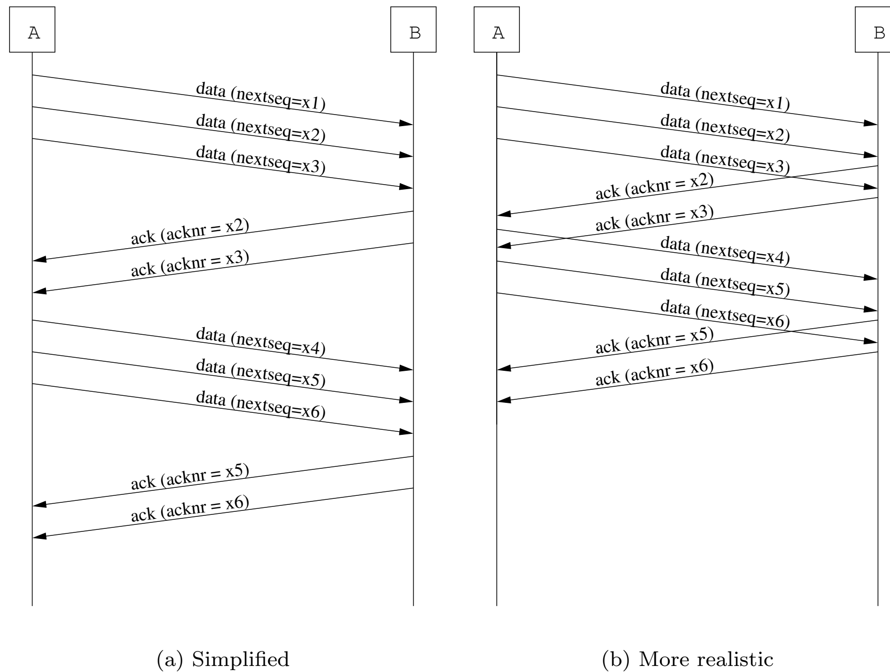


Figure 4.1: *Time-sequence diagrams of a TCP flow that is restricted by the receive window*

packet is again equal to the receive window (this occurs after data packet x_6).

Figure 4.1a shows the situation in a simplified manner; the receiver first receives all the data packets and after that sends all the acknowledgments. Furthermore, the sender starts sending new data packets after all the acknowledgments have been received. Figure 4.1(b) gives a more realistic view, in which the data packets and ack packets are intertwined. The next section will use these figures as examples for calculating the number of outstanding bytes.

4.3 Calculating the number of outstanding bytes

To recognize that a TCP flow is being limited by the receive window, it is necessary to know up to how far the receive window has been filled. Calculating the number of outstanding bytes is a logical way to do this. This section will first define outstanding bytes and then look at ways to determine how many outstanding bytes there are.

Outstanding bytes are defined as bytes that have already been sent, but that have not been acknowledged yet (from the sender point of view). Figure 4.2 illustrates this. Each vertical bar represents a byte and the sequence numbers of these bytes are ascending from left to right.

The bytes in segment 1 were sent in the past and the sender has already received acknowledgments

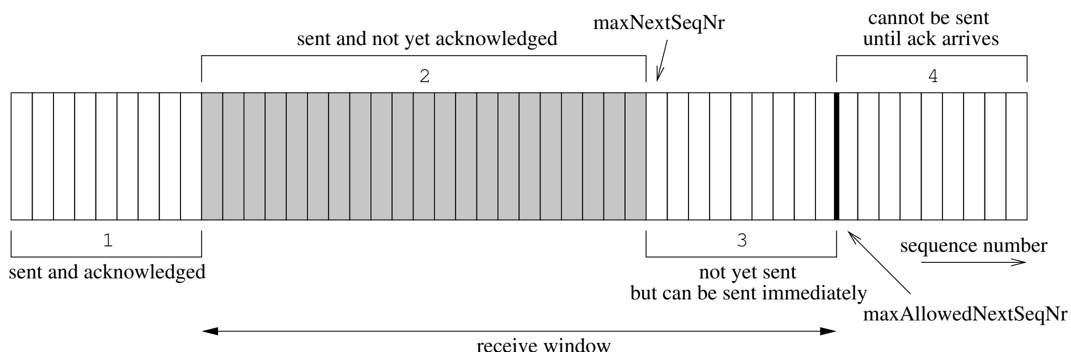


Figure 4.2: *TCP sliding window*

for them, so these packets are not relevant to TCP anymore. Segment 4 is not very interesting at present time either; the bytes in this segment cannot be sent because of the receive window. Segment 2 and 3 are the most important for our discussion. Together they have the size of the receive window, so they contain exactly the number of outstanding bytes allowed. Although all of these bytes can be sent, there will of course be moments when not all bytes allowed indeed have been sent. Therefore these two segments are separated. Segment 2 represents the bytes that have already been sent but not yet acknowledged, while segment 3 contains the bytes that have not been sent yet, but are allowed to be sent.

Thus, the number of outstanding bytes is equal to the number of bytes in segment 2.

If the amount of outstanding bytes is known, this can be compared to the advertised receive window. In case these two are equal, the data flow is apparently limited by the receive window.

The difficulty is in determining this number of outstanding bytes. Because only flows with the measurement point close to the sender are considered (as indicated in Section 3.5), this is possible. In order to give an idea of the problems that would arise if this choice would not have been made, this section will first try to develop a general method and discuss its limitations.

For a general method, the measurement point is situated somewhere between the sender and the receiver. In that case, as we will see, we do not have all the information we would like to have. Still, it is possible to draw some conclusions. We will develop a method for determining a lower limit on the number of outstanding bytes, first by looking at a simplified example and then by examining a realistic situation.

Simplified situation

In the simplified example of Figure 4.1a it is immediately clear how to measure the number of outstanding bytes. Just take the last packet of a 'burst' of data packets (Section 4.5 will deal with identifying this moment) and calculate the difference between its sequence number and the largest acknowledgment number seen so far. In theory this gives a lower bound on the amount of outstanding data (as will be explained further on in this section), but for this model it also gives an upper bound in case no loss occurs. After all, we observe the packets in the order they were sent and received and no acknowledgments and data packets are intertwined. The largest number of outstanding bytes is therefore equal to $x_6 - x_3$. This is only the number of outstanding bytes in the period between sending data packet x_6 and receiving ack packet x_5; at other moments it is smaller.

This calculation can be performed on any location of the route, i.e., the location of the measurement equipment is irrelevant. However, when we will increase the complexity of the model by giving the receiver a shorter delay for sending its acknowledgments, problems arise.

Realistic situation

Figure 4.1b shows a more realistic model of the same packet flow. The sending of ack packets is already begun before all the data packets have arrived at the receiver. Now the location of the measurement equipment becomes relevant.

When we measure close to the sender, the sequence of packets observed is different from the sequence observed when we measure close to the receiver. Both sequences are shown in Table 4.1.

If we perform the same calculation as mentioned before (taking the difference between the sequence number of a data packet and the largest acknowledgment number seen so far) on the measurements done at the sending side, we calculate a maximum number of $x_6 - x_3$ outstanding bytes, while we calculate a maximum of $x_5 - x_3$ bytes working with the data collected at the receiving side.

This example shows that the same packet flow can produce different observed sequences of packets, based on the location of the measurement equipment. When we reside on the receiving side, ambiguities are introduced. If we observe an ack packet at time t_1 and we observe a data packet at time t_2, it is not certain whether the data packet was sent before of after receiving the ack packet at the sending node, even though $t_1 < t_2$. Figure 4.3 illustrates the problem. At the receiving side (B) there is no way to distinguish between the scenario where the data packet was sent prior to receiving the ack packet and the scenario where the data packet is a 'response' to the ack packet. On the sending side this ambiguity is much less likely to occur, as can be concluded from the figure.

However, this does not mean we cannot draw any real conclusions. It is always possible to use the above mentioned method of subtraction as one of the criteria for detecting a receive window limitation. It will not always prove the existence of this limitation when it does exist, but it will never draw the

	(a) Sending side		(b) Receiving side
1.	DATA ($nextseq = x_1$)	1.	DATA ($nextseq = x_1$)
2.	DATA ($nextseq = x_2$)	2.	DATA ($nextseq = x_2$)
3.	DATA ($nextseq = x_3$)	3.	ACK ($acknr = x_2$)
4.	ACK ($acknr = x_2$)	4.	DATA ($nextseq = x_3$)
5.	DATA ($nextseq = x_4$)	5.	ACK ($acknr = \boldsymbol{x_3}$)
6.	ACK ($acknr = \boldsymbol{x_3}$)	6.	DATA ($nextseq = x_4$)
7.	DATA ($nextseq = x_5$)	7.	DATA ($nextseq = \boldsymbol{x_5}$)
8.	DATA ($nextseq = \boldsymbol{x_6}$)	8.	ACK ($acknr = x_5$)
9.	ACK ($acknr = x_5$)	9.	DATA ($nextseq = x_6$)
10.	ACK ($acknr = x_6$)	10.	ACK ($acknr = x_6$)

Table 4.1: Observed packet sequences

conclusion that the receive window was the limiting factor when this is not the case. After all, even though we do not know what the sending node *has* seen, we do know what he *has not* seen yet.

For example, consider the following sequence of packets. The quantities behind ACK and DATA indicate the acknowledgment number, respectively the sequence number of the next byte to send. The subscripts are just for referencing.

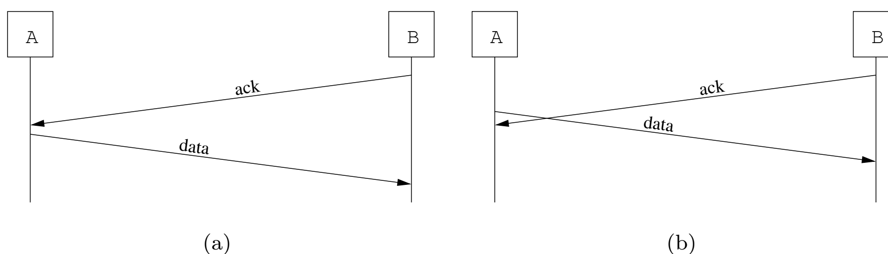
1. ACK x_1
2. ACK x_2
3. DATA x_3
4. ACK x_4

If we perform the calculation mentioned before, we produce a lower limit on the number of outstanding bytes. This lower limit is equal to $x_3 - x_2$ (assuming $x_2 \geq x_1$). After all, ack packet x_2 is the last possible packet the sender has received. Because the measurement point is situated between the sender and the receiver, ack packet x_4 cannot have reached the sender prior to sending data packet x_3.

Because of packet reordering it is possible that $x_2 < x_1$. Therefore, instead of subtracting the last observed ack packet it is better to subtract the *largest* acknowledgment number seen thus far. For the remainder of this section, it is still assumed that $x_2 \geq x_1$.

Although a lower limit on the number of outstanding bytes has been derived, it is not certain if it is the real number. After all, we do not know whether or not the sender already received ack packet x_2 before sending data packet x_3.

If the sender *has* seen ack packet x_2 before sending data packet x_3, the amount of outstanding bytes is equal to $x_3 - x_2$. If the acknowledgment was however *not* yet received at that point, the receiving node was still under the assumption that a smaller number of packets was acknowledged. Therefore, the number of outstanding data bytes was equal to $x_3 - (x_2 - \phi) = x_3 - x_2 + \phi$, $\phi \geq 0$. Because this results in an even larger number of outstanding bytes, $x_3 - x_2$ produces a lower limit on



(a) (b)

Figure 4.3: *Ambiguities at the receiving side*

the number of outstanding data bytes.

Concluding, we can be sure that we get a lower limit on the amount of outstanding data when we take the difference between the sequence number of a data packet and the largest acknowledgment number seen thus far. At the receiving side there is a chance that this is not the upper limit, while on the sending side this occurs much less often. As already mentioned in Section 3.5, our analyses will therefore only consider measurements performed at the sending side.

4.4 Relating outstanding data to the receive window

In the previous section, a way to obtain a lower limit on the number of outstanding bytes has been explained. Once we know this limit, it can be compared to the receive window. When the lower limit is equal to the receive window, we know for sure that the receive window is the limiting factor. After all, then we know the number of outstanding bytes is at least as large as the receive window (because of the lower limit) and we already knew it is at most as large (because of the TCP specification). This directly results in the conclusion that the number of outstanding bytes is equal to the receive window and the receive window is therefore the limiting factor.

A new issue that now arises is how to determine the receive window that was known by the sender at the moment of sending a data packet. As it turns out, relating outstanding data to the receive window should not be performed directly, but can better be performed implicitly by comparing the maximum next sequence number the sender will send with the maximum next sequence number the receiver allows by means of acknowledgments and the receive window. A detailed explanation will be given at the end of this section.

In order to get an idea of the way of thinking and the path up to this conclusion, some faulty decisions are explained first.

Faulty method 1: relating outstanding data to the last seen receive window

Let us take a look at what would go wrong if we just take the last observed receive window advertisement by going through the following example sequence.

1. DATA $nextseq = 10001$
2. DATA $nextseq = 10501$
3. ACK $acknr = 10001$, $rwnd = 1000$
4. ACK $acknr = 10501$, $rwnd = 300$
5. DATA $nextseq = 10801$

A sequence like this one could be produced in the following situation. The receiver has an initial buffer space of 1000 bytes. At a certain moment, it received data bytes up to sequence number 10000 (packet 1). These bytes were passed to the application immediately, so the receive window advertised in the acknowledgment is still 1000 (packet 3). A little bit later the 500 new bytes of data packet 2 were received, but the application at the receiving side could not yet handle them, so they stay in the buffer for a while. Moreover, the size of the buffer was decreased to 800 bytes; a shrinking of the window. Although shrinking of the receive window is strongly discouraged, it is permitted by the TCP standard [Pos81]. After shrinking, the buffer has only 800 bytes in total and is already filled with 500 bytes of data. Therefore, only 300 bytes are available and this amount is advertised by the receiver (packet 4).

Later on we observe a data packet with $nextseq = 10801$ (packet 5). If we would now take the last seen receive window (300 bytes) as the receive window to perform our calculations with, the following result would be achieved.

$$receive_window = 300 \text{ bytes}$$
$$lower_limit_on_bytes_outstanding = 10801 - 10501 = 300 \text{ bytes}$$

These calculations would lead to the conclusion that the receive window is the limiting factor. In fact, however, it is not 100% sure this is the case. It could be possible that the sender has not received the second acknowledgment at the moment it sent its data packet. Therefore, the receive window is

never reached in this example. After all, data was sent up to sequence number 10800, while the last acknowledged byte had sequence number 10000 and the receive window was 1000 bytes. The sender could still send 200 bytes but did not do this, so it was not the receive window that limited the speed of the data flow. It is possible (maybe even likely) that the decreased buffer size causes a receive window limitation in the future, but formally is would be incorrect to conclude this limitation for the given measurements already. This proves that using the last observed receive window size in our calculations can result in incorrect conclusions.

Faulty method 2: relating outstanding data to the highest seen receive window
Another option would be to use the largest receive window that has been observed for this flow. Although this would have eliminated the problem we had using method 1, a new problem arises. We will use the following example sequence of data and acknowledgment packets to show the limitations of this problem.

