Technische Universität München
Fakultät für Informatik

Institut Eurécom
Sophia-Antipolis

Master's Thesis

Network Tomography Tools
Werkzeuge zur Netzwerktomographie

Michael Dyrna

Aufgabensteller:	Prof. Dr. J. Schlichter
Betreuer:	Prof. Dr. E. W. Biersack
	Prof. Dr. J. Schlichter
Abgabedatum:	15. September 2005

Abstract

Network Tomography is *the collection and analysis of data about network functions on all network layers*. It is essential in order to learn about computer networks and to improve them.

The objective of this thesis is to infer key variables from TCP connections such as the *congestion window* and the *round trip time*. We have examined approaches to infer these variables from TCP traces as well as an approach that reads them from the operating system.

The tangible result of this thesis is a set of software tools that infer the round trip time from TCP traces as well as a web server that makes several key variables of a TCP transmission visible to the user.

Zusammenfassung

Unter Netzwerktomographie verstehen wir *das Sammeln und Analysieren von Daten über Netzwerkfunktionen auf allen Netzwerkschichten.* Sie ist von grundlegender Bedeutung, um etwas über Rechnernetze zu erfahren und um sie zu verbessern.

Das Ziel dieser Arbeit ist, wichtige Variablen, wie zum Beispiel das *Congestion Window* und die *Round Trip Time*, aus TCP-Verbindungen abzuleiten. Wir haben Ansätze untersucht, diese Variablen aus TCP Traces abzuleiten, sowie einen Ansatz, der sie vom Betriebssystem ausliest.

Konkretes Ergebnis dieser Arbeit ist eine Reihe von Werkzeugen, die die Round Trip Time aus TCP Traces ableiten, sowie ein Web Server, der dem Benutzer wichtige Variablen einer TCP-Übertragung anzeigt.

Contents

List of Figures

Chapter 1

Motivation and Fundamentals

1.1 Motivation

Computer networks, once only available to universities, governments and big enterprises, have reached numerous end users in many countries today in form of the Internet, the extensive, worldwide computer network available to the public.

The technology of the Internet goes back to the 1970s. The *TCP/IP protocol suite*, which replaced the simpler *Network Control Program* (NCP) in 1983 and which is still in use today, is a set of communication protocols that follow a strict partition into interchangeable *network layers*, each one solving a specific set of problems, reaching from the physical encoding of bits and bytes (*physical layer*) over transporting data between two directly connected hosts (*data link layer*), the switching of packets along an adequate path of hosts (*network layer*), providing reliability between two end hosts (*transport layer*) to application specific issues (*application layer*).

Internet protocols are a continuous subject of research and further development. Although the Internet layers were designed to be independent of each other, new requirements and achieved improvements on one layer can push another layer to its limits. For example the growing available bandwidth over long distance achieved by fiber optics and satellite links has necessitated to extend the *Transmission Control Protocol (TCP)*, that is the dominant reliable end-to-end transport protocol used in the Internet.

One important goal that has not yet been completely achieved in spite of all past research is to understand TCP's performance and the factors that can limit its behavior in practice.

So the objective of this thesis is to develop *Network Tomography* tools for analyzing TCP transmissions. Therefore, two approaches will be examined and implemented in software tools. The first one will deduce certain parameters from TCP traffic recorded somewhere *between* the two hosts involved in the transmission, whereas the second one will query the operating system of one of the two hosts for these parameters.

1.2 Network Tomography of TCP

We define network tomography in general as *the collection and analysis of data about network functions on all network layers, from physical data transmission to the behavior and performance of Internet systems and applications.*

When we conduct network tomography we can profit in many ways. First, we can deduce knowledge about how to design computer networks (queue management, buffer and bandwidth provisioning, fair utilization of resources, etc.) and how to optimize networking applications. Furthermore we can detect wrong behavior of network components. Paxson et al. [23] for example found a number of implementation errors in TCP implementations by conducting network tomography. Last but not least we might be able to contribute to improvements of existing network protocols, like TCP has experienced numerous improvements over the last more than twenty years, or design better future protocols.

In this thesis we will concentrate on network tomography on the transport layer of the Internet, which is the Transport Control Protocol (TCP).

The two important characteristics of network transport performance are *delay* and *throughput*. The former denotes the time a data packet needs to travel along the network path from the source host to its destination. The latter denotes the amount of data that can be transmitted per time. Network, queueing and processing delay, packet loss and other pathologies manifest themselves in the effective throughput perceived by the application that uses TCP.

Network tomography on the transport layer allows us to examine numerous questions. What limits throughput? Are end hosts the bottleneck because they cannot generate (sender) respectively process (receiver) data faster? Is the physical network the bottleneck? I.e. is congestion on intermediate switching points limiting data transfer?

Has the transport protocol undesirable side effects under certain unforeseen circumstances? Does certain behavior of other network hosts influence performance? What effects does the interaction of the network layers have?

To answer these questions we need tools to “look into” what happens during data transmission with TCP. This thesis will provide a few of them.

1.3 Reliable Stream Transport

1.3.1 Motivation

Computer networks at the lowest level generally provide unreliable packet transport mechanisms [6]. That means that they can route packets from one host to another, while it is neither guaranteed that the packets are delivered in the right order, nor that they are delivered at all. Packets might also be duplicated by the network.

In contrast, an indispensable requirement in most networking software is to send or receive a stream of data, not packets. Implementing reliable stream transport based on a packet oriented routing service in every application is anything but trivial and would require to solve the same problems such as error detection and retransmission over and over by every network application developer.

It is therefore only logical that one has found a general purpose solution integrated into the operating system's network protocol stack providing applications with a simple programming interface similar to file access while hiding networking details.

1.3.2 Properties

The functionality of reliable stream transport is as follows: A user process on the sending host passes a stream of data bytewise to the transport service which splits it into packets and transmits them to the receiving host using the underlying protocol layer [6]. On the receiving side the packets are reassembled and delivered bytewise to the receiving application.

If a user process wants to use the reliable stream transport service it has to make a *virtual circuit connection* first, i.e. establish a connection analogous to opening a file. The remote host and application must accept the connection. Both actions are achieved by exchanging messages over the network. If a host detects a communication failure (e.g. because hardware along the communication path has failed) or if the remote side closes the connection, it reports the event to the application. The circuit is considered *virtual* because the reliability is only an illusion provided by the stream transport service.

The packet size is determined by the service. Buffering allows the user process to transmit chunks of data smaller or larger than that fixed packet size. The transport service collects data and only sends them when the full packet size is reached.

A *push* mechanism is provided so that the user process can induce the immediate transmission of the current buffer even if it is not full yet. That is needed for interactive protocols where the sender of a request expects an answer from the remote side although the request does not fill a full packet.

A connection that allows concurrent data transfer in both directions is called a *full duplex connection*. From the application's point of view, the connection consists of two independent streams in opposite directions. An advantage from the transport service's point of view is that it can transmit control information for the one direction together with the data in the other direction in the same packet and thus reduce network traffic.

1.3.3 Providing Reliability

Three kinds of network failure must be detected and overcome to provide reliability to the application: Packets might get lost along the network path, they can be delivered out of order or duplicated.

In order to detect and remedy loss the stream transport service uses a technique known as *positive acknowledgement with retransmission*: The receiver sends back messages to the sender acknowledging the reception of each packet. The sender keeps the sent packets in a buffer until it receives the acknowledgement from the remote host. Furthermore it starts a timer for every data packet upon sending. If no acknowledgement is received within a certain time the sender retransmits the packet. The procedure is shown in figure 1.1[1].

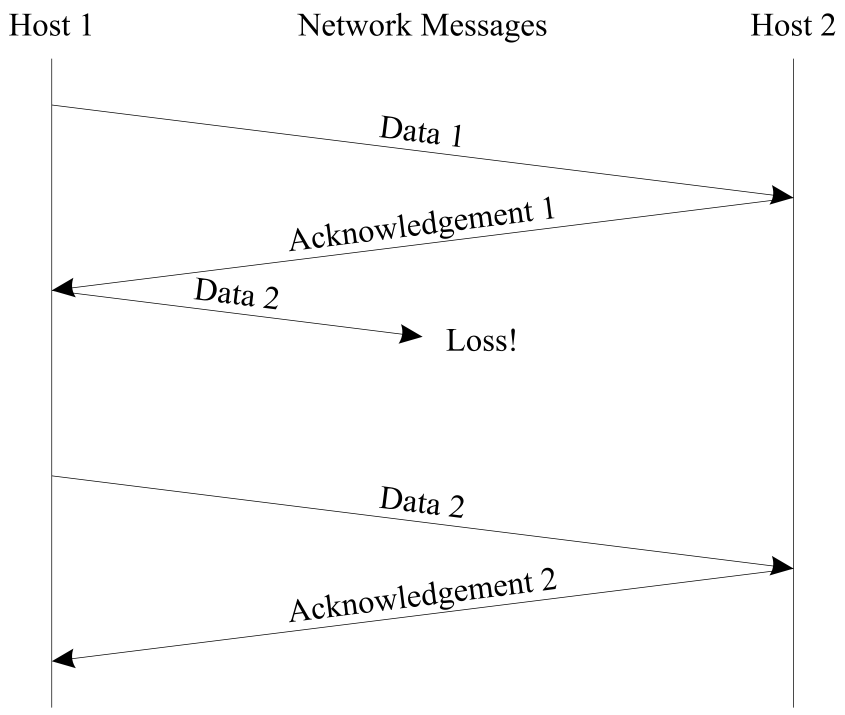


Figure 1.1: Sequence diagram for the positive acknowledgement with retransmission sequence diagram

Ascending sequence numbers are assigned to the packets in each direction. This assures that the receiver is able to reassemble the data stream in the right order. This solves at the same time the problem of duplicated packets: A packet with a packet number that has already been received is just dropped.

The three mentioned problems can occur to acknowledgement messages as well: A lost acknowledgement will eventually cause the sender to needlessly retransmit a packet which will be perceived as a duplicate at the receiver side. Acknowledgements that have been delivered out of order can be assigned to the right data packets by the sequence numbers. A duplicate acknowledgement is perceived by the sender as an acknowledgement for a packet that is no longer in its send buffer and will simply be dropped.

[1]In *sequence diagrams* vertical distance represents increasing time and diagonal lines across the middle represent transmitted network messages.

1.3.4 Sliding Window

The method described above perfectly ensures that the stream written by the sending application arrives without corruption at the receiver. However, it will perform very poorly because it does not use the network to its full capacity at all and thus wastes available bandwidth. Consider for example a round trip time of 50 ms and a packet size of 1000 bytes. Then a sender could only transmit 20 000 bytes per second, although the network might have a much higher capacity.