1. DATA $nextseq = 10001$
2. DATA $nextseq = 10701$
3. ACK $acknr = 10001$, $rwnd = 1000$
4. ACK $acknr = 10701$, $rwnd = 500$
5. DATA $nextseq = 11001$
6. DATA $nextseq = 11201$

A sequence like this one could be produced in the following situation. The receiver has a buffer space of 1000 bytes. At a certain moment, it received data bytes up to sequence number 10000 (packet 1). These bytes were passed to the application immediately, so the receive window advertised in the acknowledgment is still 1000 (packet 3). A little bit later the 700 new bytes of data packet 2 were received, but the application on the receiving side could only handle 200 of them. Therefore, the buffer had only 500 free bytes left and the receive window was decreased to 500 (packet 4).

The sender received the acknowledgments and in return has sent new data bytes, first 300 new bytes in response to receiving the first acknowledgment. Then, the second acknowledgment packet arrives. The receive window is equal to 500 and the last acknowledged packet had sequence number 10700, so data packets up to sequence number 11200 can be sent (and are sent). It is clear that the receive window *is* the limiting factor here.

If we take the maximum receive window ever observed for this flow (1000 bytes), we would perform the following calculations.

$receive_window = 1000$ bytes
$lower_limit_on_bytes_outstanding = 11201 - 10701 = 500$ bytes

This would lead to the conclusion that the receive window was not the limiting factor, even though it is. This proves that using the highest observed receive window size in our calculations can result in incorrect conclusions.

Solution: relating the sequence numbers
Fortunately there is a solution that does work (assuming we have a correct value for the number of outstanding bytes, so only reliable on the sending side). Each time we observe an acknowledgment packet, we calculate the maximum next sequence number the sender is allowed to send by adding the receive window to the acknowledgment number. Furthermore we keep track of the maximum next sequence number of the sender (defined in Section 2.4). Figure 4.2 illustrates that the difference between these two quantities gives the number of bytes still available in the receive window. The following mathematics prove this statement.

$$\begin{aligned} maxAllowedNextSeqNr - maxNextSeqNr &= receiveWindow + lastAck - maxNextSeqNr \\ &= receiveWindow - (maxNextSeqNr - lastAck) \\ &= receiveWindow - bytesOutstanding \end{aligned}$$

At first sight, a value of zero for this difference would be the criterion for a receive window limitation

(after all, in this case the receive window and the number of outstanding bytes are equal). As we will see in Section 4.6, other criteria also apply. Nonetheless, knowing the difference between the receive window and the number of outstanding bytes will prove to be very useful.

By taking the *maximum* of all the observed allowed next sequence numbers, the problems of the two described wrong methods are solved. By taking the maximum, we are always sure we are not performing our calculations with a receive window that is too small. The problems that arise with method 1 (relating outstanding data to the last seen receive window) are solved, because shrinking the receive buffer will not lead to an incorrect decrease of the receive window anymore. Also the problems observed with method 2 (relating outstanding data to the highest seen receive window) are solved, because the maximum sum of the acknowledgment number and the receive window will still increase in case of an application that does not read data fast enough.

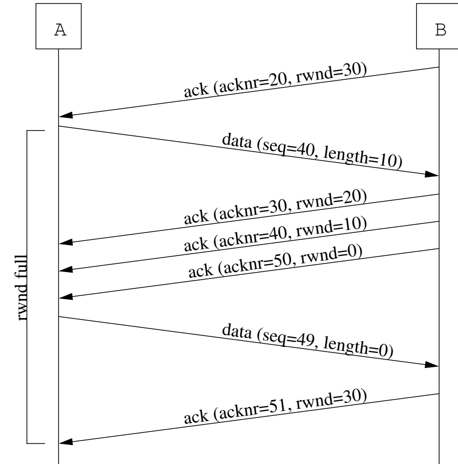


Figure 4.4: *Receive window limited flow*

Taking the maximum of all the observed next sequence numbers also eliminates some problems. After all, if the sender first has sent data up to sequence number x and later on it sends a packet with next sequence number $x - \phi$, the number of outstanding bytes will still be $x - lastAckNr$. An example where not taking the maximum would lead to problems is illustrated in Figure 4.4. In this figure we observe something that happens occasionally: the receiver advertises an empty receive window. When it takes a while before the window is updated, the sender can send a keep-alive message. The keep-alive message contains a sequence number that is equal to the last acknowledged byte (so it is using a sequence number that is already used and acknowledged) and has length zero. Because the sequence number is one lower than what would be logical and the length is zero, the next sequence number to be sent appears to be decreased by one. In the figure we have a value of 50 for the maximum allowed next sequence number after the ack burst, while the next sequence number is equal to 49 after the data packet. Therefore, it would look like there is still 1 byte available to be sent and therefore the conclusion would be that the receive window is not the limiting factor. Of course this would be an incorrect conclusion and taking the maximum solves this.

4.5 Framework for performing receive window checks

So far we have looked at ways to detect receive window limitations, by looking at the number of bytes left in the receive window. However, the receive window will not be filled all the time, even in a flow that is strictly limited by it. Let us take a look at a receive window limited flow, illustrated in Figure 4.5. We will develop criteria for detecting the moments when the receive window filling should be checked.

Underlying principles

Human beings can easily see that the sender transmits up to the maximum sequence number it is allowed to send, waits some time and after receiving acknowledgments starts sending again; up to the new maximum allowed next sequence number. To a human there would be no discussion about whether or not this flow is receive window limited. For automatic detection however, formal definitions of receive window limited flows have to be specified.

As we established earlier, the equality between the maximum allowed next sequence number and the maximum next sequence number can be used. However, this equality will not be present during the complete flow. For example, if we measure at time P (see Figure 4.5), the number of outstanding bytes is 20, while the receive window gives permission for 30. If we then measure at time Q, the number of outstanding bytes will suddenly *be* equal to the receive window. Obviously, we cannot use look at random points in time at the mentioned equality.

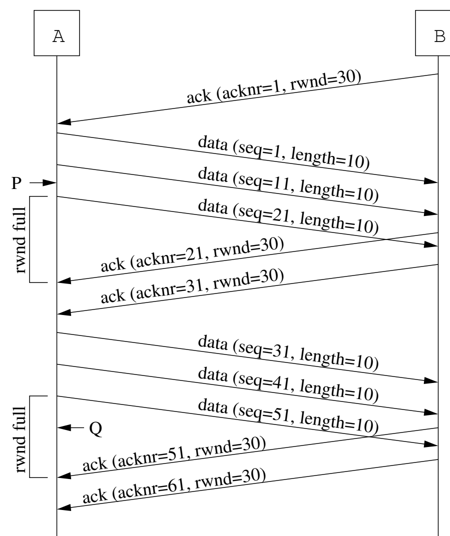


Figure 4.5: *Receive window limited flow*

Another method would be to perform a check for each packet we observe. In this example scenario, that would result in a receive window limitation of approximately 20 percent (or 33 percent if we only perform checks when we observe a data packet). Clearly, although always providing some indications for a receive window limitation, this method still results in a relatively low percentage for a flow that is obviously receive window limited.

A solution to this problem has been found. Because we are measuring at the sending side (as explained in Section 3.5), it will be easy to identify the end of a 'data burst'; a couple of data packets that are sent close to each other, before waiting for new acknowledgments. This moment is called a *peak*. We would like to check for a receive window limitation at the time of such a peak, because this is the moment a receive window limitation is likely to occur (earlier in a burst the maximum will not be reached yet, otherwise the next packet would not be allowed to be sent).

At the time we are checking for a receive window limitation, it is useful to know up to how far the sender has sent and up to how far the sender is allowed to send based on the acknowledgment number and the receive window advertised in the acknowledgment seen last before the data burst. After all, as mentioned earlier, the difference between these two numbers indicates the number of bytes that are still available in the receive window.

Criteria and mechanism

We can achieve a proper timing for receive window checks by checking if the packet under examination is an acknowledgment packet and if the previous packet of the flow it belongs to was heading the other way. In this case we are examining the first acknowledgment packet following a data burst and the receive window limitation checks have to take place.

To know up to how far the sender has sent, we keep track of the maximum next sequence number the sender will use, which is updated for each data packet seen. To know up to how far the sender is allowed to send, we keep track of the maximum allowed next sequence number, updated for each acknowledgment packet seen. Because we want to keep track of the maximum, we only update the values if they are not decreasing. Both qualities and the reason for taking the maximum are discussed in Section 4.4.

In order to handle the wrapping of sequence numbers correctly, a large difference (we use the value 100,000) between the maximum allowed next sequence number and the next sequence number is used as an indication that we have started at 0 again; in that case, the maximum allowed next sequence number should be reduced to the current next sequence number. The same holds for the maximum next sequence number.

4.5.1 Additional criteria due to measurement errors

As mentioned in Section 3.4.2, there have been some problems with logging all the headers. As a result of this, fake gaps occur in the repository. Therefore, the number of outstanding bytes can appear to be higher than it is in reality. This section will explain the problem and provide a solution.

Figure 4.6 shows an example of a flow where two packets did not get recorded (indicated by a gray color), to illustrate the problem. At time P a peak is detected and we have a value of 51 for the maximum allowed next sequence number. The maximum next sequence number is equal to 41. Therefore, the number of bytes left in the receive window seems to be 10, even though in reality this value was never smaller than 30. Not logging an acknowledgment packet produces errors in the calculation. Therefore an additional criterion for performing receive window checks is proposed: *when a gap is observed, wait until the second normal acknowledgment has been seen before checking the number of outstanding bytes again* (in that case, the first data packet after the first normal acknowledgment will be used as the start of a data burst). Because gaps that are surrounded by duplicate acknowledgments or retransmissions are not always correctly detected, we choose also to *wait for the second normal acknowledgment after seeing a duplicate acknowledgment or a retransmission.*

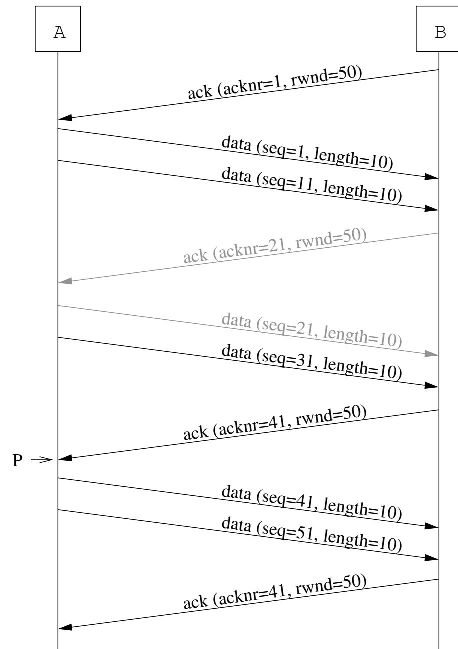


Figure 4.6: *Receive window checking*

Algorithmic description
Figure 4.7 gives an algorithmic framework for detection based on this theory.

First the conditions for a first acknowledgment packet after a data burst are checked and if they are met, the receive window detection is performed. Afterwards, the two mentioned quantities are updated.

As a preview, three methods for detecting the receive window limitation are already placed in the *checkRwndLimitations*-method. In the next section, these methods are explained.

4.6 Different kinds of receive window limitations

In the previous sections we have identified a method for calculating the number of bytes the sender is still allowed to send (by taking the difference between the maximum allowed next sequence number and the maximum next sequence number). At first sight, a value of zero for this difference (a complete filling of the receive window) would indicate a receive window limitation.

However, as it turns out, there are other scenarios in which the number of outstanding bytes is (almost) never equal to the receive window, even though these scenarios *are* receive window limited. In this section a more in-depth overview of possible receive window scenarios will be given and for each scenario a detection algorithm is presented. These algorithms are written in the form of functions that can be incorporated in the framework, derived in the previous section. The results of these detections can be added, in order to draw conclusions based on the sum. After all, we are interested in the receive window limitation and not specifically in the implementation details that cause these limitations.

Recall that we only consider measurements performed at the sending side (as explained in Section 3.5).

```
For each packet:
      if (this packet is an ACK packet and the previous packet
          of this flow was heading the other way and this is
          at least the second normal acknowledgment after the previous gap) {

           checkRwndLimitations();

      }
      if (this packet is a DATA packet) {

           nextSeqNr := TCP_seqNr + packetLength
           maxNextSeqNr := max(maxNextSeqNr, nextSeqNr)
           if (maxNextSeqNr - TCP_seqNr > 100000) {

               maxNextSeqNr := nextSeqNr

           }

      }
      if (this packet is an ACK packet) {

           allowedNextSeqNr := ackNr + rwnd · 2^windowScaling
           maxAllowedNextSeqNr := max(maxAllowedNextSeqNr,
                                      allowedNextSeqNr)
           if (maxAllowedNextSeqNr - TCP_seqNr > 100000) {

               maxAllowedNextSeqNr := allowedNextSeqNr

           }

      }

checkRwndLimitations() {

      checkCompleteRwndUtilization()
      checkIntegerPacketRwndUtilization()
      checkBlockBasedRwndUtilization()
      numberOfChecks++

}
```

Figure 4.7: *Framework for detecting the receive window limitations*

4.6.1 Complete window utilization

The complete window utilization scenario is in fact the most natural receive window limitation scenario. In this case, the sender will completely fill the receive window with data. Figure 4.8 illustrates this scenario.

Criteria

The criterion for detection of this scenario is easy: because the complete receive window is filled, the number of outstanding bytes at the end of a data burst will be equal to the receive window. Symmetrically, the sender will have sent exactly up to the maximum allowed next sequence number.

The ratio of the number of times a complete window utilization is detected and the times the function is performed gives a measure for the complete window utilization scenario of the receive window limitation.

Figure 4.9 contains an algorithm for detecting the complete window utilization receive window limitation. This function can be included in the framework as presented on page 24.

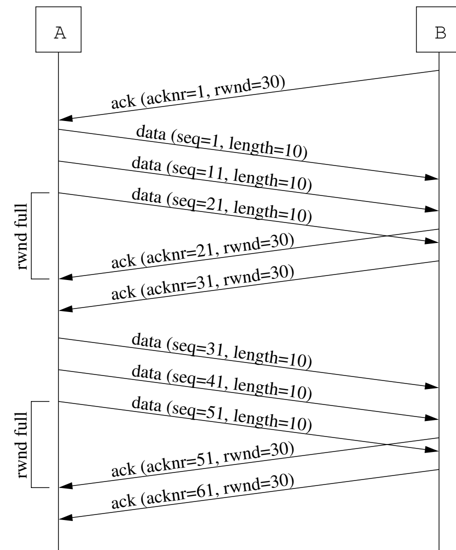


Figure 4.8: *Complete window utilization*

```
checkCompleteRwndUtilization() {
      if (maxNextSeqNr == maxAllowedNextSeqNr)
           completeRwndUtilizations++
}
```

Figure 4.9: *Algorithm for detecting complete window utilizations*

4.6.2 Integer MSS window utilization

The integer MSS window utilization scenario is a receive window limitation variant in which the sender is reluctant to send non-full packets. Figure 4.10 illustrates this scenario.

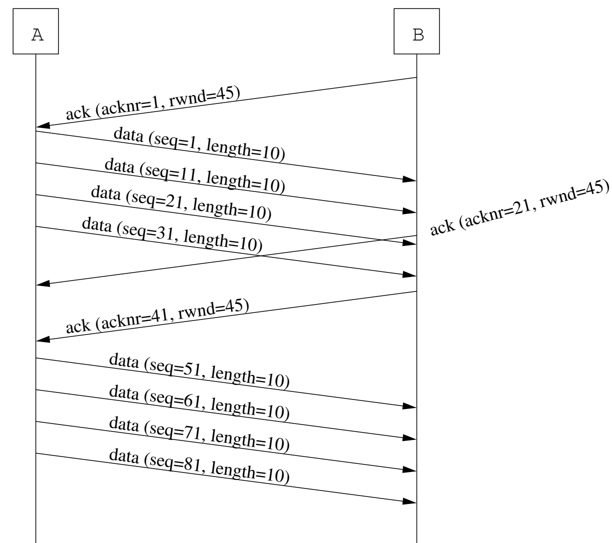


Figure 4.10: *Integer MSS window utilization*

Criteria
As we can see, after receiving the first acknowledgment the sender would be able to send up to sequence number 45. However, after sending up to 40 it stops. It appeared that TCP does not want to send non-full packets under certain conditions. These kind of flows are just as well limited by the receive window.