The *sliding window technique* is an extension of positive acknowledgement with retransmission that allows more than one packet to be sent before an acknowledgement is received. A packet is called *unacknowledged* if it has been sent but no acknowledgement has been received yet. The maximum number of unacknowledged packets is called the *window size*. The actual number of unacknowledged packets is called the *flight size*[2]. The sender still maintains a timer for each packet and retransmits it if no acknowledgement has been received within a certain time.

We imagine the window lying on the stream of data ready to be sent and “sliding” along the stream by one packet to the right as soon as the leftmost packet has been acknowledged and so forth (figure 1.2). This partitions the stream into three sets: The data to the left of the window has been transmitted successfully; the data within the window has already been sent but not yet acknowledged; the data to the right of the window has not yet been sent.

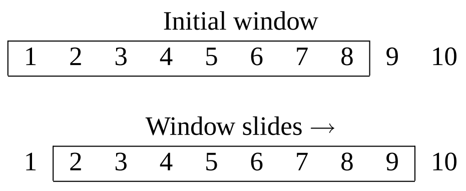


Figure 1.2: A sliding window protocol with eight packets in the window. As soon as an acknowledgement for the first packet is received the ninth packet can be sent.

The performance of a sliding window protocol depends linearly on the window size (and is of course limited by the speed at which the network accepts packets). Consider the example from above where the round trip time was 50 ms and a packet size was 1000 bytes. If you set the window size to 8, the reachable bandwidth will be approximately 160 kilobytes per second instead of only 20. If you set the window size to 1, the protocol would be identical to the positive acknowledgement and retransmission method.

[2]Consequently the flight size is less or equal to the window size.

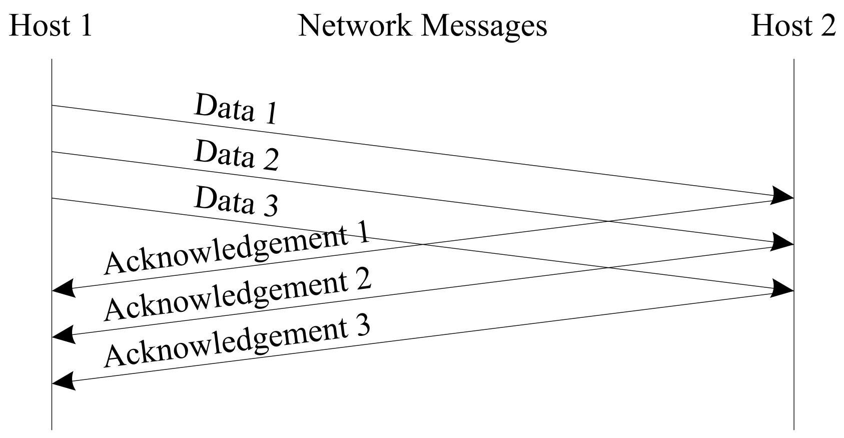


Figure 1.3: The sequence diagram for the sliding window protocol

1.4 The Transmission Control Protocol (TCP)

The *Transmission Control Protocol* (TCP) is the reliable stream transport protocol of the TCP/IP protocol suite. Together with the *Unreliable Datagram Protocol (UDP)* it forms the *Transport Layer* in the Internet.

It specifies the format of data and acknowledgements, the mechanisms for providing reliability, flow control by the receiver, network congestion control, how to distinguish between multiple destinations and connections on a machine, as well as how two applications initiate and terminate a connection.

The interface to the application layer — i.e. how an application opens and closes a connection and how it transmits data — is *not* part of the TCP specification but depends on the implementation.

The original specification of the Transmission Control Protocol from 1981 can be found in [25]. [5] in 1989 updated the standard and clarified several points.

Although within the TCP/IP protocol stack TCP is built on the Internet Protocol (IP) as network layer it is designed to be deployable in any other environment as well because it makes only few assumptions on the underlying communication system.

1.4.1 Addressing

TCP allows multiple processes on a given host to communicate at the same time and it demultiplexes incoming traffic among them. To identify the destination application within a host TCP uses *protocol port numbers*, which are 16 bit integer numbers.

An *endpoint* is a pair of integers: the host address (IP address) and the port number. Several endpoints can be associated to one process. A TCP *connection* is a pair of endpoints. This implies that one endpoint can be part of several connections.

For example one application process can connect to a web server's endpoint 193.55.113.240:80[3] from its endpoint 82.83.197.81:1069, whereas an application process on another host can connect to exactly the same endpoint from its local endpoint 213.23.23.18:1139.

1.4.2 TCP Segment Format

The data unit of the TCP specification is a *segment*. Each segment consists of a variable-length header and the data it transports. Figure 1.4 depicts the structure of the TCP header.