The criterion based on which is decided whether or not a flow is categorized as integer MSS window utilization limited, is composed of three parts:

- The number of bytes left to be sent has to be more than zero, otherwise the full window utilization scenario would apply.
- The number of bytes left to be sent should be less than the Maximum Segment Size (MSS), otherwise another packet could have been sent.
- The size of the last packet has to be equal to the MSS. If it was not, it is obvious that the flow is not limited by the integer MSS window utilization scenario of the receive window limitation. After all, the packet could have contained more data if that was available.

Figure 4.11 contains an algorithm for detecting the integer MSS window utilization receive window limitation.

```
checkIntegerPacketRwndUtilization() {

     if (0 < maxAllowedNextSeqNr - maxNextSeqNr < MSS &&
         sizePreviousDataPacket == MSS)

          integerPacketRwndLimitations++

}
```

Figure 4.11: *Algorithm for detecting integer MSS window utilization*

4.6.3 Block-based window utilization

While manually looking at some flows, we have identified flows that *are* receive window limited, even though the number of bytes the sender is allowed to send (almost) never becomes less than one MSS. Such scenarios would therefore not be detected by the two algorithms described earlier in this section.

As it turns out, these flows send their data in specific sized bursts, which we call *blocks*. Figure 4.12 shows an example of a flow that displays this kind of behaviour. Each time the data is sent in blocks of 4096 bytes. After sending the first block, $7000 - 4096 = 2904$ bytes are available, so at least a full-sized packet of 1460 bytes could be sent. Still the sending entity waits until it can send a complete block.

Detecting this receive window limitation scenario requires some extra work. Prior to being able to detect the limitation, a method has to be developed to identify the just described block-based sending behaviour. To prevent errors from occurring while analysing flows that for example display block-based sending behaviour in only the first half of the flow, it is advisable to perform this check regularly (e.g. once per two hundred packets).

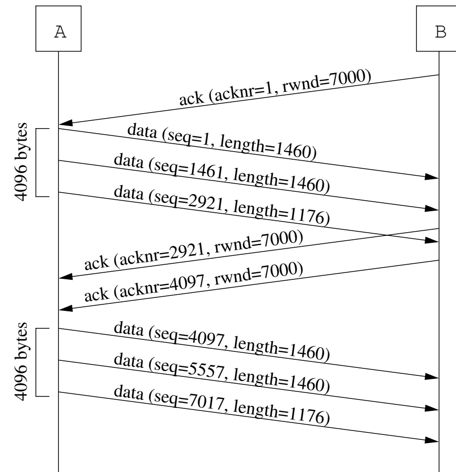


Figure 4.12: *Block-based window filling*

Criteria

As mentioned, with block-based sending behaviour, the sender sends its data in specific sized blocks. The size of the first packets of such a block will be equal to the MSS and only the last one will be smaller, in order to arrive at the block size. This can be used to derive a criterion for detecting the block-based sending behaviour.

To identify the block-based sending behaviour, we look at the size of an amount of blocks. A block is defined as a sequence of packets following a packet that is non-full and non-empty, up to and including the next packet that is also non-full and non-empty. Therefore, each block consists of zero or more full-sized packets, ended by one packet that is non-full and non-empty. In the example scenario of Figure 4.12 two blocks of 4096 bytes are shown.

After the amount of blocks have passed by, a check can be performed on whether or not a certain percentage of the blocks had the same size. In our implementation ten blocks are analysed and when nine or ten of them have the same size, block-based sending behaviour is assumed. Therefore, one block with a different length is treated as an exception in a flow that is still block based (presumably this exception occurred because of a fake gap, see Section 3.4.2). Further research could be performed to check whether or not these values are optimal. For a block-based flow, the block size is of course equal to the size of the equally sized blocks.

An algorithmic description of the block-based sending behaviour detection is presented in Figure 4.13.

The criterion based on which is decided whether or not a flow is categorized as block-based window utilization limited, is composed of four parts. These have to be checked at the times, proposed by the framework of Section 4.5. The four parts are:

- Block-based sending behaviour is detected for this flow.
- The number of bytes left to be sent has to be more than zero, otherwise the full window utilization scenario would apply.
- The number of bytes left to be sent should be less than the block size, otherwise another block could have been sent.
- The last data packet should have had the length of the small 'ending packet' that we saw identifying the block-based sending behaviour. After all, if a flow is indeed limited by the receive window in combination with the block-based sending behaviour, it only sends its data in full blocks. The last packet before halting for acknowledgments should therefore be the end of a block and thus have the size (smaller than the MSS) that is each time required to fill a block up to its specific size.

Figure 4.14 contains an algorithm for detecting the block-based window utilization receive window limitation.

```
For each packet:
      if (packetNumber % 200 == 0) {
           numberOfBlocksRecorded := 0
           de-initialize lastEndPacketSeqNr
      }
      if (numberOfBlocksRecorded < 10) {
           if (this is a DATA packet && packetLength < MSS) {
               if (lastEndPacketSeqNr is initialized) {
                   blocks[numberOfBlocksRecorded] := TCP_seqNr - lastEndPacketSeqNr
                   numberOfBlocksRecorded++
               }
               lastEndPacketSeqNr := TCP_seqNr
               lastEndPacketLength := packetLength
           }
      }
      else if (numberOfBlocksRecorded == 10) {
           if (nine or ten blocks in block-array are of equal size) {
               • The flow is block based
               • blockSize := the size of these nine or ten packets
               • endPacketLength := blockSize - n · MSS
                 (n such that 0 < endPacketLength < MSS)
           }
      }
}
```

Figure 4.13: *Algorithm for detecting block-based sending behaviour*

```
checkBlockBasedRwndUtilization() {
      if (this flow is block based) {
           if (0 < maxAllowedNextSeqNr - maxNextSeqNr < blockSize &&
               sizePreviousDataPacket == endPacketLength) {
               blockBasedRwndLimitations++
           }
      }
}
```

Figure 4.14: *Algorithm for detecting block-based window utilization*

4.7 Identifying the cause of a receive window limitation

Two distinct causes of the receive window limitation have been identified, as already mentioned in Section 2.3. Some flows always immediately return acknowledgments with the receive window equal to the largest receive window observed thus far, thereby indicating that the application layer already cleared the buffer in the time between receiving the data packets and sending their acknowledgments. Therefore, it seems that the receive window could easily have been made larger and the speed is not really limited by the capability of the receiver to process the data in time, but just by a sub-optimal configuration in the TCP stack of the operating system.

Other flows exhibit the behaviour of advertising a receive window that is *not* all the time equal to the largest receive window observed thus far. It seems that in this scenario the application layer is indeed having trouble reading the data in time, so the receive window is performing its task as it is supposed to.

Identification criterion

In order to identify the cause of a receive window limitation, we look at the size of the receive window advertised in each acknowledgment. Count the number of times the advertised receive window is smaller than the largest receive window observed thus far (which is assumed to be the size of the receive buffer) minus twice the MSS. We allow a decrease of twice the MSS, because that is the maximum amount of bytes that in general will be acknowledged in one acknowledgment packet (using duplicate acknowledgments). When two data packets arrive shortly after each other it is possible that TCP sends its acknowledgment before the application had a chance to read the data, so we only consider a drop of the receive window size larger than twice the MSS as an indication of a slow application.

When the whole flow is processed, the number of acknowledgments with such a decreased receive window can be presented as a percentage of the total number of acknowledgments. A high percentage would then indicate that the receive window is working as it is supposed to (the application layer cannot process the data any faster), while a low percentage would indicate that the operating system has chosen its receive buffer size sub-optimal. Chapter 8 will discuss the precise interpretation of the percentage.

Chapter 5

Recognizing the Send Buffer Limitation

5.1 Introduction

Because the TCP specification contains a mechanism (the receive window) for advertising the room left in the receive buffer, it is relatively easy to detect receive window limitations with certainty. However — as announced in Section 2.3 — there is another buffer in the protocol stack that can just as well limit a TCP flow, even though it is not put down in a quantity in the TCP protocol: the send buffer.

In this chapter we will develop criteria for detecting the send buffer limitation. Section 5.2 will present a diagram of a send buffer limited flow, to show what the limitation is about. Then, in Section 5.3, an approach for detecting a send buffer limitation is presented. The approach is approximately equal to the detection mechanism of the receive window limitation: the room left in the send buffer is checked, just like the room left in the receive window was in the previous chapter. This check is again performed at the time of a *peak* (as defined in Section 4.5) and that way we count the number of send buffer limited peaks. The percentage of peaks that are send buffer limited can then be used as a measure for detecting the send buffer limitation, just like we used the percentage of receive window limited peaks in the previous chapter.

However, some extra problems arise, because we do not know the exact size of the send buffer. Section 5.4 will deal with estimating this size. Additional problems arise when the estimation of the send buffer is increased. After all, what should be done with all the previous peaks that were counted as being send buffer limited based on the previous estimation of the send buffer? Section 5.5 describes an advanced yet efficient solution.

5.2 A send buffer limited flow

An illustration of the send buffer limitation is given in Figure 5.1. Although the receiver allows the sender to send one hundred bytes, only 60 bytes are sent in each 'block'. Apparently the sender is not able to send any more data. However, because we can only *guess* the size of the send buffer, it is difficult to conclude a send buffer limitation with certainty.

The displayed flow could just as well have a huge send buffer, but not send any faster because it contains streaming audio that is transmitted in packets of 60 bytes per say 50 milliseconds. Although this ambiguity will not be solved yet in this chapter, the detection of this limitation in Chapter 7 will provide a solution.

5.3 Detecting the send buffer limitation

Because the send buffer limitation looks almost exactly like the receive window limitation, detection can also be performed in practically the same way. The time at which the send buffer limitation can be identified is also at a peak of the number of outstanding bytes, as explained in Section 4.5 (the same reasoning applies as was the case for the receive window). Therefore, the framework provided

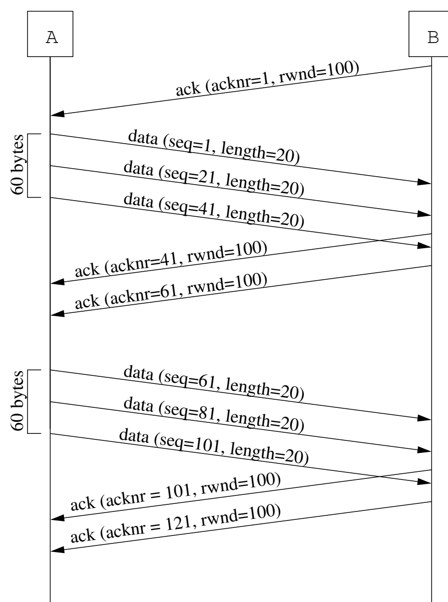


Figure 5.1: *Send buffer limitation*

for the receive window limitation detection can be reused. Also, the same kind of utilizations occur. Just like the receive buffer is sometimes filled completely, sometimes with an integer multiple of the MSS and sometimes with blocks, so is the send buffer. The same detection methods as derived in Chapter 4 can thus also be used. In this chapter the utilizations with respect to the send buffer will be called block-based buffer utilization, integer MSS buffer utilization and complete buffer utilization.

Of course one minor adjustment will have to be made. In case of the framework, up to how far the sender is allowed to send will now have a slightly different meaning. Instead of how far the sender is allowed to send as dictated by the receive window, we now keep in mind how far the sender is 'allowed' to send by the send buffer. Because we do not know the real size of the send buffer, an estimation will be used. In case the estimation of the send buffer is increased, some previously counted send buffer limited peaks are to be discarded (excluded from the count). After all, these previous calculations were made using a wrong (too low) send buffer estimation.

The difficulty is left in estimating the send buffer size and deciding which peaks need to be discarded; these problems are covered in the next two sections.

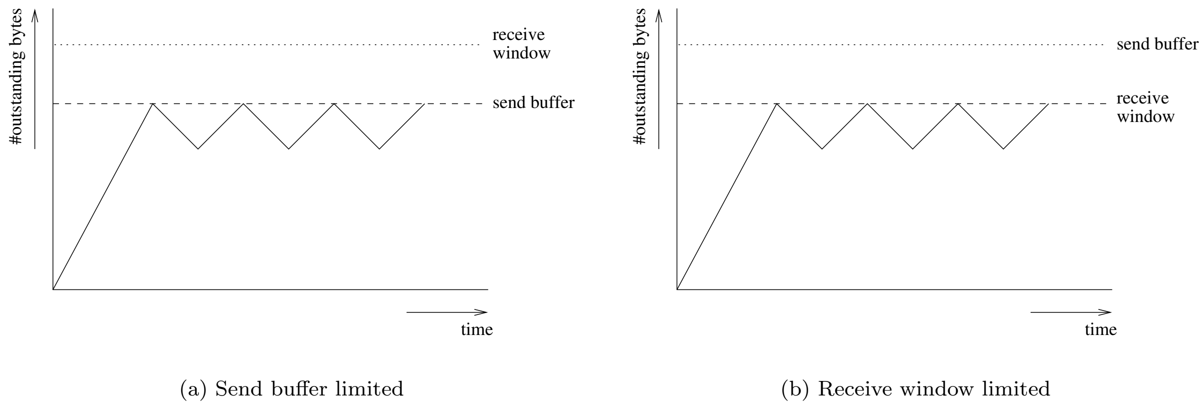
5.4 Estimating the send buffer size

As described in the introduction, no mechanism in the TCP specification can be used to get to know the size of the send buffer. However, it is possible to make an estimation by examining the amount of outstanding data. As we will see, a lot of difficulties arise because of measurement errors and the different utilization schemes.

As mentioned, a flow cannot have more outstanding data than there is space in the send buffer. In the case of a send buffer limitation, we will therefore observe an upper limit on the number of outstanding bytes, comparable to what we would see at a receive window limited flow. Figure 5.2a gives an illustration of the characteristic behaviour. We observe the send buffer as being the limit up to how far the sender increases its number of outstanding bytes. The receive window resides somewhere above the send buffer and therefore is not of influence on the flow.

5.4.1 Basic criterion

Estimating the send buffer simply comes down to keeping track of the maximum number of outstanding bytes. We must however keep in mind that a send buffer size estimation based on the maximum number of outstanding bytes will not always result in the size of the send buffer, but sometimes in the size of the receive window. After all, when we look at a receive window limited flow (see Figure 5.2b), the exact same behaviour is shown. Therefore, a receive window limitation should always be checked



(a) Send buffer limited

(b) Receive window limited

Figure 5.2: *Diagrams of send buffer and receive window limited TCP flows*

prior to checking for a sending buffer limitation. After all, the correct value of the receive window *is* known.

5.4.2 Additional criteria due to measurement errors

As mentioned in Section 3.4.2, there have been some problems with logging all the headers. As a result of this, fake gaps occur in the repository. As we can see from Figure 4.6 in Section 4.5.1, the number of outstanding bytes can therefore appear to be higher than it is in reality. This also results in problems for the send buffer estimation, because a too large number of outstanding bytes would increase the estimation of the send buffer incorrectly.