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
```

Source Port			Destination Port	
Sequence Number				
Acknowledgement Number				
Offset	Reserved	Flags	Window	
Checksum			Urgent Pointer	
(Options)				(Padding)

Figure 1.4: TCP header format

The 16 bit *Source Port* and *Destination Port* fields contain the TCP port numbers that identify the two endpoints of the connection[4].

The 32 bit *Sequence Number* identifies the position in the sender's byte stream and is needed by the receiver to re-assemble the segments in the right order. Note that this is not a segment number as in the example in the previous section but the stream position of *the first byte* of the segment. The initial sequence number is a random value determined at the beginning of a connection (see 1.4.3). The *Acknowledgement Number* contains the next sequence number that the sender of the segment expects to receive, which is the sequence number of the last received segment plus its length (see 1.4.4).

The 4 bit *Data Offset* field gives the beginning of the payload within the segment. This is necessary since the *Options* field (see below) has variable length. It is given as a multiple of 4. Consequently the length of the header in bytes must be divisible by 4. This is achieved by inserting *Padding* after its end if necessary.

The *Flags* field contains six control bits. Their meaning is given in figure 1.5.

In the 16 bit *Window* field the source indicates to the destination the available size of its receive buffer.

[3]The colon is the delimiter between host address and port number.

[4]The source and destination host addresses do not appear in the TCP header since they are part of the network protocol, i.e. IP in the Internet.

URG	*Urgent Pointer* field significant
ACK	*Acknowledgment Number* field significant
PSH	Push Function
RST	Reset the connection
SYN	Synchronize sequence numbers
FIN	No more data from sender

Figure 1.5: TCP flags

The 16 bit *Checksum* enables the receiver of a segment to verify that it has not been corrupted by lower network layers during the transmission.

The 16 bit *Urgent Pointer* in not used in present applications. For an explanation please refer to [30].

In the variable length *Options* field TCP senders can transmit control information for extensions of TCP that have not been foreseen in its original specification. Relevant options for this thesis are the *Maximum Segment Size* option (see 1.4.3) and the *Timestamp* option (see 2.3.5). An option consists of a 1 byte *kind*[5], a 1 byte *length* and zero or more *parameters*[6].

Note that the payload length is not provided within the TCP header but that the whole segment's size is determined by the network protocol packet in which the TCP segment is embedded.

1.4.3 Connection Setup

The TCP protocol requires both endpoints to agree to communicate with each other. Therefore, one application must perform a *passive open* by indicating to the operating system that it wants to receive incoming connections on a specified port number. The operating system then assigns the desired port number to the corresponding process if it is still available. The other application must perform an *active open*, that means that it requests its operating system to connect to the remote endpoint. The two TCP implementations then exchange messages to establish the connection. Upon success the two applications can exchange data until one of them closes the connection.

TCP uses a *three-way handshake* procedure to establish a connection. The messages exchanged are shown in Figure 1.6 (assuming that no message needs to be retransmitted and that only one side initiates the connection).

The first segment has only the SYN bit set in the flags field. The second one in the opposite direction has the ACK bit set, indicating that it acknowledges the preceding SYN segment, as well as the SYN bit for continuing the handshake. The third segment has only the ACK bit set

[5]for example 0x04 for Maximum Segment Size or 0x08 for TCP Timestamp

[6]for example a proposed maximum segment size or a TCP timestamp and echo

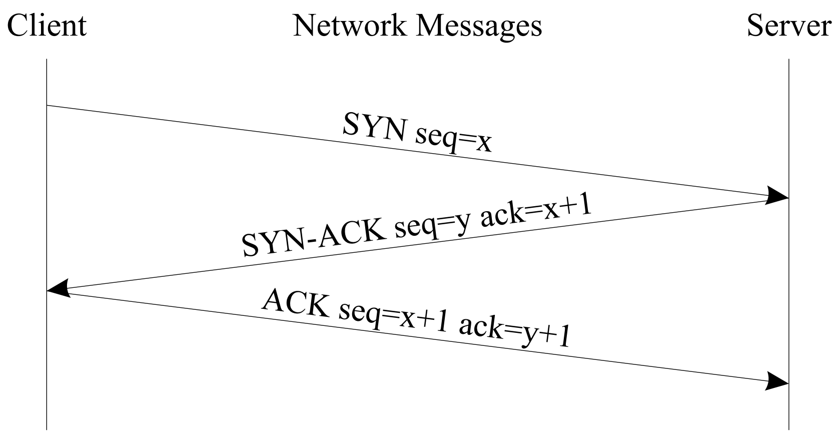


Figure 1.6: Three-way handshake sequence diagram

and thus acknowledges the preceding SYN-ACK segment. We will refer to the three handshake segments as *SYN*, *SYN-ACK* and *ACK*.

Besides guaranteeing that both sides are ready to transfer data, the three-way handshake exchanges the (randomly chosen) initial sequence number in both directions.

During connection initiation the *maximum segment size* (MSS) is negotiated by including the *MSS option* in the SYN and the SYN-ACK segment (see 1.4.2). The maximum segment size is the minimum of the proposal of both sides or 536 bytes if one side does not send an MSS option. The size includes the (variable) length of the TCP Options field.

For a detailed description of the procedures for closing or resetting a TCP connection please refer to [6].

1.4.4 Sending Data and Detecting Loss

TCP uses a sliding windows mechanism similar to the one introduced in section 1.3.4. The TCP receiver advertises the maximum window it can handle (which is the available size of its receive buffer) in the window field of the TCP header. Additionally the sender adapts the window size depending on network conditions in order to prevent the network from being flooded with more packets than it can handle[7]. The actual window size used by the TCP sender is the minimum of these two constraints.

The TCP sliding windows mechanism operates at byte level rather than on segment level, i.e. the unit of the header fields and TCP state variables is bytes rather than segments.

Segments can be lost or delivered out of order, so at any time the receiver will have reconstructed zero or more bytes contiguously from the beginning of the stream, but may have additional pieces of the stream from segments that arrived out of order. The receiver always

[7]We will see in 1.4.5 that this limitation has been introduced after the original TCP specification.

acknowledges the longest contiguous prefix received correctly. The pointer in the *acknowledgement* field of the TCP header indicates the highest byte position of the contiguous prefix plus one, which is the same as the next segment number the receiver expects to receive.

This method is called *cumulative*. Cumulative acknowledgements are easy to generate and unambiguous. Moreover a lost acknowledgement does not force retransmission if the acknowledgement of a subsequent data segment is received before the timer expires.

A major drawback of this method is that if one data segment is lost, the sender does not get feedback about the delivery of subsequent data segments but only about the position of the longest complete prefix. So when a timeout occurs, the sender has to chose between two potentially inefficient responses: If the next data segments have been delivered, retransmitting all of them is a waste of bandwidth, whereas if the subsequent segments got lost as well, awaiting the new acknowledgement number after each retransmitted segment is unnecessarily time consuming.[8]

In the Internet the network delay between two hosts is highly variable and can change over time. Therefore, TCP does not use a fixed value for its retransmission timer but calculates the time-out continuously. The time after which a segment is retransmitted if an acknowledgement has not been received depends on previously measured round trip time samples as well as on their variance.

RFC 1122 [5] allows a TCP receiver to delay acknowledgements and to send an acknowledgement only for every second data packet. The maximum delay is limited to 500 ms. Current implementations use a limit of 200 ms. In an interactive connection where one host sends a request and expects a response from the other host, the delay enables the acknowledgement to be *piggybacked* to the generated response data, saving one segment. In a bulk connection where one host sends a large amount of data to the other only acknowledging every second data packet has the obvious advantage of reducing control traffic from the data receiver to the sender.

1.4.5 Congestion Control

The original TCP specification [25] considered only flow control imposed by the receiver: The window size limits the number of segments that the sender injects into the network before receiving an acknowledgement. In practice, however, the sender must also respond to *congestion* on the network path by reducing its transmission rate if necessary. Therefore, [10] introduced mechanisms for *congestion control*.

[8]RFC 2018 [18] solves this problem by introducing the *Selective Acknowledgement Option (SACK)* which enables the receiver to inform the sender about successfully transmitted out-of-sequence segments while not changing the meaning of the acknowledgement number field in the TCP header. The authors of [19] stated that in 2004 64.7% of the web servers in the Internet they examined supported SACK correctly.

Slow Start and Congestion Avoidance

Congestion is a state where at least one router along the network path receives more packets than it can forward. It will queue the excessive packets. If overload continues and the router's queue grows to its limit, it must drop packets.

The timeout and retransmission mechanism alone would aggravate the state of congestion because it would respond to increased delay by unnecessarily retransmitting packets. So a mechanism is needed to respond properly to network congestion by reducing the transmission rate.

[10] provides the *slow start* and the *congestion avoidance* algorithms for congestion control, which are nowadays part of every TCP implementation. The author assumes that packet loss in the Internet is in the majority of cases an evidence for congestion rather than for corruption.

The author introduces the *congestion window* variable (to which we will refer to as `cwnd` in the following): The TCP sender has to ensure that the amount of unacknowledged data (data *in flight*) does not exceed `cwnd`. Note that the constraint for the amount of unacknowledged data not to exceed the receiver's advertised window is also still valid.

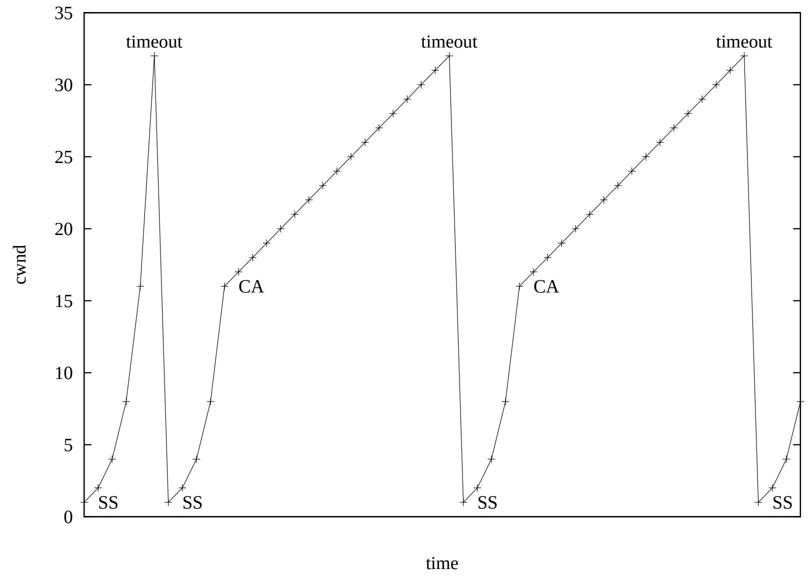


Figure 1.7: Developing of the congestion window during slow start and congestion avoidance

At the beginning of a connection the TCP sender is in *slow start* and initializes `cwnd` to one segment and increases it by one segment upon the reception of every ACK. This results in an exponential growth of `cwnd`: After reception of the first ACK, `cwnd` grows from one to two and two segments will be sent. Upon arrival of their corresponding two ACKs, `cwnd` will be increased from two to four and so on[9]. The slow start phase is marked with "SS" in figure 1.7.

[9]So the term *slow* is in fact misleading.

Every time congestion is observed (indicated by a retransmission timer to expire), the TCP sender saves half of the current `cwnd` (but at least one segment size) in the *slow start threshold* variable (*`ssthresh`*), resets `cwnd` to one segment and continues to perform slow start.

As soon as `cwnd` exceeds `ssthresh`, the TCP sender enters *congestion avoidance* where it widens the congestion window only by one segment per window size[10] which results in linear growth. The congestion avoidance phase is marked with "CA" in figure 1.7.

The 4.3 BSD Tahoe Implementation: Fast Retransmit

A TCP receiver always acknowledges the longest contiguous prefix received and thus sends the same acknowledgement number more than once if out of order segments arrive[11]. Thus, the reception of so called *duplicate ACKs* can be an indication of packet loss. Since it might also be an indication of reordered segments, the TCP implementation of the operating system *4.3 BSD Tahoe* assumes packet loss only after the reception of three duplicate ACKs[12]. In that case it retransmits the expected segment immediately without waiting for the retransmission timer to expire.

In 4.3 BSD Tahoe the sender then enters nevertheless slow start with `cwnd` reset to one segment as depicted in figure 1.7.

The 4.3 BSD Reno Modification: Fast Recovery

The operating system's next version *4.3 BSD Reno* introduced a new method called *fast recovery* (described in detail in [31]) that changes the behavior after fast retransmit: When three duplicate ACKs are received, set `ssthresh` to half of `cwnd`. Set `cwnd` to `ssthresh` plus three segments. For each subsequent duplicate ACK increment `cwnd` by one (and thus transmit another segment). The next time new data is acknowledged, set `cwnd` to `ssthresh` and leave fast recovery. This behavior is depicted in figure 1.8.

The rationale behind this modification is that the reception of a duplicate ACK does not only mean that a segment has been lost, but also that a subsequent segment has arrived at the receiver and that the data flow between source and destination is not completely disrupted.

Note that the fast recovery algorithm does not apply after a timeout. In that case slow start is entered as described earlier.

[10]Implementations in fact increase `cwnd` by 1 / `cwnd` per received ACK. According to [30] many implementations add another segment size divided by 8.

[11]In fact the original specification [25] is ambiguous about whether or not an ACK segment should be sent when an out of order segment was received. However, all current implementations do so.

[12]Note that this is *four* segments with the same acknowledging number.

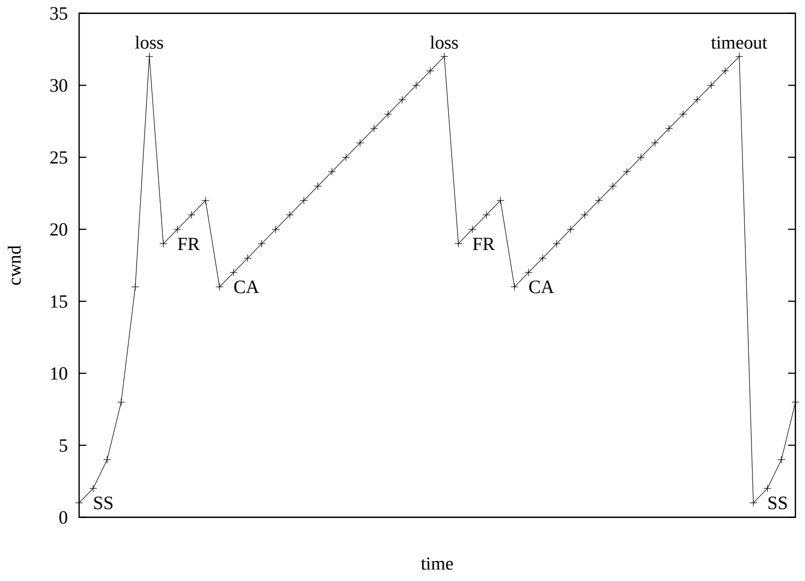


Figure 1.8: Developing of the congestion window if Fast Recovery is used

Increasing TCP's Initial Window

In 1999 RFC 2581 [4] allowed to initialize the congestion window at the beginning of a connection to up to two segments.[13] This modification is nowadays widely accepted and implemented. RFC 2414 [3] experimentally allows an initial congestion window of up to four segments depending on the maximum segment size.

The main advantage is in conjunction with delayed ACKs (see 1.4.4): With an *initial congestion window (`icwnd`)* of only one segment the receiver may delay the sending of the first ACK by up to 500 ms. Since the specification dictates to send an undelayed ACK for at least every second segment, the increase of `icwnd` to at least two segments avoids this needless delay during the first round trip.

The more obvious advantage is that an increased `icwnd` can reduce the transmission time by one round trip time, which is especially useful for small data transfers such as web access.

The 4.4 BSD NewReno Modification

The *4.4 BSD NewReno* version of TCP [8] introduces another modification in the Fast Retransmit algorithm in order to respond properly to multiple losses within one window.

Therefore, during fast retransmit it distinguishes between ACKs that acknowledge the whole window and ACKs that acknowledge only part of it, which means that the loss that triggered fast retransmit was not the only one.

[13]This does *not* affect the congestion window size after a loss.

When entering fast recovery, i.e. after reception of three duplicate ACKs, the new variable *recover* holds the next sequence number to be sent, which is the acknowledgement number expected when all segments currently in flight will have been acknowledged by the receiver. During fast recovery, arriving ACKs will be compared to that stored value. If the acknowledgement number is smaller, it is a *partial ACK* and `cwnd` will be deflated by the number of bytes the ACK acknowledges minus one segment size. Otherwise `cwnd` is set to `ssthresh` like in the Reno version and fast recovery mode is left.

1.5 Active vs. Passive Examination

There are two major ways to examine TCP communication: One can either send specifically prepared packets to a TCP implementation in order to provoke certain behavior and thus create TCP traffic in an *active* way. Or one can observe TCP traffic between two hosts by recording packet traces and analyzing them, which we consider *passive* because that way we do not influence the traffic.

The active approach is very useful if we want to examine only a small number of hosts. It is indispensable for studying the response to *unusual* behavior such as packet loss: The modified TCP sender can just leave out one or more data segments to simulate loss, whereas we would have to wait until loss eventually occurs if we examined traffic between normal TCP senders. The active approach is furthermore the only way to study response to *abnormal* behavior as it is the case when trying to exploit security vulnerabilities: Susceptibility to the problem described in [16][14] for example can only be probed if a purposefully modified ICMP packet is sent to the particular host.

The main advantage of passive measurement is that you can conduct enquiries about a large number of end-to-end connections by recording traffic at a central network switching point and analyzing it. The authors of [12] for example used traffic from a Tier-1 Internet service provider that they found representative to infer the round trip time distribution in the Internet and to find out what is limiting throughput in the majority of cases. Another case where only passive but not active measurement is possible is the examination of the behavior of clients, because you usually cannot connect to them: In [19] the authors wanted to learn about web client behavior by tracking traffic generated by their web server.

1.6 Analysis of Packet Traces vs. Access to TCP State Variables

The passive approach of network tomography can further on be divided into two major categories: Either you can observe segments from a point on the network path *between* the two end

[14]"Multiple TCP/IP implementations do not adequately validate ICMP error messages. A remote attacker could cause TCP connections to drop or be degraded using spoofed ICMP error messages."

hosts — which is the more widespread way — or you have access to certain state variables of the communication software on one or both end hosts.

The former approach has been discussed in 1.5. For the latter one it is obviously necessary to have access to at least one end host, which makes it impractical to examine a large number of connections. Besides, the source code of the TCP implementation must be available to get access to the variables. If these preconditions are fulfilled this method gives much more precise values than the ones derived by packet analysis.

One software that enables access to the values of the TCP implementation of the *Linux* operating system is the *Web100 toolkit* [2].

Chapter 2 will be devoted do the analysis of packet traces and chapter 3 will treat access to the TCP state variables using the Web100 toolkit.

Chapter 2

Analysis of Packet Traces

2.1 Introduction

Analysis of packet traces is the most common discipline in network tomography. It consists of three steps: First a trace of packets must be recorded at some point of the network. Then the recorded packets are filtered and interesting connections are selected.[1] The main activity is the off-line examination of the data either with existing tools such as *tcptrace* [21] or with self-written software.

Unfortunately in certain cases the traffic recorded at the measurement point does not exactly correspond to the traffic arriving at the receiving host: Packet loss, permutation and duplication can occur between the measurement point and the receiver or the recording software or hardware might miss packets itself. If only the header without the payload is recorded, the analysis tool cannot check if the receiver will drop the packet due to an incorrect checksum. Furthermore due to changes in routing parts of the traffic might be redirected so that they do not pass at the measurement point.

In the following we will focus on inferring the *congestion window* (`cwnd`) and the *round trip time* (RTT) from packet traces.

2.2 The Congestion Window

The idea and algorithm presented in this section are derived from [12]. During implementation and testing a lot of unsolved issues and ambiguities have been discovered which we will outline as well.

[1]The term “interesting” can be diverse: E.g. one might want to omit incomplete connections, i.e. those with missing beginning because parameters negotiated during connection setup cannot be restored easily.

2.2.1 Motivation and Idea

As described in 1.3.4, the throughput of a connection depends linearly on the window size. The congestion window has thus great influence on throughput. If we know the size of the congestion window we can furthermore determine if a TCP connection is starved for application-data[2] by comparing `cwnd` to the amount of data actually *in flight*. Carefully observing the manner in which the congestion window changes in response to loss also allows to distinguish the different TCP flavors described in 1.4.5 or even to discover non-conformant TCP implementations.

The basic idea of the algorithm developed by the authors of [12] is to simulate the state of the two TCP hosts by assuming that packets observed at the measurement point are an exact representation of what the hosts receive with some time lag.

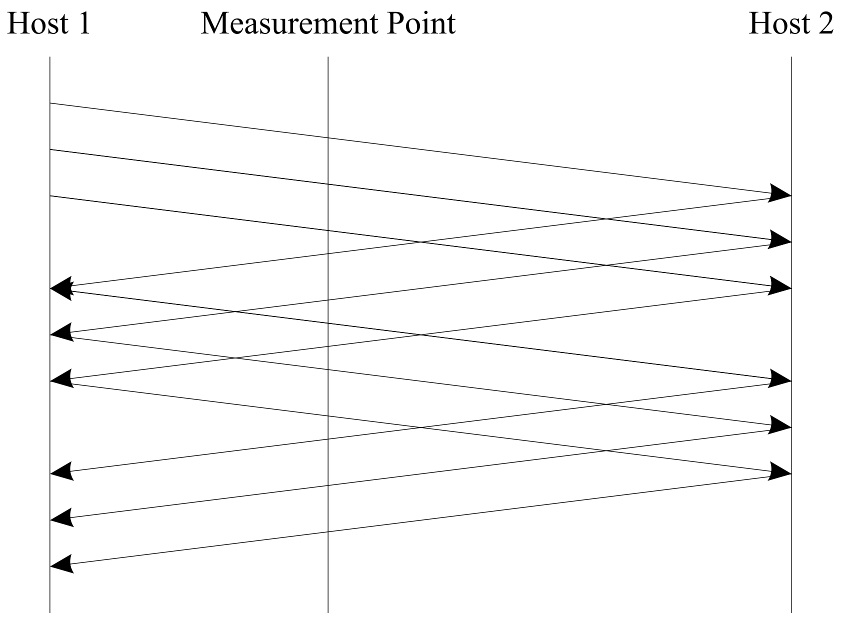


Figure 2.1: TCP congestion window estimation

Therefore, it is necessary that the input trace is bi-directional, i.e. that traffic in both directions is available as input. In the Internet routing is symmetric under normal circumstances, so that this is not a very restrictive constraint.