Almost the same criterion can be used as was the case for the receive window checks. However, instead of waiting for the second normal acknowledgment after a gap, retransmission or duplicate acknowledgment, only waiting for the first normal acknowledgment suffices. After all, once a normal acknowledgment has been seen the calculation of the number of outstanding bytes will be correct again.

Furthermore, some gaps are not detected at all, for example if only an acknowledgment gets lost. Therefore, it is advisable not to increase the estimated value of the send buffer immediately when the number of outstanding bytes exceeds it, but to wait until the number of times this happens has passed a certain threshold. We have used the value $\max(5, \frac{numberOfPacketsSoFar}{100})$; further research could perhaps establish a more optimal value. Of course for a repository without any measurement errors this detection scheme produces some unnecessary distortion, but not using it for our faulty repository would result in too large send buffer estimations (and therefore completely wrong conclusions).

Figure 5.3 shows an algorithmic description for estimating the send buffer size, based on these criteria. The next section will cover criteria for excluding previously detected send buffer limited peaks from the count in case of an estimation update.

5.5 Dealing with send buffer size estimation updates

With block-based buffer utilization and integer MSS buffer utilization (explained in Section 4.6.2 and Section 4.6.3), some problems arise with deciding which previously detected send buffer limited peaks are still to be counted with respect to the current estimation of the send buffer size. Just discarding all previously detected peaks would seem logical, but does not always produce the best results. The problems are similar for both block-based buffer utilization and integer MSS buffer utilization. In this section we will explain the issues using the block-based buffer utilization. Figure 5.4 gives an illustration of an example flow in which these problems arise. By substituting MSS for block size, all the arguments also hold for the integer MSS buffer utilization.

```
For each packet:  {
      if (this packet is the first one after a gap) {

           waitForAckAfterGap := true

      }
      else if (this packet is a DupACK or a retransmission) {

           waitForAckAfterGap := true

      }
      else if (this packet is an ACK packet) {

           waitForAckAfterGap := false

      }

      if (waitForAckAfterGap == false && numberOfOutstandingBytes > estimatedSendBuffer)
      {

           numberOfTimesLargerThanSB++
           if (numberOfTimesLargerThanSB > max(5, numberOfPacketsSoFar/100)) {

               if necessary decrease counter of send buffer peaks
               estimatedSendBuffer := numberOfOutstandingBytes
               numberOfTimesLargerThanSB := 0

           }

      }
}
```

Figure 5.3: *Algorithm for estimating the send buffer size*

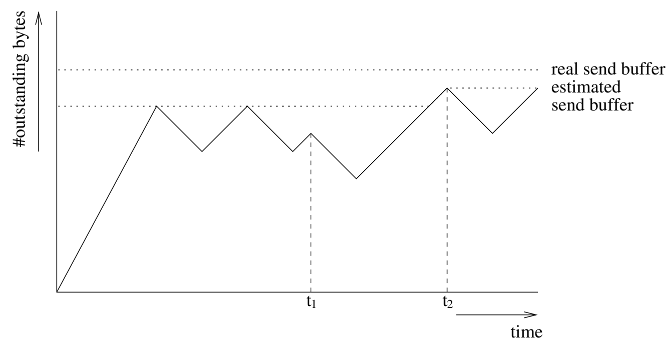


Figure 5.4: *Send buffer limitation*

We can observe how the sender at first only sends up to a certain amount of outstanding bytes (which will be the estimated send buffer in that period), because of block-based buffer utilization. However, for some reason a small packet has been sent at some moment in time (t_1). This has resulted in a shift in the sending pattern, through which the send buffer can be filled better. So, the estimation of the send buffer increases at t_2. This however does not mean that the peaks in the first part of the example were not send buffer limited. Therefore, only counting the send buffer limited peaks that occurred after the last increase of the send buffer size estimation is not always the right choice.

As we have seen, block-based buffer utilization results in flows that do not send up to the send buffer limit, but stay below it. An increase could then just be a more efficient filling of the send buffer because of a shift in the sending pattern. We should therefore take into account how far the send buffer may rise before we dismiss the results obtained thus far. Of course for a flow with block-based buffer utilization this amount can differ from a flow that uses integer MSS buffer utilization.

Providing criteria that result in the best possible conclusions would require an implementation to keep track of the heights of all peaks or go through the repository twice. Figure 5.5 gives an example that supports this statement. Illustrated is a flow that is expected to be send buffer limited. With the send buffer estimated at the height of the thick line, the first peak is of course considered send buffer limited.

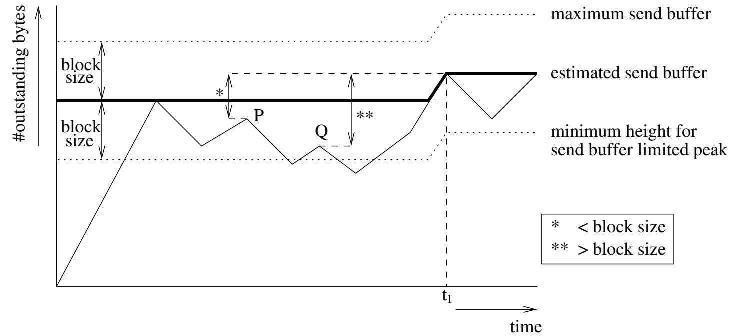


Figure 5.5: *Send buffer limitation*

Peak P will also be considered send buffer limited, because we saw earlier that the block-based buffer utilization allows for a small remainder of space in the buffer (as long as it does not exceed the block size). The lower dotted line visualizes the bottom limit of peak heights that based on the current estimation of the send buffer will be considered send buffer limited.

Now consider the change in the estimation of the send buffer size at t_1. The increase is less than the block size, so the first peak can still be considered a send buffer limitation incorporating the new size estimation. Peak P is also still within a block size below the new estimation, so also this peak is still considered a send buffer limitation. However, peak Q (that was detected as a send buffer limited peak with respect to the old send buffer estimation) is now more than a block size below the new estimation. Therefore, this peak does not comply with the criteria for a send buffer limited peak anymore and should be excluded from the send buffer limited peaks count.

At the time the send buffer size estimation is updated, however, the exact height of the preceding peaks will not be known anymore in a memory efficient implementation. Therefore, we cannot know whether or not P or Q were positioned above or below the new minimum height for send buffer limited peaks, without going through the repository twice. To support an efficient implementation, we have chosen to develop a solution which can be implemented without having to go through the repository twice and without having to save a lot of data in memory.

5.5.1 A solution for approximating correctness

Our main concern with developing criteria for selecting which preceding peaks to continue taking into account as send buffer limited peaks in case of an increase of the send buffer size estimation was to stay on the 'safe side'; we rather incorrectly do not count peaks that were in fact send buffer limited, than incorrectly do count peaks that were not. To achieve this, only previous peaks of which we are sure they also are send buffer limited with respect to the new send buffer size estimation are taken into account.

To overcome the problem of not knowing the height of all the preceding peaks, we divide all the previously detected send buffer limited peaks (that are still expected to be real send buffer limited peaks considering the current estimation of the send buffer, which could be higher than the estimation at the time of the peak) into two categories: peaks that were equal to the current estimated send buffer size (the 'max category') and peaks that were smaller (the 'non-max category'). Furthermore, we keep track of the lowest accepted send buffer limited peak for the current estimated send buffer; the lower boundary of the second category just described. For example, in the flow of Figure 5.5 the lowest accepted peak prior to the update of the send buffer estimation is Q. After all, Q was detected as a send buffer limited peak with respect to the estimated send buffer size prior to the update and no other send buffer limited peak lower than Q has been detected for this send buffer size (even though this would have been possible, because Q still lies above the minimum height for send buffer limited peaks).

When an update for the send buffer estimation takes place, three situations can occur:

- Firstly, it is possible that the new estimation is more than a block size larger than the previous estimation. In this case, all previous peaks cannot have been send buffer limited, so they are discarded. No peaks belong to the max and non-max categories anymore. See Figure 5.7a.
- Secondly, it is possible that the new estimation is less than a block size larger than the lowest accepted send buffer limited peak. In this case, all the previously accepted peaks are still valid for the new estimation. All the peaks that prior to the update belonged to the max category will now be transferred to the non-max category, because the old maximum is not the maximum anymore. See Figure 5.7b.
- Thirdly, it is possible that the new estimation is less than a block size larger than the previous estimation, but more than a block size larger than the lowest accepted send buffer limited peak. Now all the peaks in the max category are still valid, but the peaks in the non-max category should be discarded. After all, we cannot guarantee anymore that all of these peaks are send buffer limited. The old max category will become the new non-max category and the old non-max peaks are discarded. See Figure 5.7c.

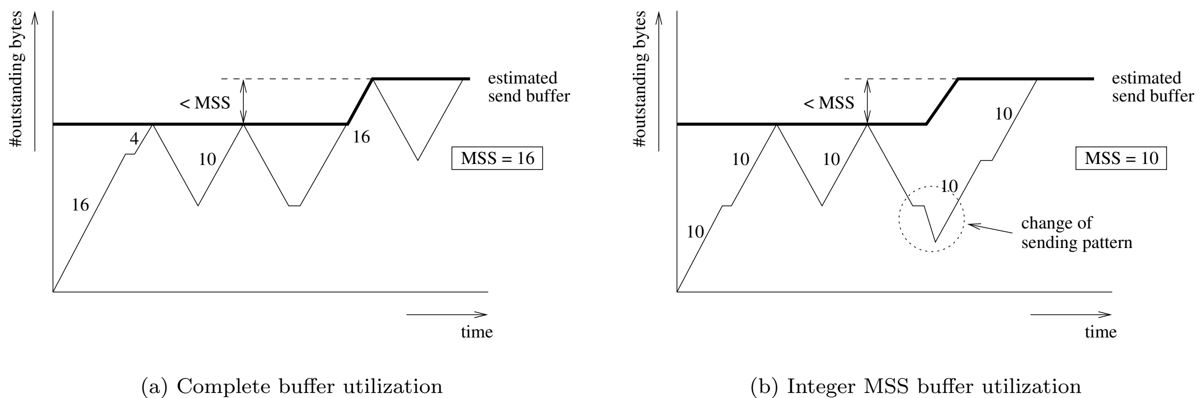
Using these criteria, after each update we will only consider the peaks that still look send buffer limited in the new circumstances. Besides, when some previous peaks were just a little below the then-estimated send buffer, they will still be considered when the distance between the lowest of those peaks and the new estimation is less than a block size.

The only scenario that results in a too low send buffer limitation percentage is when the lowest accepted send buffer limited peak is more than a block size below the new estimated send buffer size, while there were some other peaks in the non-max category that *would* still be considered send buffer limited with respect to the new estimation. Therefore, correctness is only approximated and not completely achieved.

5.5.2 Distinguishing between the different buffer utilizations

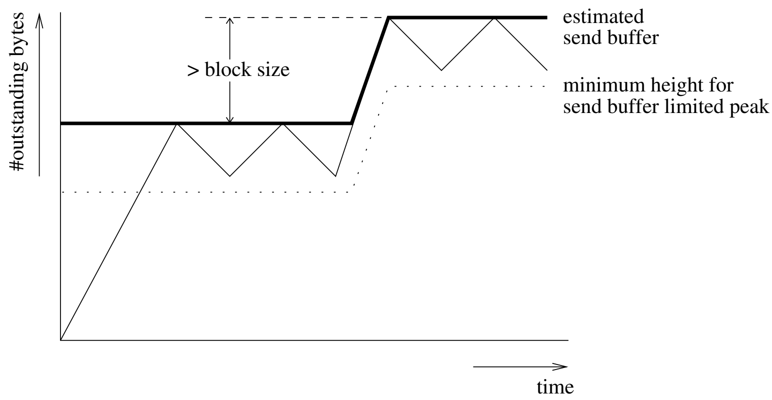
In the previous section a slightly simplified version of the criteria for counting the send buffer peaks with respect to estimation updates has been given. So far no distinction between peaks of the different utilizations have been made. However, as it turns out, this distinction is necessary.

Consider the example flows illustrated in Figure 5.6. Both flows first have two send buffer limited peaks up to 20 bytes outstanding data. Then, a send buffer estimation update takes place. In both flows, the increase is less than the MSS. The flow of Figure 5.6b will after the update therefore have to keep taking into account the first two peaks, because these peaks still appear to be send buffer limited (because of the integer MSS buffer utilization principle – recognized by the equality of the size of the last packet of a peak and the MSS – the send buffer could not be filled completely). The send buffer limited peaks count of the flow in Figure 5.6a, however, should discard the first two peaks. After all,

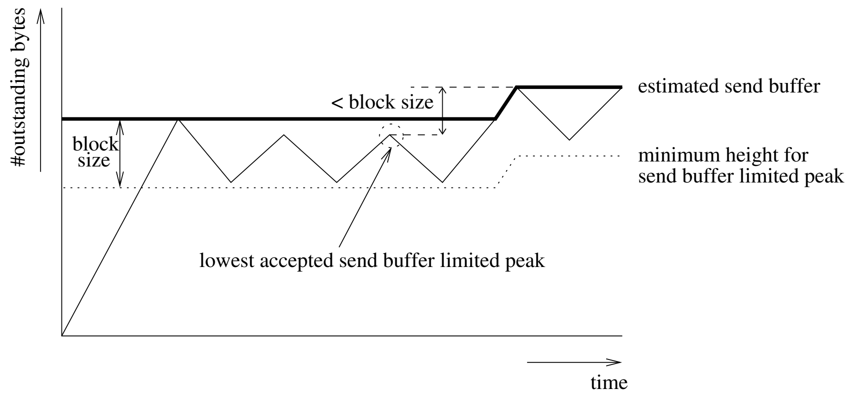


(a) Complete buffer utilization (b) Integer MSS buffer utilization

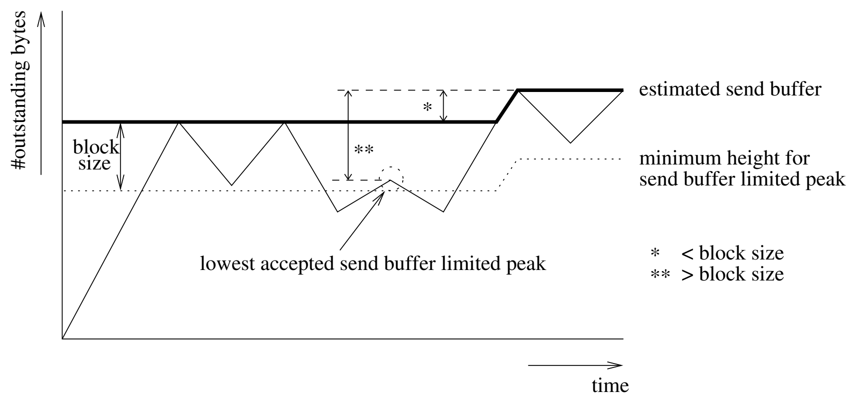
Figure 5.6: *Different window utilizations*



(a) Scenario 1



(b) Scenario 2



(c) Scenario 3

Figure 5.7: *Send buffer estimation update scenarios*

these peaks were not categorized as integer MSS buffer utilization, but are clearly an example of the complete buffer utilization (because the flow is not block based and the last packet of the peak was smaller than the MSS). Therefore, the first peak could have been higher if the send buffer was the limiting factor, because the flow is not constrained by the integer MSS buffer utilization restrictions.