2.2.2 The Algorithm

The TCP simulation takes the form of a finite state machine (FSM) that changes state for every segment it receives. The state is represented by the TCP variables `cwnd`, `ssthresh` and

[2]That means that the connection could support a higher transfer rate if more data was available from the sending application.

the current congestion avoidance mechanism: slow start, congestion avoidance or fast retransmit. `cwnd` is initialized to two segments since this is the most widespread value for `icwnd`.[3] `ssthresh` is initialized to 65535 bytes. The initial congestion avoidance mechanism is slow start.

This "streaming" design allows for high performance since the algorithm processes every segment only once and does not backtrack nor reverse previous state transitions. Hence, it is suitable for analyzing a huge amount of concurrent connections.

Since there are different flavors of TCP the algorithm instantiates multiple FSMs, one for every flavor to be simulated.

Every FSM reacts to an acknowledgement of new data by increasing `cwnd` appropriately depending on the current congestion avoidance mechanism (slow start, congestion avoidance or fast retransmit). It reacts to the reception of three duplicate ACKs according to its flavor. After the reception of new data `cwnd` is reset to two segments.[4] This is a heuristic for the requirement in RFC 2581 [4] to reset `cwnd` to `icwnd` after no data has been sent for a longer period than the retransmission timeout. Note that this heuristic fails for truly bidirectional connections where both hosts send data at the same time. This is not common for most network applications, however, we have observed this with peer-to-peer communication over the eDonkey protocol and Internet telephony using the *Skype* software.

For every data packet sent, every FSM instantiation verifies if this is permitted according to the current value of `cwnd`. If the amount of sent but not yet acknowledged data is greater than the assumed congestion window, a violations counter is incremented. We assume at any time that the FSM with the lowest number of violations represents the sender's actual TCP flavor.

In many cases the flavors are undistinguishable since their behavior only differs in case of packet loss.

2.2.3 Sources of Inaccuracy

Methodical Deficiencies

As already mentioned there is a time lag between the moment the measurement point observes a segment and the moment the segment arrives at the receiving host resulting in a change of TCP state. Consequently subsequently observed packets in the opposite direction might have been sent before the change of state.

This poses a severe problem to flavor identification: When three duplicate ACKs are observed at the measurement point, the algorithm assumes `cwnd` to be decreased. This is correct, but new data segments might already have been in flight before the sender actually decreased its window. Since the algorithm performs the change in state immediately, it will misinterpret

[3] The pseudo code in [12] initializes `cwnd` to one segment, which is a choice that does not represent reality well.

[4] In the pseudo code in [12] it is reset to only one segment.

following data packets exceeding *its* simulated `cwnd` value as violations falsifying flavor identification.

A related problem not at all discussed in [12] is that the flight size — that is the number of packets sent but not yet acknowledged — cannot be trivially determined from the measurement's point of view. The observation of a data segment increases the flight size, whereas the observation of an acknowledgement decreases it. The further the measurement point is away from the sender, the later it sees data segments and the earlier it sees the corresponding acknowledgements. Consequently the inferred flight size is smaller than the one perceived by the sender, up to near zero if the measurement point is near the receiver.[5] Consequently the further the measurement point is away from the sender the less likely it is to observe violations and the more likely it is that the three tested flavors are indistinguishable.

Furthermore the algorithm detects loss only if indicated by three duplicate ACKs and not by a timeout.

Another scenario not well-handled by the algorithm is whether the application suspends the generation of new data for more than the retransmission timeout. RFC 2581 [4] demands that the congestion window is reset to `icwnd` in that case.

Problems Due to Multiplicity of TCP Implementations

More inaccuracies arise due to the great number of different TCP implementations deployed in the Internet that a passive approach is unable to distinguish. The three principal flavors described in 1.4.4 do by far not cover all prevalent implementations.

Padhye et al. [22] and Medina et al. [19] examined the evolution of TCP extensions, implementation details and violations of the TCP standard in 2001 respectively in 2004 by actively probing the most frequented web servers in the Internet. By sending purposefully constructed TCP segments to the servers the authors provoked responses that revealed certain implementation details. The insights relevant for `cwnd` estimation will be presented in the following.

The initial congestion window (`icwnd`) should be one or two segments according to RFC 2581 [4] or up to four segments according to the experimental RFC 2414 [3]. Medina et al. [19] measured that 42% of the probed web servers use an `icwnd` of one segment, 54% set it to two segments, 5% to three or four and 1% to more than four. The authors of the algorithm [12] claim that they were able to infer `icwnd` by counting the number of data segments observed before the first ACK segment. However this is *not* possible if the measurement point is too near the receiver: When the first ACK is observed, more data packets mights already have been in flight before the observed ACK arrives at the data sender resulting in an underestimation of `icwnd`. Furthermore this method does not work for a small advertised receiver window (`rwnd`): In that — rather theoretical — case it is ambiguous if `rwnd` or `icwnd` limits the number of segments

[5] In the latter case the delay between a data segment and its corresponding acknowledgement should be minimal so that never more than one segment *seems* to be unacknowledged.

in the first flight. If `icwnd` is misjudged, all following `cwnd` estimates will be incorrect as well since every change depends on the previous value.

Medina et al. [19] discovered another detail wrongly implemented in 3% of the hosts: They do not halve the congestion window after loss. The algorithm would thus underestimate `cwnd` if applied to one of these hosts.

RFC 2861 [9] specifies that `cwnd` should not be increased as described in the original TCP specification if the sending application or the advertised receiver window is limiting throughput in order to avoid bursts after the limitation disappears. Medina et al. [19] measured `cwnd` after a period of receiver limited throughput: About 70% of the probed web servers used some limitation of `cwnd` growth, but more than half of them limited growth only less than demanded by RFC 2861 [9]. About 16% seemed to perform slow start normally, and 0.6% showed a congestion window even larger that after regular slow-start. Since the behavior after application or receiver limited throughput is so diverse and cannot be inferred in a passive approach, this observation adds great uncertainty to the algorithm in those cases.

2.3 The Round Trip Time

2.3.1 Definition and Motivation

The round trip time (RTT) is defined as the total time between a sender transmitting a segment and the reception of its corresponding acknowledgement. This interval includes propagation, queueing, processing and other delays at routers and end hosts [27].

The RTT is an important metric to evaluate the performance of a TCP connection: Assuming a constant window size, the throughput of a TCP connection depends inversely proportional on the RTT.

Besides, changes in the RTT during the lifetime of a connection are of great interest since an increase might indicate that the queue of an intermediate router is filling up which in turn is likely to be an indication of network congestion.

Inferring the round trip time from a measurement point somewhere between the two end hosts is by far not as trivial as for the TCP implementation at the sending host. Researchers have developed numerous approaches in recent years which will be presented together with their preconditions, strengths and drawbacks in the following sections.

2.3.2 SYN-ACK Method

One way to measure only one round trip time sample per connection is to observe the SYN and the ACK segment exchanged during three-way handshake. This simple method has been described in several publications, e.g. in [14].

The idea is to take the time interval between the SYN and the ACK segment (first and third arrow in figure 2.2) as an estimate for the round trip time. If the delay jitter that is introduced in

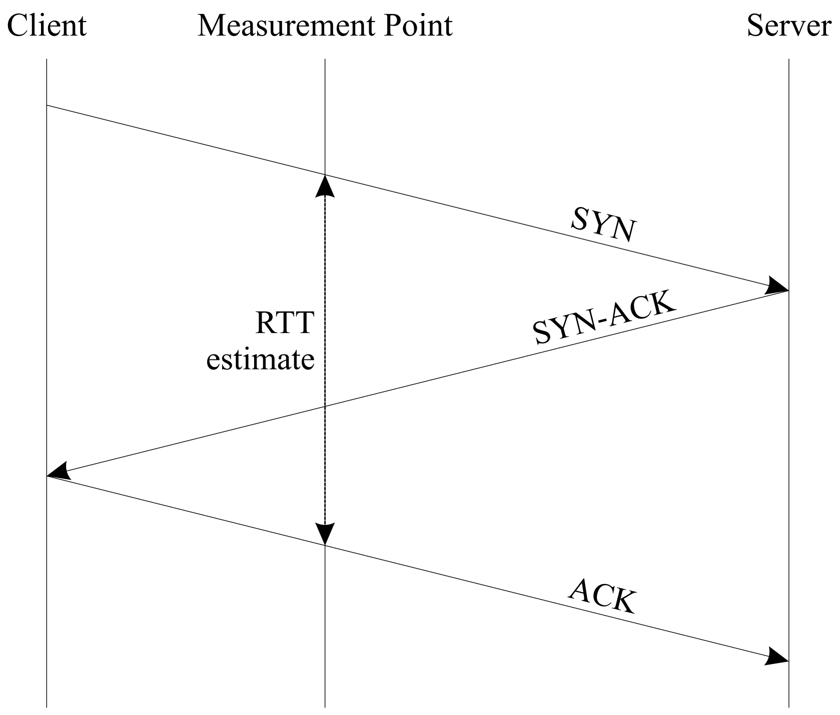


Figure 2.2: RTT derived from three-way handshake

the network between these two segments is not significant, the measured interval corresponds accurately to the delay perceived by the TCP client[6].

If the initial SYN is retransmitted due to a time-out, its last occurrence must be considered. If the ACK is lost before the measurement point, the interval between the last SYN and the first ACK may include a retransmission timeout and the estimate must be discarded in that case. RFC 2988 [24] demands an initial retransmission timeout of 3 seconds.

Inaccuracy is introduced only if the SYN-ACK segment from the TCP server or the ACK segment from the TCP client is delayed. One reason for that can be firewalls or packet filters. However, practice shows that this delay is negligible.

If we have access to a bidirectional trace we can infer the approximate position of the measurement point relative to the two hosts as a side effect: If the interval between SYN and SYN-ACK is very small compared to the interval between SYN-ACK and ACK, the measurement point is located near the TCP server and vice versa. If there is no great difference between the two intervals, the measurement point is somewhere in the middle of the path.

If we have only access to the segments transmitted from TCP server to client, [27] proposes to measure the time interval between the SYN-ACK segment and the first observed data

[6]We will refer to the host that requested the connection as the *TCP client* and to the host that accepted the connection as the *TCP server*.

segment. This is very doubtful since depending on the application the TCP server does not necessarily send data as soon as the connection is established, which would cause an overestimation of RTT.

Surprisingly, Zhang et al. [35] claim that "the SYN-ACK handshake tends to underestimate the actual round trip time". However, Shakkottai et al. [27] contradict and state clearly that the three-way handshake method does *not* underestimate the round trip time compared to two other methods they developed examining the data flow during the connection. One possible explanation for the observation by the former ones might be that a considerable part of the TCP connections they analyzed contributed to queue growth and thus provoked an increase in round trip time during data transfer. An example is given in figure 2.3.

That would mean that the estimate from three-way handshake is only improper as an average RTT value for the connection but must rather be considered as an estimate for the round trip time only valid at the beginning of the connection.

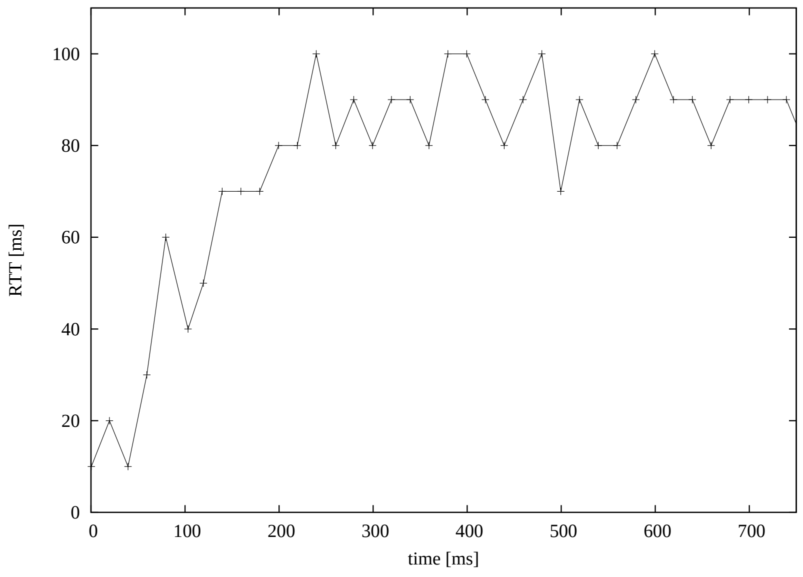


Figure 2.3: The round trip time increases due to queueing.

2.3.3 Measurement Point near the Sender

If we have inferred from the SYN-ACK method that the measurement point is located near one of the two hosts, say on the same LAN or even on the same host, we can easily measure the round trip time for every data segment sent from that host: We only need to associate the sent data packet with the received corresponding ACK segment as shown in figure 2.4. If no loss has occurred, the acknowledgement number of the ACK in question is the data segment's sequence number plus its length. If loss has occurred, the duplicate acknowledgement number is ambiguous and the association cannot be made. This is exactly how that the TCP sender measures the round trip time in order to adjust its time-out value (see e.g. [6]).

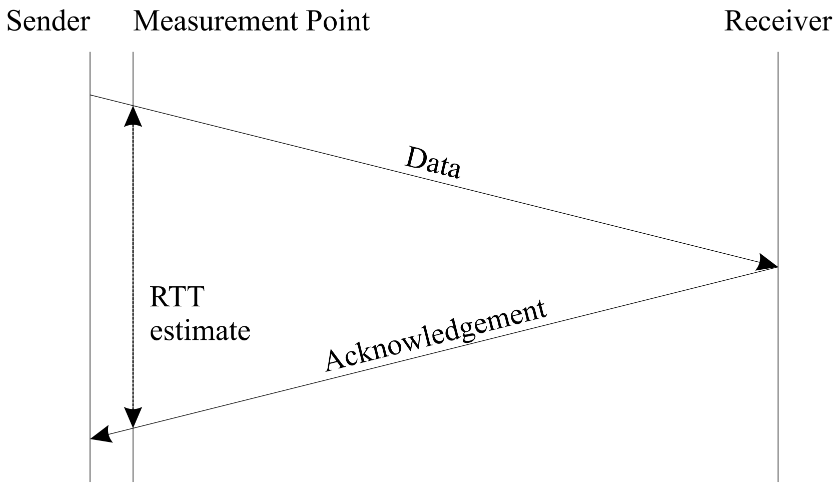


Figure 2.4: RTT estimation with measurement point near the sender

The only inaccuracy is that we neglect the propagation delay from the sender to the measurement point and back, which is very low given that the trace was recorded near the sender.

An uncertainty worth discussing is the use of delayed acknowledgements (see 1.4.4) which is widely deployed in today's Internet: The RTT estimate includes the delay imposed by the receiver. One approach could be to detect and filter delayed acknowledgements. Since RFC 1122 [5] demands an undelayed ACK for every second data segment, one could develop a heuristic for determining which ACK has not been delayed. On the other hand one can argue that the delay imposed by the receiver is part of the round trip time perceived by the sender and should thus also be part of the estimated RTT.

2.3.4 Based on Knowledge about the Congestion Window

The method presented in this section is derived from and explained in detail by Jaiswal et al. [13].

The approach described in the previous section fails if the measurement point is *not* located near the TCP sender. The delay from the sender to the measurement point and back could not be neglected in that case. The basic idea of Jaiswal et al. is illustrated in figure 2.5. Since we are not able to directly measure the sender's RTT sample, we instead measure *(i)* the round trip from the measurement point to the receiver and back and *(ii)* the round trip from the measurement point to the sender and back. The sum of the two delays is our estimate of the round trip time.

If the two measured delays did not change from one segment to the next, the RTT estimate

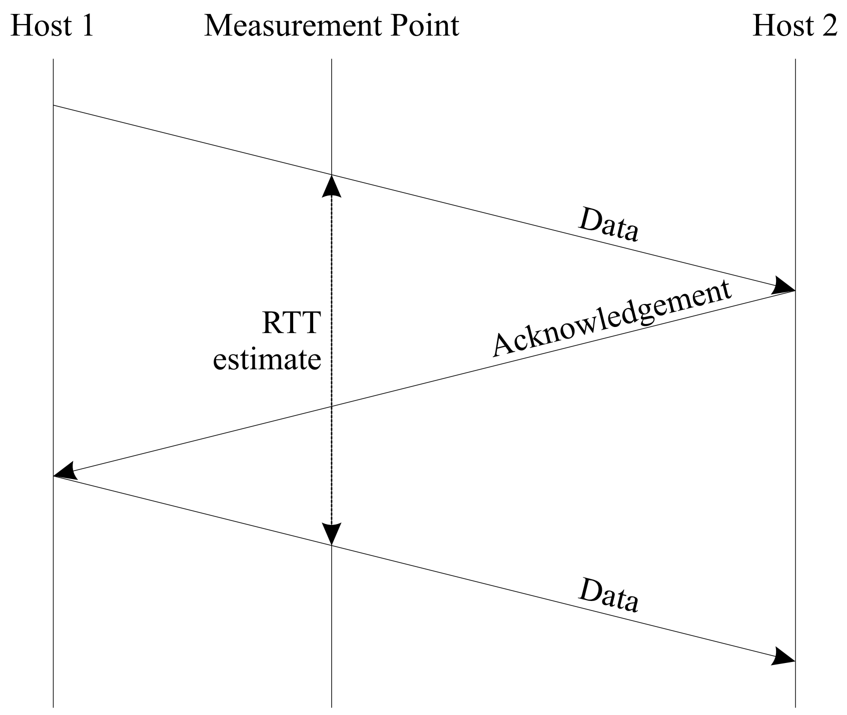


Figure 2.5: RTT estimation based on knowledge about the congestion window

would be exact. In reality however, these quantities vary and the estimate can only be approximate.

The association of the data segment to the corresponding ACK segment is trivial and has been described in 2.3.3. We will refer to it as the *first association* in the following.

The far greater challenge is to infer which next data segment has been triggered by the ACK, which we will refer to as the *second association* in the following.

Therefore, we need to know the congestion window at the moment the ACK arrived at the sender. As shown in 1.4.5, the TCP sender updates `cwnd` upon reception of an acknowledgement and sends the next pending data segment if the amount of unacknowledged data plus one segment size is smaller than the congestion window *and* smaller than the receiver's advertised window. Hence, the sequence number of the next triggered data segment is the acknowledgement number of the ACK plus the minimum of congestion and receiver window minus one segment size.

That means that the method relies on the algorithm by the same authors described in 2.2 that infers `cwnd`. Its accuracy depends on the correctness of the latter: An overestimation (underestimation) of `cwnd` leads to an overestimation (underestimation) of the round trip time: From the explanation above it follows directly that if `cwnd` is overestimated, a subsequent new data segment (third arrow in figure 2.5) is considered instead of the correct one, resulting in an overestimation of RTT.

If a flow recovers from loss, we must interrupt the RTT estimation because the first association cannot be made any more. This is similar to the RTT measurement performed by the sender. The RTT estimation must resume as soon as the lost segment has been retransmitted. Both conditions are already tracked by the algorithm that calculates the congestion window and can be reused for this purpose.

A problem not at all mentioned by [13] are application limited senders. Assume an application generates data slower than TCP could send them[7]. This would introduce a certain delay between the reception of the acknowledgement and the sending of that new data segment that the algorithm incorrectly would include in the round trip time estimate.

So if we want to use this algorithm without making restrictions on the kinds of traffic to analyze we need to identify the state of application limited transmit.

One possible way to accomplish this would be to consider only those data packets for the first association that use the full MSS and do not have the *push* flag set. The rationale behind these conditions is that a TCP implementation that has more data ready to send would not generate unneeded overhead by using a smaller segment size than allowed. An application would set the push flag to force the TCP to immediately transmit the segment, which is usually an indication that it expects an answer from the remote side and has thus no more data to send instantly[8].

Note that the *Nagle algorithm* [30] can still introduce unrecognized cases of application limited transmit. This algorithm forces the TCP sender to accumulate small chunks of data as long as a non-MSS size segment has not been acknowledged. As a result one might observe full MSS size segments that have nevertheless been delayed by the sender and the heuristic would fail.

2.3.5 Timestamp Method

The method presented in this section is derived from and explained in detail by Veal et al. [33].

The previously presented method has one major drawback: Its accuracy depends on the exact estimation of the congestion window. As we have shown in 2.2 this cannot be achieved in many cases. Thus it would be useful to find a more exact way to find out which acknowledgement triggered the sending of which new data segment.

RFC 1323 [11] introduced a TCP option for improving round trip time estimation at the TCP sender particularly on *long, fat pipes*, that are Internet paths with high bandwidth and high delay like over satellite links for example. We can benefit from this extension for passive RTT estimation as well.

[7]We observed this for example with peer-to-peer applications that can limit the consumed bandwidth to a value chosen by the user.

[8]We have observed though that the Linux 2.4 kernel sets the push flag sometimes even if the send buffer is not empty. In that case an RTT sample would be skipped unnecessarily.

The TCP option contains two 32 bit values: One timestamp set by the sender of a segment and an echo of the last timestamp the sender received. If both hosts support that option, every TCP sender can continuously measure the round trip delay by measuring the interval between the sending of a certain timestamp and the reception of its echo. Note that the hosts do not need a synchronized time nor do they need a common unit of time because they do not interpret the value from the respective opposite host but only echo it.

Figure 2.6 provides an example. The sender transmits a segment at time x. The receiver responds with an ACK at time y and echoes the sender's timestamp x. Upon receiving the ACK, the sender transmits new data at time z and echoes the receiver's timestamp y.

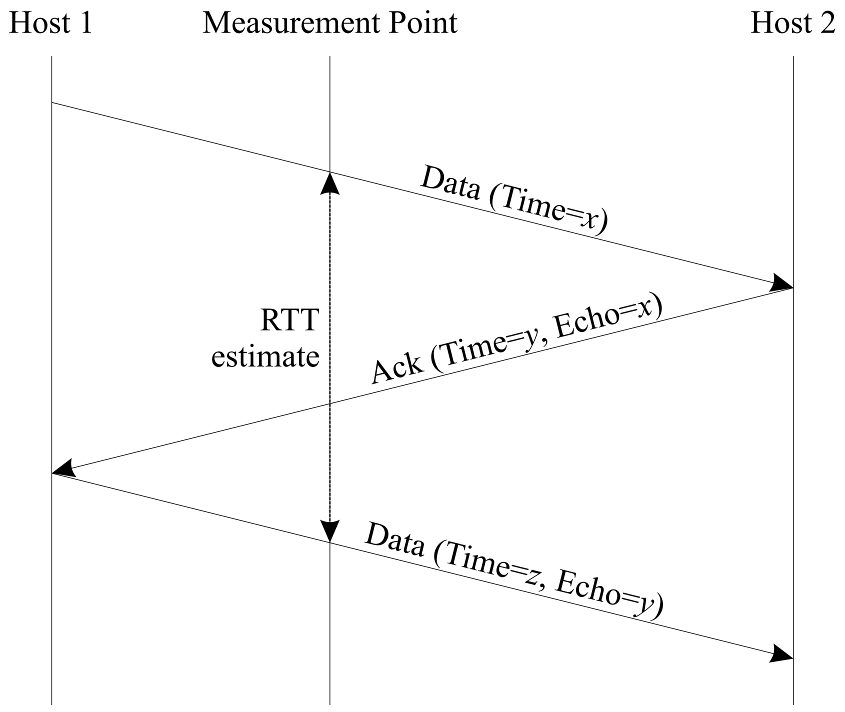


Figure 2.6: RTT estimation based on timestamps

Veal et al. [33] probed 500 well known web servers in an active way and found out that 76.5% of them supported the timestamp option. The authors assume that the deployment of that option will even increase over time. Note however, that both hosts must support the timestamp option so that it will be used in a connection.

We can benefit from this TCP option by replacing our error-prone second association: The data packet that was triggered by the ACK is simply the one that echoes the ACK's timestamp.

The timestamp granularity depends on the TCP implementation. Veal et al. [33] found out with their active probe that most of those web servers who support the timestamp option use a granularity of 10 ms (54.8%) or 100 ms (36.9%). Only few use 1 ms (7.2%) and almost none use more than 100 ms (1.1%).

Even with a fine granularity it may happen that segments carry the same timestamp, for example in a burst after the congestion window has been opened. Since the measurement point cannot determine which data segments caused which ACKs in that case, it must associate only the first segment that carries a certain timestamp with the first echo and discard all following segments with the same timestamp. As a result this method returns at most one RTT estimate per timestamp unit, which is a very acceptable constraint.

Veal et al. [33] propose to use timestamps for both associations. However, we propose to use them only for the second association because that is the one that introduced great inaccuracy as described in the previous section. The first association has worked well as long as no loss has occurred and besides does not suffer from the timestamp granularity problem[9].