This example shows that it is not only necessary to keep track of the max and non-max categories, but also to make a distinction between the different buffer utilizations. Therefore, we propose to divide the peaks that are seen thus far and are still valid send buffer limited peaks with respect to the current estimation of the send buffer size, into five categories: max block-based, non-max block-based, max integer MSS, non-max integer MSS and complete buffer. Of course the complete buffer utilization has no non-max category, because only peaks that are of the same height as the estimated send buffer are detected as complete buffer utilization. Besides this subcategorization, also the lowest accepted send buffer limited peak should be divided into two new quantities: the lowest accepted send buffer limited peak of block-based peaks and the lowest accepted send buffer limited peak of integer MSS peaks. The complete buffer utilization does not have such a quantity; the estimation of the send buffer *is* by definition the lowest accepted send buffer limited peak of the complete buffer utilization.

The detection of the send buffer limitation based on these categories works as follows. For each peak we detect in which of the five categories it falls (it is of course also possible that it does not belong to any category). Each utilization is associated with a certain value, which we will call the *update threshold*. This value denotes the size of the blocks or packets that are send as a whole, and is used to decide which previous peaks to keep taking into account and which to discard. The block-based buffer utilization is explained in Section 5.5.1 and uses an update threshold equal to the block size. The update threshold of the integer MSS buffer utilization is equal to the MSS, as follows directly from the definition in Chapter 4. The complete buffer utilization could also be seen as having an update threshold, with the value of the threshold set to zero.

The categories have been made disjoint and a peak that could belong to multiple categories is placed in the category with the highest priority. The priority sequence from high to low is block based, integer MSS and complete buffer; this way, a peak will always be placed in the category with the highest update threshold. Because of the priorities each send buffer limited peak will always be contained in only one category, so the sum of the peaks in all categories will always be equal to the number of send buffer limited peaks.

Now, when a send buffer size estimation update takes place, a better update of the thus far detected send buffer limited peaks can be made. The three utilizations can be updated independently of each other to conform to the new situation. The criteria for the integer MSS and complete buffer categories are almost exactly equal to the block-based category, so all of the theory developed in Section 5.5.1 can be applied. The only difference is in the update threshold; for the detailed discussion of the block-based buffer utilization the block size is used to decide what should happen, while for the integer MSS and complete buffer utilizations their own update thresholds should be used. In case of the complete buffer utilization the first situation will always apply, because the increase of the send buffer size estimation will always be more than the update threshold (which is zero).

Chapter 6

Recognizing the Network Limitation

6.1 Introduction

In this chapter criteria will be developed for detecting a network limitation: due to limited network capabilities packet loss occurs and the TCP congestion control mechanism limits the sending speed. Because different TCP implementations apply different congestion control mechanisms [MLAW99], it is not feasibly possible to estimate the congestion window at a specific time. Therefore, we can also not identify the network limitation at a specific time and thus cannot express the limitation as the percentage of peaks that are network limited (as was done with the receive window and send buffer limitation). What *can* be done is to express the actual bandwidth of a flow as a percentage of the maximum achievable bandwidth. For this chapter, the maximum achievable bandwidth is defined as the maximum bandwidth that can be achieved on the network path under the restrictions of the TCP congestion control mechanism, and is estimated by the TCP Friendly formula. Of course we are dealing with an approximation. When the percentage approaches one hundred percent, the network is likely to be the limiting factor.

Section 6.2 will give some information about the TCP Friendly formula and how it can be used to detect the network limitation. As we will see, to use the formula, certain quantities about a flow have to be known: the loss percentage and the round-trip time. Section 6.3 will deal with estimating the loss percentage, while Section 6.4 will explain the estimation of the round-trip time. Finally, Section 6.5 will put everything together.

6.2 TCP Friendly formula

The TCP Friendly formula (derived in [MSM97]) is mostly used to prescribe the amount of bandwidth a UDP flow is allowed to send. Using the value given by the formula as an upper limit for UDP flows, will result in a fair distribution of bandwidth between the two transport protocols.

In our research the formula is used for a completely different reason. The approximation it gives of the maximum achievable bandwidth can be used as a norm to which we can compare the actual bandwidth used. Expressed as a percentage, this will be called the *bandwidth percentage*. When this percentage approaches one hundred percent, the network is likely to be the limiting factor. Because the TCP Friendly formula is only an approximation and because flows limited by other phenomena can also utilize a large percentage of the maximum achievable bandwidth, however, a high bandwidth percentage does not necessarily mean the network is the limiting factor. Chapter 8 will provide a more in-depth discussion on the interpretation of the results.

The TCP Friendly formula is given by equation 6.1. The formula is based on a lot of assumptions and is therefore just a rough approximation. Details can be found in [Kla05].

Mostly the value $\sqrt{3/2}$ is chosen for c, but according to [PFTK98], the formula then does not take into account delayed acknowledgments, even though most TCP implementations seem to be implementing this mechanism [Kla05]. [PFTK98] proposes a constant of $\sqrt{3/4}$ instead of $\sqrt{3/2}$ when delayed acknowledgments are implemented. In our analysis this value will be used. Of course the

$$BW = \frac{MSS}{RTT} \frac{c}{\sqrt{p}} \tag{6.1}$$

BW	the maximum achievable bandwidth
MSS	the maximum segment size
RTT	the round-trip time
p	the loss rate
c	a constant

resulting value of BW for flows that do *not* implement delayed acknowledgments will be a factor $\sqrt{2}$ too small.

The loss rate p can be calculated in two ways: by calculating the number of *retransmissions* or by calculating the number of *loss events*, and expressing this value as a ratio of the total number of packets. As it turns out, the exact number of retransmissions is not very interesting for detecting the network limitation. What we *are* interested in, is the number of times the congestion window is halved (after all, that is what influences the sending speed). We will now formally define loss events and explain why they are more important than the number of retransmissions.

Figure 6.1 illustrates the well-known TCP saw tooth. When the network is the limiting speed factor — as is the case in this example — the TCP congestion control mechanism will keep increasing the number of outstanding bytes until a packet is lost. Then, the number of outstanding bytes allowed (the *congestion window*) is halved, and the increasing phase starts again. Whether only one or maybe two or three packets got lost on the top of the "tooth" is not that important, because in both cases the congestion window is halved only once. Therefore, instead of using the number of retransmissions, we look at the number of *loss events* for our estimation of the loss percentage. A loss event is defined as an event that causes the congestion window to halve and can, for example, be the loss of one packet but also the loss of two packets. The loss events in the example of Figure 6.1 are indicated by dotted circles.

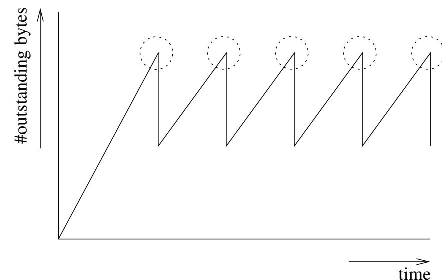


Figure 6.1: *Loss events*

6.3 Estimating the loss rate

One of the quantities essential to the TCP Friendly formula is the loss rate. In order to be able to estimate the loss rate, loss events have to be detected. Difficulties arise, because there are cases where a loss event looks exactly the same as a reordering event. Some effort has been made to develop criteria for distinguishing between the two, but — provided it is even possible at all — more research will be necessary.

However, when we are measuring at the sending side (as indicated in Section 3.5), the chances are very small that reordering occurs between the sender and the measurement point, so each packet with a not-expected sequence number (more precisely defined later on) can with quite some certainty be considered a retransmission.

We will first look at a way to count retransmissions. This will be used to detect loss events later on. Then, our method will be compared to a more intuitive version, to show why we have chosen the slightly more difficult version. Finally, the limitations of the developed method are discussed.

6.3.1 Counting retransmission

The criterion for detecting a retransmission is simple: when a packet has a sequence number that is lower than the highest next sequence number expected (as defined in Chapter 4), it is considered a retransmission. Such a sequence number is called a not-expected sequence number.

To understand the mechanism, see Figure 6.2. Two packets get lost, and more data is following. After a while, the TCP entity on the sending side detects the loss because of three duplicate acknowledgments. As a response, a packet is retransmitted. At the time we observe the retransmitted

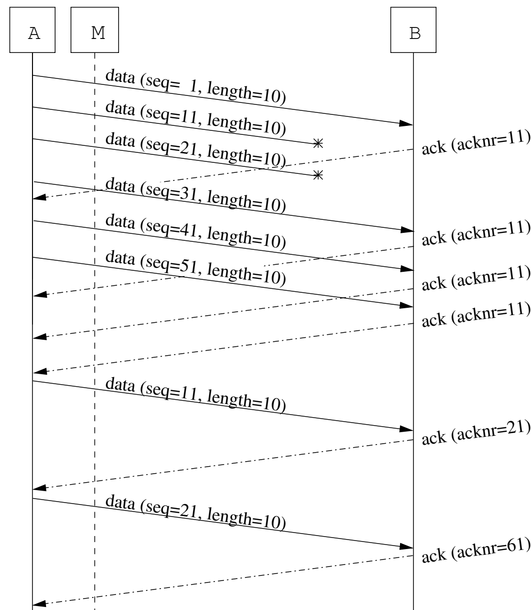


Figure 6.2: *Counting retransmissions*

packet, we have already seen a packet with sequence number 51 and length 10. Therefore, at this moment the highest sequence number expected is equal to 61. Instead of a packet with sequence number 61 we observe the retransmitted packet with sequence number 11. Because 11 is smaller than 51, the packet will correctly be counted as a retransmission. Also the next packet, with sequence number 21, is and will be detected as a retransmission. After all, the highest sequence number expected is still 61 and 21 is also smaller than 61.

6.3.2 Counting loss events

A loss event can be detected by only counting the first retransmission of a retransmission "burst". So, when multiple retransmissions occur after each other, only the first one is counted. The exact criterion used is as follows: when the first retransmission occurs, we keep track of the highest next sequence number expected at that moment. After that, all retransmissions with a sequence number lower than or equal to this one will not be counted as loss events. When a retransmission occurs with a sequence number higher than the highest next sequence number expected recorded at the time of the last loss event, this is considered a new loss event and the highest next sequence number expected is updated to the current value.

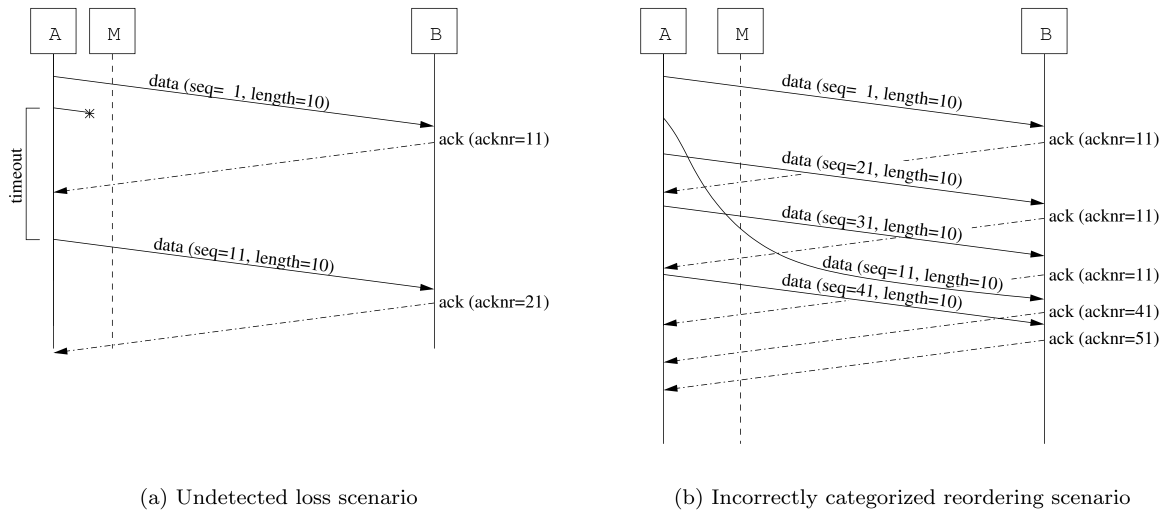
Although there could be scenarios where this results in not counting all loss events (when retransmissions get lost), it will produce correct results in most cases. We will get a lower limit on the number of loss event, and again (just like with the send buffer detection) provide a 'safe' detection that will not incorrectly detect too many loss events.

Notice that a packet with a sequence number *higher* than the highest expected next sequence number does not get extra attention compared to a packet with a sequence number that is equal to it. It indicates a *'gap'*; a packet got lost between the sender and the measurement point. In this case, the lost packet will be detected later on when the gap is filled. Moreover, a fortunate consequence is that "fake gaps" (see Section 3.4.2) will not be counted in our loss percentage, because these gaps are never filled.

Our detection mechanism compared to a simpler version

To most people, defining the end of a retransmission burst as a data packet which *is not* a retransmission, would probably be more intuitive than our mechanism. However, some problems arise when using this method.

Table 6.1 illustrates data observed in a real flow in the repository, which will be used to compare our mechanism to the simpler version. Normally one would expect to see a flow send up to a certain



(a) Undetected loss scenario

(b) Incorrectly categorized reordering scenario

Figure 6.3: *Flaws with the loss event detection*

top, lose one or more packets, start retransmitting the lost packets and then continue sending from the point where the loss occurred. This example, however, shows that there can be other scenarios. Here, during the retransmission of the lost packets new packets are also sent. If we would define the end of a retransmission burst as a data packet which *is not* a retransmission, packet 3 would indicate the end of a burst. Therefore, the given scenario would be counted as two loss events, even though the congestion window has been decreased only once.

With our little more complicated detection mechanism of keeping track of the highest next sequence number expected, no such problems would occur. At the time packet 3 is detected as the start of a loss event, the highest next sequence number expected is 24121 ($23585 + 536$). When the next retransmission occurs at packet 5, its sequence number (18225) is smaller than the highest next sequence number expected at the time of the start of the last loss event (24121). Therefore, packet 5 will not also be counted as a loss event.

1. data (seqnr=23585, length=536)
2. data (seqnr=**17153**, length=536)
3. data (seqnr=24121, length=536)
4. data (seqnr=**18225**, length=536)
5. data (seqnr=24657, length=536)
6. data (seqnr=25193, length=536)

Table 6.1: *An observed packet sequence*

Flaws in the loss event detection

Unfortunately, not all loss events are recognizable using the aforementioned criterion. In case a packet gets lost in the area between the sender and the measurement point and no more packets are sent before the timeout (see Figure 6.3a), it is impossible to detect a loss event. However, since the sender does not send packets even though it is allowed to, the chance is small that the flow is network limited. Moreover, we are measuring close to the sender, so this scenario is unlikely to occur.

A second flaw occurs when reordering takes places in the area between the sender and the measurement point, as illustrated in Figure 6.3b. In that case, our methodology would detect a loss event, even though the congestion window will not be halved (after all, the reordered packet arrives before three duplicate acknowledgments would be sent). Because reordering is scarce and the distance between the sender and the measurement point is small, however, this scenario is unlikely to occur.

6.4 Estimating the round-trip time

The second quantity essential to the TCP Friendly formula is the round-trip time (RTT). This section will explain why measuring the RTT at the *sending* side is essential. Then, a basic estimation methodology will be presented. First a simplified version is presented, together with evidence for the need of a little more complicated method. Finally, this methodology is extended even more, to improve its capability to deal with RTT estimations tainted with retransmissions.