The problem of application limited transmissions is the same as described in the previous section. The heuristics to filter samples where this is the case can — and have to — be applied correspondingly.

2.3.6 More Approaches

The following approaches for estimating the round trip time are not relevant for the software developed in the context of this thesis. They are only presented to give an overview about what other methods exist.

Identification of Flights

In [35] Zhang et al. describe their *Round Trip Time Estimator* which is part of their *TCP Rate Analysis Tool (T-RAT)* aiming at determining what is limiting throughput in a TCP connection. The algorithm does not require a bidirectional trace but is in turn judged by the authors as not highly accurate.

The algorithm first generates a set of 27 candidate RTT values exponentially distributed between 3 ms and 3 seconds. Then it groups the packets into potential flights in the following way: P_0 is the first packet of a flight at time T_0. Let P_1 be that packet between $T_0 + RTT$ and $T_0 + 1.7 \cdot RTT$[10] with the largest inter arrival time[11]. Let P_2 be the first packet after $T_0 + 1.7 \cdot RTT$. If $P_2 > 2 \cdot P_1$ assume that P_2 is the first packet of the next flight, otherwise P_1.

The set of flights is then evaluated by trying to match its behavior to that of TCP: If four consecutive flights show *additional* increase in *flight size*, i.e. the number of bytes transmitted in the flight, the state of the first flight is set to *congestion avoidance (CA)*. If four consecutive flights show *multiplicative* increase in flight size, the state of the first flight is set to *slow start*

[9] If we demanded that both the timestamp of the first data segments and the one of the ACK have not been seen before this would be a pretty hard constraint for small round trip delays and coarse timestamp units.

[10] The constant factor takes account of RTT variance. The value of 1.7 is empirically optimal.

[11] The *inter arrival time* is the interval between the arrival of the last and the current packet.

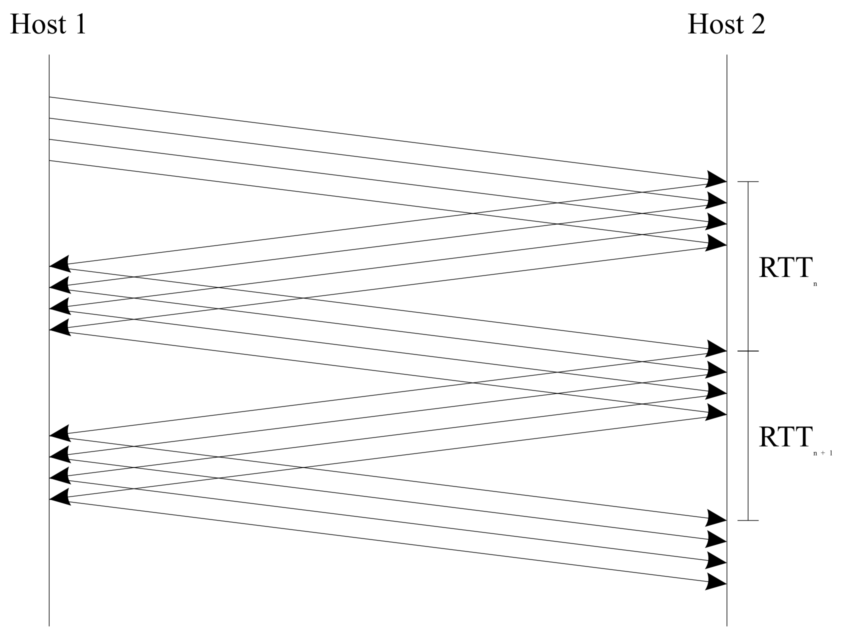


Figure 2.7: Flights of a TCP bulk data transfer: short inter arrival times within the flight, long inter arrival times between flights

(SS). If loss occurred within four consecutive flights or if the bahavior cannot be matched to the the two former TCP states, the state of the first flight is set to *unknown (UN)*. Each candidate RTT is assigned a score amounting to the number of flights in state *CA* or *SS* but not *UN*. That candidate RTT with the highest score is chosen as the RTT estimate.

Flight Method

Shakkotai et al. aim at estimating one average RTT per TCP connection and require only a unidirectional packet trace for their *flight method* [27].

TCP's slow start and congestion avoidance algorithm create so called *flights*, that are groups of segments sent back to back. Since every flight is triggered by the reception of one or more ACKs, the time between the leading edges of two successive flight corresponds to the round trip time (figure 2.7).

From the measurement point's perspective a flight appears as a sequence of segments with nearly identical (relatively small) *inter arrival times (IAT)* followed by one larger IAT.

The algorithm by Shakkottai et al. simply considers the difference between two successive IATs. If the difference is smaller that a defined threshold, it assumes that the corresponding packet belongs to the ongoing flight. Otherwise it assumes that the segment is the beginning of the next flight. The difference between the leading edges of two successive flights is taken as one RTT sample. The connection's estimated RTT is the average of all samples.

The main reason why Shakkottai et al. required only one RTT value per connection is that they examined the RTT distribution among a large number of flows. However, we assume that the samples taken by the algorithm contain so many outliers that only the average of all samples can be trusted. For our purpose of tracking the changes of the RTT over the time of a connection we judge the algorithm to be too imprecise, though.

The authors have also studied the nature of flights and found that flights are not a common phenomenon and that their average size is very small[12]. They conclude that the cause for flights is a small window. The observation that flights can much easier be identified during slow start than during congestion avoidance supports this thesis.

We believe that one technical reason is that queueing in routers contributes significantly to blurring of flights.

Discrete Autocorrelation of Inter-Arrival Times

Veal et al. [33] have enhanced the flight method presented in the previous subsection. First, they do not take the time at which a segment has been observed at the measurement point into account but the TCP timestamp (RFC 1323 [11]) set by the sender. This moderates the blurring introduced by queueing between the sender and the measurement point. Second, they do not measure the pure inter arrival times but infer the round trip time by discrete autocorrelation of the number of packets per time slot. Unlike the flight size method this allows to detect more complex patterns than just alternating bursts and gaps.

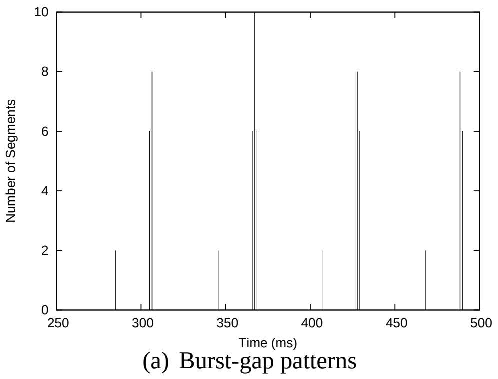
Discrete autocorrelation measures how well a data set is correlated with itself at a certain offset. If the correlation is strong, the data matches its offset closely. The offsets with a high correlation are candidates for the frequency of recurring patterns in the data set. The strength of the maximum correlation is a degree of how repetitive the data set is. An example is given in figure 2.8.

The algorithm by Veal et al. uses discrete autocorrelation to make RTT estimates. It repeats the RTT estimation once per measurement interval T which is supplied as a parameter. During this interval, the number of packets that arrive at timestamp t is stored in array $P[t]$ ranging from 0 to $T-1$. When the count is complete, the discrete autocorrelation $A[l]$ is computed for each offset l from 1 to $T/2$. The RTT estimate is computed as max(A).

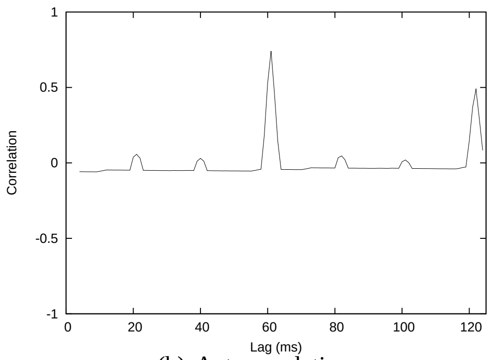
The process is repeated in order to produce multiple estimates over time of the TCP connection.

The algorithm is constraint by timestamp granularity and the measurement interval. According to *Nyquist's theorem*, the sampling resolution must be at least twice the maximum desired frequency to be sampled. For the algorithm of Veal et al. this means that the timestamp granularity has to be at most half of the round trip time. Half of the measurement interval chosen

[12]They constat that flight sizes larger than 7 packets are very rare. This is much smaller than the expected congestion window size.



(a) Burst-gap patterns



(b) Autocorrelation

Figure 2.8: Autocorrelation for self-clocking patterns (taken from [33] with the author's permission)

places an upper bound on the maximum RTT that can be measured since at least two complete round trips are needed to fully compare one round trip with its offset.

Rate Change Method

Shakkotai et al. [27] base their *Rate Change Method* on a fluid view of TCP rather than a packet view and require only a unidirectional trace.

The fluid view does not consider the data flow on a packet level but rather

- the number of bits transferred
- the data rate, which is the number of bits transferred per time, which is the first derivative with respect to time
- the change in data rate, which corresponds to acceleration, which is the second derivative of bits transferred with respect to time.

We know from section 1.4.5 that the change in data rate in congestion avoidance (during periods without loss) is one segment size per round trip time. With this knowledge the authors were able to set up an equation that allows to calculate the round trip time (RTT) against the change in the number of bits transferred (x) per time (t):

$$\mathrm{RTT} = \sqrt{\frac{\mathrm{MSS}}{\frac{d^2x}{dt^2}}} \tag{2.1}$$

Hence, their algorithm considers sets of ten packets each by summing up the size of data transmitted, measuring the time elapsed and inserting both values into the equation above. Since the change in data rate is different during slow start, the algorithm discards the first 15 packets. Also the last measurement is discarded since at the end of a transfer the data does not fill a complete window any more. Finally the algorithm calculates the average of all RTT samples.

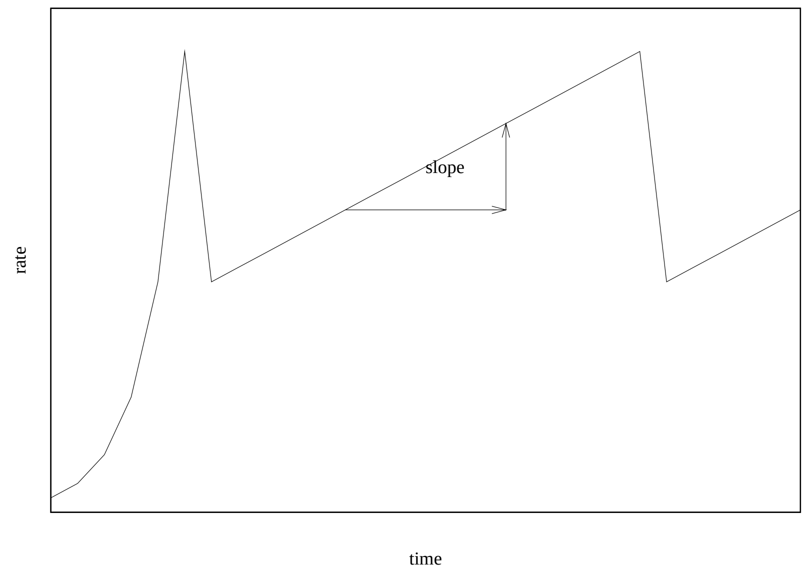


Figure 2.9: Fluid view of TCP: The slope depends on the RTT

We have implemented this algorithm as an experiment and found out that it does not work in cases where queueing has equalized the data rate resulting in a slope that is 0.

2.3.7 Summary of RTT Estimation Methods

We have presented numerous approaches to deduce the round trip time from a packet trace. Those methods that require only unidirectional traces mainly rely on the identification of flights that recur with a frequency that corresponds to the round trip time. Those methods that consider segments in both directions of a connection have the potential of being more exact and returning more samples. The methods that use the TCP timestamp option can only be applied to connections where both hosts support that option. The observation of the messages exchanged during three-way handshake returns only one RTT sample per connection, whereas the other methods track the RTT during the whole connection. Finally, the rate change method has been presented as the only approach based on a fluid model of TCP.

2.4 Requirements

We have written a software that infers the round trip time from packet traces. For this software we established the following requirements.

2.4.1 Choice of Algorithms

The software must implement

- the algorithm by Jaiswal et al. [12] to emulate the congestion window.
- the three-way handshake method, including the decision where the measurement point is located
- the measurement near the data sender
- the two methods that associate three segments to infer the round trip time: the algorithm by Jaiswal et al. [13] relying on the knowledge of the congestion window and the algorithm by Veal et al. [33] relying on the TCP timestamp option, depending on whether or not the TCP timestamp option is present. We will refer to those two as *the three-way algorithms* in the future.

These algorithms have been chosen because the system in which the software will be deployed disposes only of bi-directional TCP traces, and in many cases the traces were actually generated on the sending host so that the trivial method of measuring the delay between data packet and corresponding acknowledgement is often sufficient.