Basic RTT estimation

The most natural way to estimate the RTT is by looking at the time difference between sending a data packet and receiving an acknowledgment for it. Figure 6.4 illustrates the estimation of a single RTT. The estimated value will always be below the actual value, because the measurement point is not located at the sender. The closer we get to the receiver, the higher the difference between the estimation and the actual value of the RTT will be. Some ideas have been proposed for an RTT estimation that works for every measurement point [VLL05], but further research is necessary before this can be incorporated in our work. Because we are measuring close to the sender (see Section 3.5), we assume the distance between the sender and the measurement point to be zero.

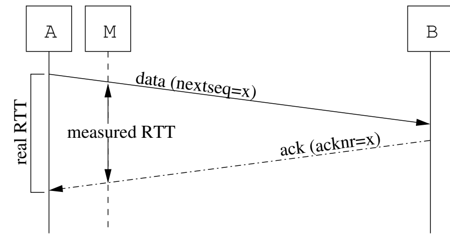


Figure 6.4: *Estimating the RTT*

Detailed RTT estimation methodology

Our way of estimating the RTT is to estimate it a number of times — the different estimations are not overlapping in time — and take the average at the end. Estimating the RTT is performed by calculating the difference between the time an acknowledgment arrives and the time the corresponding data packet was sent.

One might expect that the easiest way of doing this is by keeping track of the time of one data packet and start waiting. When the corresponding acknowledgment arrives, calculate the time difference. Then, start the cycle again for the next data packet. In order to prevent waiting infinitely long, stop waiting and start the cycle again for the next data packet, when an acknowledgment has been received for a packet sent after the packet of which the timestamp has been remembered.

Although the idea of this methodology is correct, Figure 6.5 illustrates why at least two timestamps should be remembered. First, we keep track of time P, when the first data packet is sent. Later, at time Q, an acknowledgment for another data packet arrives (because of delayed acknowledgments). The acknowledgment number indicates that a packet sent after the packet for which the timestamp has been remembered is acknowledged, so the timestamp will not be used anymore. At time R, the next data packet is observed and that time will be used for the next RTT estimation. When the next acknowledgment is seen, however, it is again for a data packet sent after the packet of which we recorded the time. Therefore, again no estimation can be made. Of course this scenario could go on for an entire flow, which would result in not being able to estimate the RTT.

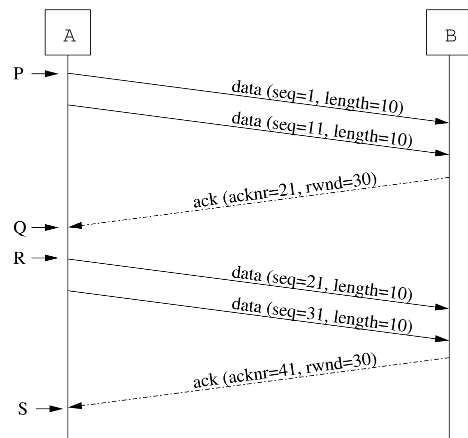


Figure 6.5: *RTT and DupACKs*

As a solution, not one but two timestamps are kept track of. For the example of Figure 6.5, not only the timestamp of the first but also of the second data packet will be used. Using this methodology, the chance of getting an acknowledgment for a timestamped data packet is very large. After an RTT estimation has been performed, both timestamps will not be used anymore and the next two data packets are to be timestamped. This way, the number of RTT estimations will approximately be equal to the duration of the flow divided by the actual RTT.

Of course keeping track of every timestamp would result in a better RTT estimation, but would also consume more memory in an implementation. Further research could be done to identify the advantages and disadvantages.

An implementation should keep in mind that TCP uses sequence number wrapping. If it does not, an acknowledgment of a packet with a sequence number close to the wrapping boundary and a length that causes it to pass that boundary would not be recognized correctly.

Dealing with retransmissions

Retransmissions can result in bad RTT estimations, if not handled with care. Figure 6.6a illustrates the problem when the timestamped data packet gets lost and retransmitted. The methodology developed so far just looks at the time difference between a data packet and its acknowledgment and starts over

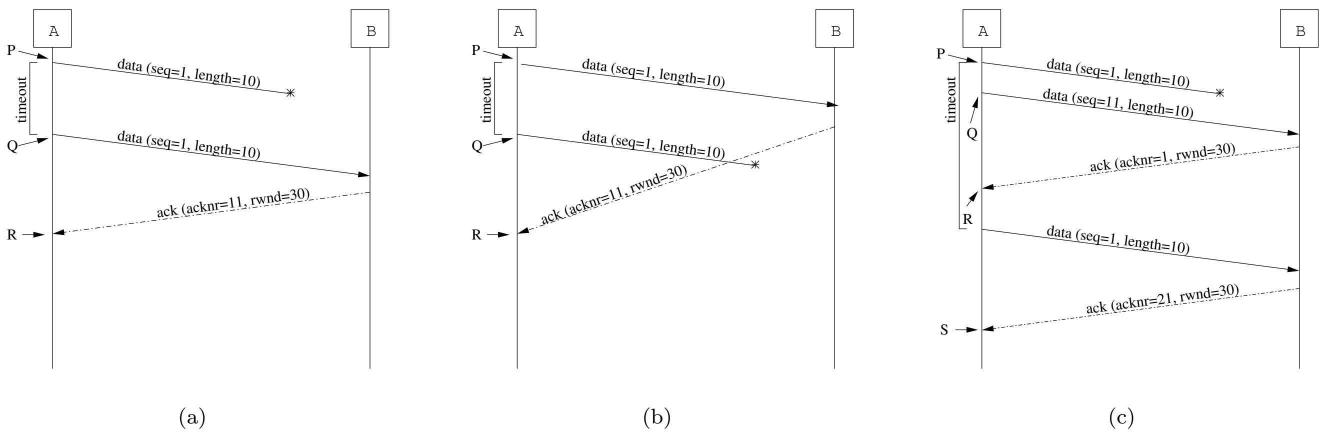


Figure 6.6: *RTT estimation with loss*

when an acknowledgment of a data packet sent after the timestamped packet arrives. In the scenario illustrated in Figure 6.6a, this would result in an RTT of $R-P$. Of course this is not the actual RTT, which is equal to $R-Q$.

Not only the round-trip time measurement of a lost packet can fail; also the measurement of a retransmission can be incorrect. Figure 6.6b illustrates a scenario where this happens. When the retransmission is timestamped, a round-trip time measurement of $R-Q$ will result, even though the actual RTT is equal to $R-P$.

Besides lost packets and retransmissions of lost packet, even a retransmission of a data packet sent earlier than a timestamped packet can cause problems. Figure 6.6c shows an example in which this happens. Assume that the timestamps P and Q were to be used in an RTT estimation. Because the data packet sent first was lost, an acknowledgment packet indicating this lost packet should still be sent is observed. This acknowledgment acknowledges a data packet sent prior to P and Q, so no estimation can be made. Later on, when the second acknowledgment is observed, an estimation of $S-Q$ would be made; clearly incorrect.

As indicated by the Requirements for Internet Hosts [Bra89], each TCP implementation should use Karn's algorithm for RTT estimations [KP95]. This algorithm just ignores round-trip time measurements based on packets seen more than once. That way, the problems indicated by Figure 6.6a and Figure 6.6b are solved. However, Figure 6.6c shows that also a retransmission of a previous packet can cause problems: problems that are not addressed by the TCP standard. In order to achieve valid RTT estimations, we therefore ignore not only the round-trip time measurements for retransmitted packets but also the measurements for packets that were sent between a previously sent packet and its retransmission.

More formally the mechanism can be described as follows. When a retransmission is observed, round-trip time measurements of all packets seen previously for which an acknowledgment has not been seen yet and with a sequence number equal to or higher than the sequence number of the retransmission, will not be used anymore. After all, these are the packets that are influenced by the retransmitted packet; packets with a sequence number smaller than the retransmitted packet have nothing to do with it, so their round-trip time measurements can still be taken into account.

This way fewer estimations are performed, but the described problems will cease to occur and our RTT estimation gives a good approximation of the actual RTT value.

6.5 Putting it all together

The previous sections have provided mechanisms for estimating the round-trip time and the loss rate. Now, the TCP Friendly formula can be used to calculate the maximum achievable bandwidth. Furthermore, the actual bandwidth can easily be calculated by dividing the number of bytes transmitted by the number of seconds between the first and the last packet of the flow. Retransmissions should not be taken into account, so an implementation can just subtract the sequence number of the first

data packet from the acknowledgment number of the last ack packet (although in case of the wrapping of sequence numbers one should subtract the sequence number of the first data packet from 2^{32} and add the acknowledgment number of the last ack packet).

Expressing the actual bandwidth as a percentage of the maximum achievable bandwidth calculated with the formula then results in the bandwidth percentage.

Chapter 7

Recognizing the Application Layer Limitation

7.1 Introduction

After we have discussed all the limiting TCP speed factors regarding the TCP layer and the network layer, the application layer remains. Manual examination of some flows has shown that the application layer indeed does sometimes limit the speed of a flow.

Two different application layer limitations can be identified: sometimes the application layer just does not provide data fast enough (resulting in periods with zero outstanding bytes) and sometimes the application appears to have its own kind of acknowledgment or request mechanism (resulting in a limited number of outstanding bytes the application layer will provide to the TCP layer).

Section 7.2 will discuss the first kind of limitation, Section 7.3 will discuss the second.

7.2 Lack of data

The most natural application layer limitation is a lack of data. An example of a flow that is limited by this variant is given in Figure 7.1.

The figure shows one of the possible scenarios that result in what we call the lack of data scenario. The flow represents a streaming audio application that sends its data in blocks of 600 bytes, one block every 50 milliseconds. As we can see, the flow would be able to send more data if it would be available. Therefore, the lack of data limits the speed of the flow.

Another scenario for which a lack of data limits a flow is an interactive application, like an instant messenger or an IRC session. In those cases, data is only available when the user types something. Most of the time the network (and the protocol stack) would be able to process the data faster than the user provides it, so also in this scenario the lack of data limits the speed of the TCP flow.

We will first develop a basic detection mechanism, then extend it to prevent the receive window from corrupting the results. Finally, we will discuss the problems that arise because of the existence of fake gaps and consider the location of the measurement point.

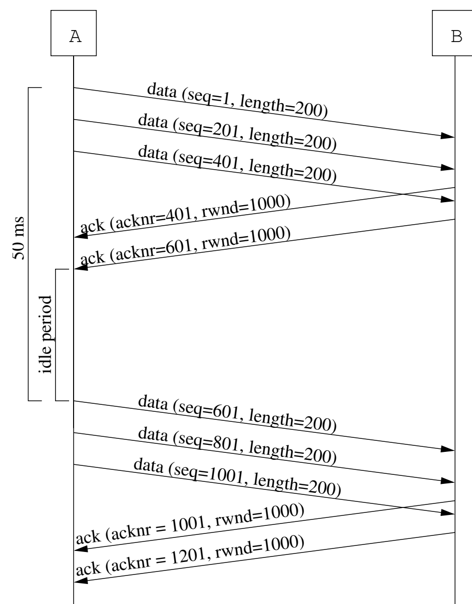


Figure 7.1: *Lack of data*

Basic detection methodology

The definition of the lack of data scenario directly leads to a basic detection criterion: calculate the percentage of time that the number of outstanding bytes is equal to zero. A period of time in which the number of outstanding bytes is equal to zero will be

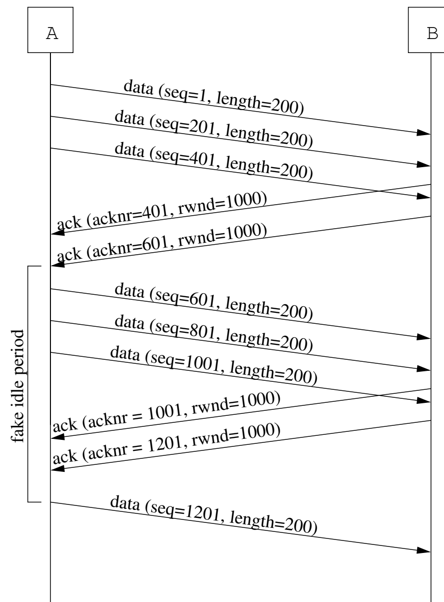


Figure 7.2: *Flow with fake gap*

called an *idle period*. Formally, the length of an idle period is defined as the time between receiving an acknowledgment that acknowledges all previously sent data and the first data packet sent after it. This is all seen from the point of view of the sender. A flow with enough data provided by the application layer will almost all the time have a nonzero amount of outstanding bytes, because receiving an acknowledgement directly results in sending new data. Therefore, even when only 1 percent of the time there is no outstanding data, this would still imply a lack of data limitation.

Manual examination of some flows indicated that a lot of flows show idle periods at the beginning, even though the rest of these flows does not contain any. Therefore, we have chosen to only consider idle periods appearing from the fifth packet.

Improving detection: taking into account the receive window

Even when all previously sent bytes are acknowledged, it is still possible that the sending TCP entity is not allowed to send new data. This occurs when the receive window advertised by the receiver is equal to 0. In this case, the receiving application has apparently not yet emptied the receive buffer, so even though TCP has acknowledged all the data the size of the receive window is still equal to zero.

In such a scenario, the receive window and not the application layer is the limiting factor. Therefore, it is necessary to consider only idle periods in which the sender is allowed to send data. To also correctly process flows that are sending using the integer MSS window utilization and the block-based window utilization, not only a receive window equal to zero but also a receive window smaller than the MSS (in case the previous packet length was equal to the MSS) and a receive window smaller than the block size (in case the flow is block based) should result in not considering the corresponding idle period in the calculation.

The result of fake gaps

As explained in Section 3.4.2, fake gaps occur in the repository. Figure 7.2 illustrates a flow with a fake gap (not recorded packets have again been indicated by a grey color). For this example, a large fake idle period would result. However, this problem will only occur when the gap starts exactly at the time the number of outstanding bytes is zero. Moreover, the total duration of the gaps of a flow will mostly be small in comparison to the total duration of the flow, so even one or two fake idle periods will probably not cause a lot of problems.

However, to be as precise as possible, an idle period will only be counted when the first data packet sent after it has the same sequence number as the acknowledgment number of the last received acknowledgment before it. The scenario of Figure 7.2 will therefore not be considered an idle period (after all, the first packet observed after the idle period has sequence number 1201 while the last acknowledgment packet seen before it has acknowledgment number 601). Only when packets get lost

during the fake gap and a retransmission of in this example the packet with sequence number 601 occurs after the fake gap, incorrect results will be the consequence.

Measuring at the sending side
Section 3.5 already mentioned that we are only considering flows for which the measurement point is located at the side of the sender. Also for the detection of idle periods this is very fortunate. At the receiving side it is not easy determine whether or not (and if so, for how long) the sender has periods with no outstanding data. After all, the length of an idle period is defined from the point of view of the sender and at the receiving side we do not know at what time an acknowledgment arrives at the sender or at what time a data packet has been sent.

7.3 Application layer acknowledgments or requests

Besides not providing data fast enough, the application layer can also limit a TCP flow by intentionally placing an upper bound on the number of outstanding bytes. Some form of application layer acknowledgment or request mechanism is assumed to be causing this. We call this the acks with data scenario.

Figure 7.3 illustrates a flow in which this scenario occurs. As we can see, the sender transmits 150 bytes of data and then starts waiting for acknowledgments. When such flows were observed in the repository, our first guess was a receive window or send buffer limitation. However, for some flows the receive window was a lot larger then the number of outstanding bytes and also the send buffer did not seem to be the limiting factor. Furthermore, no loss occurred and almost no idle periods (see Section 7.2) were seen, so also neither the network or a lack of data could cause the speed limitation.

As it turned out, these kind of flows appeared to have acknowledgments that contain a small amount of data (as indicated in the figure). The only logical explanation that remains is that the application layer is limiting the speed of the flow, because it wants to receive some kind of application layer acknowledgment before it provides the TCP entity with more data.

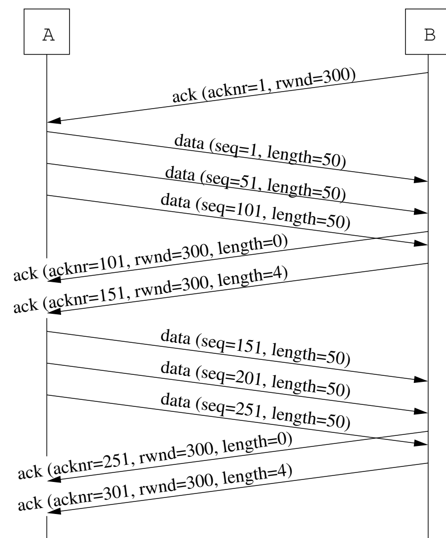


Figure 7.3: *Acks with data scenario*

Detecting the acks with data scenario
After we have seen a lot of TCP flow speed limitations that were difficult to recognize, the acks with data scenario is the first one that does not ask for a lot of work. Our detection is easy: just count the number of acknowledgments that contain data and express this value as a percentage of the total number of acknowledgments.

Because we only consider half-duplex flows (see Section 3.3), a TCP flow would normally have no data in the acknowledgments. Therefore, even a few bytes are still a good indication of the acks with data scenario.

Chapter 8

Conclusions: Interpreting the Results

8.1 Introduction

In the four preceding chapters criteria for detecting the receive window limitation, the send buffer limitation, the network limitation and the application limitation have been developed. The application limitation was divided in two different scenarios. All detection mechanisms can be used to express the limitation as a percentage. Five percentages are used, which are defined as follows:

Receive window limitation percentage
In Chapter 4 we saw how the receive window limitation detection counts the number of peaks that appear to be receive window limited. We express this number as a percentage of the total number of peaks: the *receive window limitation percentage.*

Send buffer limitation percentage
In Chapter 5 a detection mechanism for the send buffer limitation was developed. The number of peaks that appear to be send buffer limited was counted. We express this number as a percentage of the total number of peaks: the *send buffer limitation percentage.*

Bandwidth percentage
In Chapter 6 the network limitation was treated. We presented the actual bandwidth of a flow as a percentage of the maximum achievable bandwidth: the *bandwidth percentage.*

Lack of data percentage
The first application limitated scenario of Chapter 7 is the lack of data scenario. We have developed criteria for presenting the scenario as the percentage of time the number of outstanding bytes is zero: the *lack of data percentage.*

Acks with data percentage
The second application limitated scenario of Chapter 7 is the acks with data scenario. We have developed criteria for presenting the scenario as the percentage of acknowledgments that contained data: the *acks with data percentage.*

So, for all five limitation detections we have come up with a way to express the limitation as a percentage. A single flow can have non-zero percentages for multiple limitations, for example when it is receive window limited (so a non-zero receive window limitation percentage) but also some loss occurs (so a non-zero bandwidth percentage). We cannot just establish the limitation with the highest percentage, since for example a percentage of 10 percent for the lack of data scenario implies this is the limiting factor, while a percentage of 25 percent for the receive window limitation does not. Therefore, thresholds for the percentages are necessary. Furthermore, mostly only one phenomenon is limiting a flow. Hence, we will establish priority rules.

This chapter will first discuss some assumptions made, in Section 8.2, and will then cover all limitations in Section 8.3. The interpretation of the percentages presented in this section will be

covered. Section 8.4 will discuss the priorities of the different limitations, which prescribe which phenomenon limits a flow when multiple limitation percentages are above their individual threshold.

Since the goal of our research was to develop detection criteria for different speed limitations of a TCP flow, the criteria developed in the previous chapters combined with the interpretations presented in this chapter can be considered the conclusions of our work. However, to also prove the usefulness of the detection methods, an implementation has been made. Section 8.5 will cover the initial findings when some files of the repository were processed.

8.2 Assumptions

Our main assumption in this chapter will be that a flow is limited by the same phenomenon the entire time. After all, we have also only looked at detecting limitations over the entire duration of the flow (e.g., both a flow that is completely limited by the network for the first 25 percent of the time and a flow that is never limited by the network but all the time uses 25 percent of the available bandwidth will just result in a bandwidth percentage of 25 percent). Further research could provide a method for dynamically divisioning flows in time periods where a certain phenomenon is the limiting factor.

Furthermore, we assume all the flows to be of reasonable length (our threshold is 100,000 bytes in the data direction), such that start-up phenomena like the slow-start phase of TCP congestion control are negligible.

8.3 Interpretation of the limitation percentages

8.3.1 The receive window limitation

In Chapter 4 detection criteria for the receive window limitation have been developed. Firstly, we created a framework that takes care of performing receive window checks at peaks of the number of outstanding bytes. Secondly, three kinds of receive window utilizations were found and detection criteria for them have been developed.

When the detection criteria for the three utilizations are combined, the number of times the receive window was (almost) full at the time of a peak results. Of course for a flow of 10,000 peaks a resulting value of 100 should not be interpreted the same as for a flow of 100 peaks. Therefore, the result is presented as a percentage of the total number of peaks: the receive window limitation percentage.

Because of a variety of circumstances, the receive window limitation percentage will (almost) never become one hundred percent. For example, during the first phase of the TCP congestion control mechanism the congestion window will for almost all flows be the limiting factor. A flow that at first has to 'get started' and since, for example, the twentieth packet is receive window limited for the rest of the time, has to be considered receive window limited in general. Furthermore, a flow that is *not* limited by the receive window will generally not have a large receive window limitation percentage. Therefore, our threshold for establishing the receive window as the limiting factor of a flow is set to 50 percent.

Identifying the cause of a receive window limitation

Besides detecting the receive window limitation itself, we also provided a way to identify the cause of a receive window limitation; this will either be the application that cannot handle the data any faster or a sub-optimal configuration of the operating system.

The percentage of the acknowledgments that contained a decreased receive window — which we will call the decreased receive window percentage — was calculated. Since a receiving application that has *no* problems with reading the data in time will almost never advertise a decreased receive window, even a flow with a small decreased receive window percentage will still indicate that the speed of reading the data (and not the configuration of the operating system) is the cause of the receive window limitation. Therefore, we use a threshold of just 25 percent for establishing the correct operation of the receive window. The cause of the receive window limitation is assumed to be a sub-optimal configuration in the TCP stack when the decreased receive window percentage is below 25 percent, while the speed of data processing of the application is assumed to be the cause in case the percentage is 25 percent or higher.

8.3.2 The send buffer limitation

In Chapter 5 detection criteria for the send buffer limitation have been developed. The framework and utilizations detection mechanisms of Chapter 4 have been reused, but criteria for estimating the send buffer size had to be developed. Furthermore, we had to cope with updates of the send buffer size estimation.

Just like we did for the receive window limitation, also for the send buffer limitation the different utilizations (defined in Chapter 4) can be combined. We define the send buffer limitation percentage as the number of send buffer limited peaks expressed as a percentage of the total number of peaks.

The send buffer limitation percentage can be interpreted in the same way as the receive window limitation percentage. Also for this quantity we therefore propose a threshold of 50 percent before the send buffer limitation will be established.

8.3.3 The network limitation

In Chapter 6 detection criteria for the network limitation have been developed. We have seen how to estimate the loss rate, estimate the RTT and use this information to calculate the bandwidth percentage. This percentage indicates the used percentage of the maximum possible bandwidth by TCP according to the TCP Friendly formula.

Unfortunately, as mentioned before, the bandwidth percentage is not very accurate (for example because of different TCP implementations, lost retransmissions and the slow-start phase). Therefore, even a bandwidth percentage of 50 percent is considered to be proving the network limitation.

8.3.4 The lack of data limitation

In Chapter 7 detection criteria for the lack of data scenario (part of the application limitation) have been developed. We have seen that this limitation occurs when the application does not provide enough data. The percentage of the time that there is no outstanding data, is called the lack of data percentage.

The lack of data percentage does not have to be large to indicate a lack of data limitation. When there is always data available, there will never be no outstanding bytes (unless of course the receive window is very small, but these idle periods have not been taken into account for the lack of data limitation, as indicated in Section 7.2). There is one exception to this rule, namely that it is possible that the receive window is small and all outstanding packets are acknowledged in a short period, such that there is a little time between receiving the final acknowledgment and sending the next data byte. Therefore, there should be a threshold for the lack of data limitation, even though it need not be high. We have chosen a value of 2 percent.

8.3.5 The acks with data limitation

In Chapter 7 detection criteria for the acks with data scenario (part of the application limitation) have been developed. This resulted in the acks with data percentage: the percentage of the acknowledgment packets that contains data.

If even a relatively low percentage of the acknowledgments contains data, this indicates that an application protocol with acknowledgments or requests is being used. After all, normally the half-duplex flows examined would be expected to contain no data at all in the acknowledgments. A threshold of 10 percent for the acks with data percentage is used to identify the acks with data limitation.

8.4 Prioritizing the limitations

So far all five limitations were covered seperately, resulting in five limitation establishment guidelines. However, when for example both the lack of data scenario and the receive window limitation apply, only the lack of data is limiting the flow. As it turns out we can prioritize the limitations such that of all established limitations the one with the highest priority applies.

1. The lack of data scenario

   This limitation is dominant over all other limitations. When there is no more data available to send, improving other factors will not improve the overall speed of the flow but will only increase the length of the idle periods. Therefore, when the lack of data limitation is present, all other limitations do not apply anymore.

2. The receive window limitation

   The receive window limitation percentage gives quite a lot of certainty about the limiting factor of a flow. When the lack of data limitation does not apply and the receive window is (almost) completely filled for most of the time, this assures it is limiting the flow, because the size of the receive window is known exactly and the limitation is therefore certain.

3. The send buffer limitation

   The send buffer limitation percentage gives less certainty than the receive window percentage, since — as mentioned in Section 5.4.1 — the real size of the send buffer is not known. It is estimated by looking at the largest number of outstanding bytes, but this value is also limited by the receive window. So, when a flow is receive window limited, it is likely that the send buffer is estimated to be equal to the receive window, resulting in an incorrect estimation and thus and incorrectly high send buffer limitation percentage. Therefore, the send buffer should only be established as the limiting factor of a flow when the receive window (and of course the lack of data limitation) is *not*.

4. The network limitation

   The bandwidth percentage itself is not related to other limitations, but still it is possible that another factor is limiting a flow, even though for example 60 percent of the available bandwidth is used. Since the other limitation detections are more reliable, we therefore only consider a flow network limited if it is not limited by a lack of data, the receive window or the send buffer.

5. The acks with data scenario

   The occurrence of the acks with data scenario by itself does not necessarily mean the flow is also *limited* by this phenomenon. The occurrence of data in the acknowledgments does not necessarily mean the flow is actually speed limited by this. Therefore, the limitation is only a last resort if all other limitations are not present.

8.5 Processing the repository

Since the main goal of our research was to develop detection criteria, extended statistical analysis of the complete repository based on the detection criteria will be left for further research. However, we will give a first impression of the results that can be obtained using the limitation detection criteria developed in this report.

First, we have used our implementation of the limitation detection criteria to analyse the three repository files mentioned earlier in Section 3.4.2. Table 8.1, Table 8.2 and Table 8.3 show the information presented by the implementation.

Furthermore, Figure 8.1 shows three scatter graphs for these three repository files. On the horizontal axis we find the number of transmitted bytes of a flow (excluding retransmissions). For each of the analysed flows, the detected limitation is indicated by the location of the dot above the position

of the x-axis that represents its length. The vertical position of dots inside the range of a specific limitation is purely random.

Finally, we combined the results of five repository files (loc1-20020524-1115, loc1-20020527-1115, loc1-20020528-1115, loc1-20020526-1115 and loc1-20020625-0415) to get an overview of the occurrence of the limitations. Table 8.4 shows the results obtained. Not all files of location 1 were used, because it appeared our implementation was unable to read files larger than 2 GB.

Some preliminary conclusions can be drawn from Figure 8.1 and Table 8.4. However, these conclusions should only be used as a motivation for further research; a thorough statistical analysis is necessary to acquire scientific justifiable results.

Preliminary conclusions

- The network only limits 12.3 percent of the flows (and 13.0 percent of the bytes)
- Almost all receive window limitations are caused by a sub-optimal configuration of the TCP stack. A small change in the operating system should improve the speed of these connections.
- The acknowledgments with data scenario does not occur often.
- The lack of data scenario occurs mostly for flows with a small amount of data bytes, as can be seen from Figure 8.1. This is logical, since a flow that does not have a lot of data to send will not send a lot of data. However, Table 8.4 also shows that this is the most frequent occurring limitation.
- The undetermined flows are mostly flows with a small amount of data bytes. This was expected, since start-up phenomena can interfere with the detection mechanism.

8.6 Further work

Although this report addresses a lot of issues concerning the identification of the speed limiting factor of a TCP flow, further research should be performed to solve the problems we have identified and not solved ourselves. The following topics can be covered in future work:

- At this point, we have mostly looked at ways to recognize certain speed limiting factors. Now we know how to look at *what* happens, it is also very interesting to look at *why* certain phenomena occur.
- It is very disappointing that only flows with the measurement point close to the sender can be considered. At this point it seems very hard or maybe even impossible to also consider the other half of the flows, but future research might prove otherwise.
- Analysis of full-duplex flows is not addressed thus far. Incorporating these flows in the detection criteria will probably not be easy, but improves the completeness of our methodology.
- The network detection assumes no network reordering exists. Even though measuring at the sending side reduces the consequences of this simplification, further research could be performed to come up with a more accurate solution.
- The reason for the occurence of the acks with data scenario is assumed to be some kind of application layer acknowledgments or requests mechanism. It would be interesting to look at flows that exhibit this behaviour to determine whether or not this assumption is correct.
- Creating a repository without fake gaps would simplify a lot of the detection mechanisms and would improve their accuracy. Our detection methods could be splitted into two versions: one for faulty and one for correct repositories. Our gap detection could be used to determine which version should be used.
- The implementation of our algorithm can at the moment only handle off-line repository files, smaller than 2 GB. Extending the implementation to also support on-line analysis and files larger than 2 GB would improve its usability.

Limitation	#flows	flows%	#bytes	bytes%
Application Layer Limitation (lack of data scenario)	310	25.556%	808,107,134	20.409%
Application Layer Limitation (acks with data scenario)	7	0.577%	42,897,578	1.083%
Receive Window (slow data reading)	61	5.029%	62,172,608	1.570%
Receive Window (sub-optimal OS settings)	226	18.631%	1,153,909,106	29.142%
Send Buffer	110	9.068%	606,652,542	15.321%
Network	147	12.119%	574,497,691	14.509%
Undetermined	352	29.019%	711,400,054	17.966%

Table 8.1: *Limitation information about the file loc1-20020625-0415*

Limitation	#flows	flows%	#bytes	bytes%
Application Layer Limitation (lack of data scenario)	364	35.340%	664,930,124	12.752%
Application Layer Limitation (acks with data scenario)	44	4.272%	302,938,264	5.810%
Receive Window (slow data reading)	36	3.495%	81,655,177	1.566%
Receive Window (sub-optimal OS settings)	184	17.864%	1,277,236,867	24.494%
Send Buffer	102	9.903%	1,319,554,867	25.306%
Network	111	10.777%	1,025,274,368	19.662%
Undetermined	189	18.350%	542,866,526	10.411%

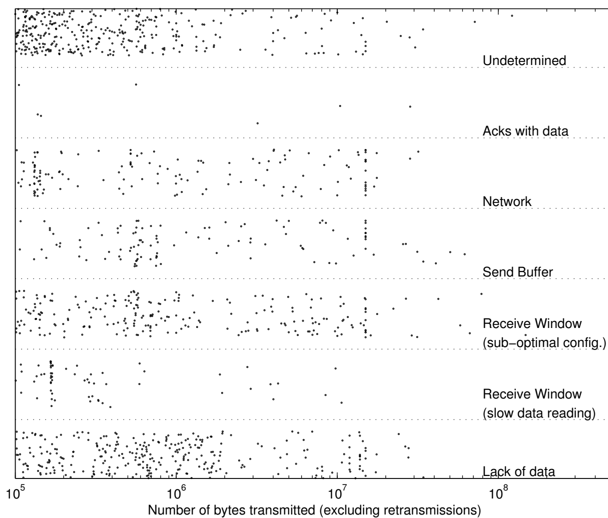
Table 8.2: *Limitation information about the file loc1-20020526-1115*

Limitation	#flows	flows%	#bytes	bytes%
Application Layer Limitation (lack of data scenario)	461	33.023%	2,133,952,588	25.015%
Application Layer Limitation (acks with data scenario)	23	1.648%	60,160,707	0.705%
Receive Window (slow data reading)	48	3.438%	128,948,744	1.512%
Receive Window (sub-optimal OS settings)	236	16.905%	2,681,577,903	31.435%
Send Buffer	148	10.602%	1,454,861,234	17.055%
Network	140	10.029%	1,186,796,163	13.912%
Undetermined	340	24.355%	884,330,113	10.367%

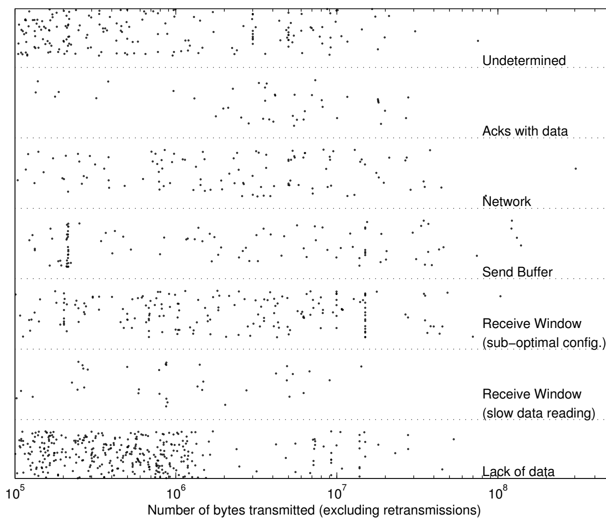
Table 8.3: *Limitation information about the file loc1-20020528-1115*

Limitation	#flows	flows%	#bytes	bytes%
Application Layer Limitation (lack of data scenario)	2055	32.256%	16,082,988,050	37.166 %
Application Layer Limitation (acks with data scenario)	110	1.727%	559,831,179	1.294 %
Receive Window (slow data reading)	249	3.908%	564,796,101	1.305 %
Receive Window (sub-optimal OS settings)	1170	18.364%	9,690,333,749	22.393 %
Send Buffer	596	9.354%	6,744,361,258	15.585 %
Network	784	12.306%	5,642,629,834	13.039 %
Undetermined	1407	22.084%	3,988,845,105	9.218%

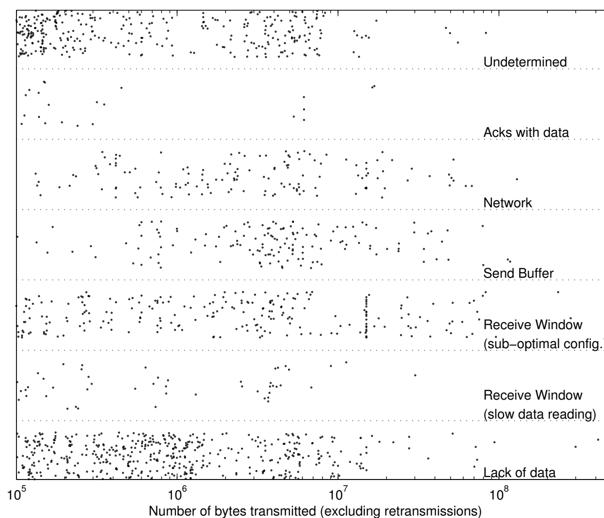
Table 8.4: *Limitation information about several files combined*



(a) loc1-20020625-0415



(b) loc1-20020526-1115



(c) loc1-20020528-1115

Figure 8.1: *Speed limitations for a few repository files*

Appendix A

Proof for Section 3.4.2

We will prove that a file that contains 12,553,221 packets in 900,000 milliseconds has approximately a chance of $6 \cdot 10^{-12}$ of experiencing at least one empty 5-milliseconds interval.

For the file described, the average number of packets in a 5-milliseconds interval is 69. By a very rough approximation the number of packets that arrive such a 5-milliseconds interval can assumed to be Poisson distributed with $\mu = 69$, which in turn can be approximated by a normal distribution with $\mu = 69$ and $\sigma^2 = 69$. Under these assumptions, we can calculate the chance that no packet arrives in a 5-milliseconds interval (Z being standard normal distributed):

$$P(N(\mu = 69, \sigma^2 = 69) \leq 0) = P(Z \leq \frac{(0-69)}{\sqrt{69}}) \approx P(Z \leq -8.3) \approx 3.4 \cdot 10^{-17} \tag{A.1}$$

Still assuming no dependency between packet arrivals, the chance of observing at least one empty 5-milliseconds interval is:

$$1 - (1 - 3.4 \cdot 10^{-17})^{900{,}000/5} \approx 6 \cdot 10^{-12} \tag{A.2}$$

Although the approximation of packet arrival by a Poisson distribution is very rough, even with a mistake of a factor 1,000 the chance of an empty 5-milliseconds interval would still be negligible.

Bibliography

[Bra89] R. Braden, *RFC 1122; requirements for internet hosts - communication layers*, 1989, Internet Engineering Task Force.

[Cer91] Vinton G. Cerf, *Rfc 1262; guidelines for internet measurement activities*, 1991, Internet Activities Board.

[Eth05] Ethereal, *The world's most popular network protocol analyzer*, `http://www.ethereal.com/`, 2005.

[Jac88] V. Jacobson, *Congestion avoidance and control*, SIGCOMM '88: Symposium proceedings on Communications architectures and protocols, ACM Press, 1988, pp. 314–329.

[Kla05] Ruud Klaver, *Experimental validation of the tcp-friendly formula*, `http://dacs.cs.utwente.nl/assignments/completed/Klaver_B-assignment.pdf`.

[KP95] Phil Karn and Craig Partridge, *Improving round-trip time estimates in reliable transport protocols*, SIGCOMM Comput. Commun. Rev. **25** (1995), no. 1, 66–74.

[MLAW99] Jeonghoon Mo, Richard J. La, Venkat Anantharam, and Jean C. Walrand, *Analysis and comparison of TCP reno and vegas*, INFOCOM (3), 1999, pp. 1556–1563.

[Mog92] Jeffrey C. Mogul, *Observing TCP dynamics in real networks*, SIGCOMM '92: Conference proceedings on Communications architectures & protocols, ACM Press, 1992, pp. 305–317.

[MSM97] Matthew Mathis, Jeffrey Semke, and Jamshid Mahdavi, *The macroscopic behavior of the tcp congestion avoidance algorithm*, SIGCOMM Comput. Commun. Rev. **27** (1997), no. 3, 67–82.

[PFTK98] J. Padhye, V. Firoiu, D. Towsley, and J. Krusoe, *Modeling TCP throughput: A simple model and its empirical validation*, Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication (1998), 303–314.

[Pos81] Jon Postel, *RFC 793; transmission control protocol*, 1981, Information Sciences Institute.

[Res03] Lawrence Berkeley National Laboratory Network Research, `http://www.tcpdump.org/`, 2003.

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

[SUR05] SURFnet, *Surfnet-netwerk*, `http://www.surfnet.nl/info/netwerk/nationaal/home.jsp`, 2005.

[vdM03] Remco van de Meent, *M2c measurement data repository*, `http://arch.cs.utwente.nl/projects/m2c/m2c-D15.pdf`, 2003.

[VLL05] Bryan Veal, Kang Li, and David K. Lowenthal, *New methods for passive estimation of tcp round-trip times.*, PAM, 2005, pp. 121–134.

Acknowledgments

After working for more than six months this bachelor's thesis is ready. I would like to thank everyone who has made this possible.

First of all, I would like to thank my mentors Aiko Pras and Pieter-Tjerk de Boer of the chair Design and Analysis of Communication Systems. When we discussed the assignment somewhere around December 2004, I mentioned to them that I wanted a difficult assignment and furthermore that I liked to get a mentor who was not afraid of giving a lot of critical comments, in order to create a bachelor thesis I could be proud of. Now we have reached the end, I can look back on in my opinion a good cooperation. Pieter-Tjerk has has given a lot a comments and indeed did not settle for less than perfection, just like I wanted. Although perfection is of course never achievable, I am very satisfied with the end result.

Besides our useful cooperation, I will also miss my talks with Pieter-Tjerk about everything from five-dimensional drawings, teaching, Linux and the Dutch and English language, to telecommunication, holidays and the (lack of) difficulty of the study of Telematics.

Next, I would like to thank my friends, especially André and Hester, who took care of assuring that I did not only work but also had a lot of fun.

Futhermore, I would like to thank Dineke van Aalst, who performed a final check on the spelling and grammar of this report.

Last, but definitely not least, I would like to thank my family, who has always been there for me.

List of Figures

2.1 *The internet protocol stack* . . . 3
2.2 *A packet transmitted over TCP* . . . 4
2.3 *Example TCP flow* . . . 5

3.1 *Repository measurement setting* . . . 7
3.2 *Headers of IP and TCP packets* . . . 8
3.3 *Acknowledgment of "lost" packet* . . . 9
3.4 *Captured bytes* . . . 9
3.5 *Filling of gaps* . . . 10
3.6 *Fake gaps in the repository* . . . 11
3.7 *Location of the measurement point* . . . 12

4.1 *Time-sequence diagrams of a TCP flow that is restricted by the receive window* . . . 16
4.2 *TCP sliding window* . . . 16
4.3 *Ambiguities at the receiving side* . . . 18
4.4 *Receive window limited flow* . . . 21
4.5 *Receive window limited flow* . . . 22
4.6 *Receive window checking* . . . 23
4.7 *Framework for detecting the receive window limitations* . . . 24
4.8 *Complete window utilization* . . . 25
4.9 *Algorithm for detecting complete window utilizations* . . . 25
4.10 *Integer MSS window utilization* . . . 26
4.11 *Algorithm for detecting integer MSS window utilization* . . . 26
4.12 *Block-based window filling* . . . 27
4.13 *Algorithm for detecting block-based sending behaviour* . . . 28
4.14 *Algorithm for detecting block-based window utilization* . . . 28

5.1 *Send buffer limitation* . . . 32
5.2 *Diagrams of send buffer and receive window limited TCP flows* . . . 33
5.3 *Algorithm for estimating the send buffer size* . . . 34
5.4 *Send buffer limitation* . . . 34
5.5 *Send buffer limitation* . . . 35
5.6 *Different window utilizations* . . . 36
5.7 *Send buffer estimation update scenarios* . . . 37

6.1 *Loss events* . . . 40
6.2 *Counting retransmissions* . . . 41
6.3 *Flaws with the loss event detection* . . . 42
6.4 *Estimating the RTT* . . . 43
6.5 *RTT vs. DupACKs* . . . 43
6.6 *RTT estimation with loss* . . . 44

7.1 *Lack of data* . . . 47
7.2 *Flow with fake gap* . . . 48
7.3 *Acks with data scenario* . . . 49

8.1 *Speed limitations for a few repository files* . . . 57

List of Abbreviations

ack	Acknowledgment
acknr	Acknowledgment Number
BW	Maximum Achievable Bandwidth
cwnd	Congestion Window
FTP	File Transfer Protocol
IP	Internet Protocol
IRC	Internet Relay Chat
MSS	Maximum Segment Size
nextseq	Next Sequence Number
OS	Operating System
RFC	Request For Comments
RTT	Round-Trip Time
rwnd	Receive Window
seq	Sequence Number
TCP	Transmission Control Protocol

Index

acknowledgment, 3
  acknr, 5
  delayed acknowledgment, 39, 43
  duplicate acknowledgment, 3, 8, 23, 40, 42
application, 1, 5
application layer, 4, 47–49
application layer acknowledgment, 49

bandwidth percentage, 39, 45
block, 27, 31, 38
block-based buffer utilization, 32–35, 38
block-based sending behaviour, 27
block-based window utilization, 27, 48

complete buffer utilization, 32, 38
complete window utilization, 25
congestion, 4
  congestion control, 4, 39, 40
  congestion window, 39, 40, 42
  cwnd, 4
connection, 8

data burst, 22, 23

ending packet, 27
Ethernet, 9

fake gap, 9, 12, 23, 27, 33, 47, 48
FIN, 9, 10
flow, 8
flow control, 4
full-duplex, 8

half-duplex, 5, 8, 49
header, 8, 23, 33
  Ethernet header, 9
  header options, 9
  IP header, 9
  TCP header, 9

idle period, 48, 49
  fake idle period, 48
integer MSS buffer utilization, 32–34, 36, 38
integer MSS window utilization, 26, 48
IP, 3
  IP address, 13
  IP address tuple, 8
  IP prefix, 13

Karn's algorithm, 44
keep-alive message, 21

lack of data scenario, 47
loss event, 40–42
loss rate, 39, 40, 44
lowest accepted send buffer limited peak, 35, 36
  lowest accepted block-based send buffer limited peak, 38
  lowest accepted integer MSS send buffer limited peak, 38

maximum achievable bandwidth, 39, 45
measurement location, 6, 12, 13, 17, 19, 22, 40, 42, 43, 49
MSS, 4, 26, 27, 32, 36, 48

network, 1, 5, 39, 47
network layer, 3

outstanding bytes, 4, 5, 15–23, 31–33, 40, 47, 49

packet, 3
packet loss, 4, 6, 39, 42, 44
passive monitoring, 8
pcap, 8
peak, 22, 31, 33–36, 38, 39
  complete buffer, 38
  max block-based, 38
  max category, 35, 36
  max integer MSS, 38
  non-max block-based, 38
  non-max category, 35, 36
  non-max integer MSS, 38
privacy, 7

receive window, 1, 4, 5, 15–23, 25–28, 31, 32, 39, 47–49
  rwnd, 4
reliability, 3
reordering, 18, 40, 42
repository, 1, 7, 12
retransmission, 3, 5, 23, 40–43
  retransmission burst, 41, 42
RTT, 4, 39, 42–44

scrambling, 8, 13
send buffer, 1, 5, 23, 31–33, 35–39, 49

sending pattern, 34
sequence number, 3, 5
    highest expected next sequence number, 10, 42
    next sequence number, 10, 20, 41
    nextseq, 5, 15
    not-expected sequence number, 40
    seq, 5
streaming audio, 31, 47
SYN, 9

TCP, 1, 3
    TCP header, 4
TCP Friendly formula, 39, 40, 44
TCP port, 8
tcpdump, 8
timestamp, 9

UDP, 39
update threshold, 38

window scaling, 9, 24