The combination of the timestamp method and the Jaiswal algorithm relying on the knowledge about the congestion windows allows the three-way measurement to be as exact as possible: The timestamp method is less error-prone than the Jaiswal algorithm, but since not all TCP hosts use the TCP timestamp option, the software can fall back to the Jaiswal algorithm.

Since the three-way algorithms only return values for non-application limited bulk transfers, the three-way handshake method guarantees that at least one round trip time sample can be returned for every trace.

The other algorithms mentioned in 2.3.6 are only useful for calculating one average round trip time per connection, but for the software being developed our objective was to be able to track the changes in round trip time over the time of the whole connection as far as possible.

2.4.2 Environment

The software is to be integrated into three software programs:

- a command line tool that reads *tcpdump* traces and prints time / RTT pairs to the standard output
- a command line tool that queries a PostgreSQL database holding traces (see [28]) and prints time/RTT pairs to the standard output
- a PostgreSQL database function that reads a trace from a table and returns a relation containing the time/RTT pairs.

2.4.3 Interfaces

To facilitate the integration into the three environments named above we chose the raw TCP header and options as input format. They should always be reconstructible even if certain environments will only have interpreted data available — such as *tcpdump*'s ASCII output or the database format specified in [28].

For retrieving the round trip time estimates, the frame software must register as notification handler function at the beginning.

2.5 Design and Implementation

The chosen algorithms have been implemented in C++. C++ is a good choice that allows an object oriented and thus more structured software design than pure C, and the written classes can nevertheless be used from a C frame program as it is the case when writing a PostgreSQL module.

The UML class diagram in figure 2.17 shows only those attributes (member variables) and operations (member methods) that are relevant for the `cwnd` and RTT calculation. Details of debugging, configuration and output have been omitted for clarity.

2.5.1 The Connection Class

The `Connection` class represents a TCP connection.

The constructor takes three `bool` parameters so that the calling routine can indicate to the class which measurement methods should be used: the SYN-ACK method, the measurement near the sender and/or the three-way methods.

The calling routine first registers an event handler using the `setNotificationHandler` method. The class and its member classes will call that registered handler whenever a new RTT sample has been calculated.

Then the calling routine passes packets in both directions to the class by calling the `transmitPacket` method. With this design the class can be used in real time environments where the RTT samples are calculated just as the packets are captured.

The `Connection` class tracks the three-way handshake by updating the `TCPPseudoState` member variable upon transmission of a SYN, SYN-ACK or ACK segment. We call the state known to the `Connection` class only a “pseudo state” because actually both hosts have their own state representing the progress of the handshake. The fact that a handshake message can get lost between the measurement point and the receiver is neglected here.

Tracking the three-way handshake is good to infer three kinds of information. First, connection specific parameters such as the maximum segment size (`MSS`) and whether the TCP timestamp option is used (`supportsTimeStamps`) can easily be deduced. Second, the first RTT sample is measured as described in 2.3.2. Third, it can be inferred whether the measurement point is near one of the two hosts: If the delay between SYN and SYN-ACK is at least 10 times the delay between the SYN-ACK and the ACK, the software infers that the trace has been captured near the *server*. If the delay between SYN-ACK and ACK is at least 10 times the delay between the SYN and the SYN-ACK segment, the software infers that the trace has been captured near the *client*. If none of the two conditions are true, we assume that the measurement point is not near one of the two hosts and that the simple method of measuring the delay between data segment and corresponding acknowledgement is not sufficient. The result is stored in the member variable `WANSendDir`.

The `Connection` class furthermore implements the actual round trip time measurement near the sender (if it has deduced that this is possible). The real TCP sender has only one single timer and thus takes only one sample per round trip. The software however holds two queues with the acknowledgement number whose reception is expected and the timestamp when the corresponding data segment has been sent. This allows to take as many round trip time samples as acknowledgements are received. So whenever a data segment in the proper direction is processed the corresponding acknowledgement number (which is the sequence number plus the data length) and the current timestamp are inserted into the two queues. Whenever a pure acknowledgement in the proper direction is processed all queue entries with a smaller acknowledgement number are discarded.[13] The difference between the stored timestamp at the head of the queue and the current timestamp of the acknowledgement is the current RTT estimate, which is exported by calling the notification handler.

2.5.2 The Host Class

The `Host` class represents one of the two hosts of a TCP connection. It is instantiated twice by the `Connection` class after observation of the three-way handshake. The `transmitPacket` method of the `Connection` class calls the `sendPacket` method of one `Host` object and the `receivePacket` method of the other.

The three-way RTT measurement methods are implemented in the `Host` class because both hosts can potentially be bulk data senders and receivers. Simply instantiating two `Host` objects makes case differentiation unnecessary and the design easier to understand.

For the three-way RTT measurement methods (figures 2.5 and 2.6) we need to associate a data segment with its corresponding acknowledgement and that acknowledgement with the data segment which the sending host transmits directly after the reception of the latter one. The first association in both cases is that the acknowledgement number must be the sequence number of the data segment plus the data length. The second association is made either by comparing the TCP timestamp of the acknowledgement to the TCP timestamp echo of the new data segment in case the TCP timestamp option is present or by predicting with the knowledge of the current congestion window (Jaiswal et al.) which sequence number the acknowledgement segment will trigger.

[13]This is for example necessary if the receiver uses *delayed ACKs* or if a previous acknowledgement has been lost.

The original algorithms in [13] respectively [33] can only infer one RTT sample per round trip. However, the `Host` class of this software can handle multiple three-way associations at the same time by storing the information necessary to make the associations in the queues `waitForAck`, `waitForAckTS`, `waitForNewData` and `waitForNewDataTS`.

The queues `waitForAck` and `waitForAckTS` are needed to track the first association: Whenever a full MSS data segment without push flag is sent, the expected acknowledgement number is inserted into `waitForAck` and the current timestamp into `waitForAckTS`.

The queues `waitForNewData` and `waitForNewDataTS` are needed to track the second association: Whenever a pure acknowledgement is received all queue entries with a smaller acknowledgement number are discarded again. If the TCP timestamp option is present, the TCP timestamp is inserted into the `waitForNewData` queue. Otherwise the acknowledgement number plus the current congestion window minus one segment size is inserted. The timestamp from the top of the `waitForAckTS` queue is inserted into the `waitForNewDataTS` queue.

Whenever a data segment is sent, its TCP timestamp echo respectively its sequence number is compared to the beginning of the `waitForNewData` queue: If the beginning of the queue is equal to the TCP timestamp echo of the data segment respectively greater or equal to the sequence number, a new RTT sample, which is the difference between the current timestamp and the beginning of the `waitForNewDataTS` queue, is exported via the notification handler.

The number of consecutive duplicate ACKs is kept in the member variable `numDupacks`. If the number is greater or equal to 3, which means that the sender will infer a packet loss, the measurements is interrupted for the reasons explained in 2.3.4.

2.5.3 The SlaveFSM Class and its Subclasses

The class `SlaveFSM` and its derived subclasses `TahoeFSM`, `RenoFSM` and `NewRenoFSM` mimic the changes of the congestion window depending on the TCP flavor as described in 1.4.5. Furthermore, they count *violations*, that is the transmission of excessive data segments so that the number of current unacknowledged packets is greater than allowed by the assumed congestion window.

The `Host` class instantiates every one of the three classes and passes received and sent segments along to the `receivePacket` respectively `sendPacket` method of every `SlaveFSM` object.

The `Host` class can estimate the flavor of the TCP sender by fetching the number of violations (`getViolations` method) and inferring that the object with the smallest number of violations represents the assumed TCP implementation. In many cases, however, the number of violations is equal and the flavor indistinguishable.

The differences of the Tahoe, Reno and NewReno flavor have been described in 1.4.5 and are depicted in pseudo code in figures 2.10, 2.11 and 2.12. `cwnd` is always initialized to two segments and `ssthresh` to 65535. `rwnd` represents the receiver window; flightsize is the number of sent but unacknowledged packets.

reception of	**action**

pure and new ACK	`if (cwnd <= ssthresh)`
	`  cwnd = cwnd + MSS`
	`else`
	`  cwnd = MSS * MSS / cwnd + MSS / 8`

third or more duplicate ACK	`ssthresh = max(min(rwnd, cwnd) / 2, 2 * MSS)`
	`cwnd = 1`

data segment	`cwnd = 2 * MSS`

Figure 2.10: Pseudo code for `cwnd` handling in TCP Tahoe

reception of	**action**

pure and new ACK	`if (state == DEFAULT)`
	`  if (cwnd <= ssthresh)`
	`    cwnd = cwnd + MSS`
	`  else`
	`    cwnd = MSS * MSS / cwnd + MSS / 8`
	`else if (state == FAST_RECOVERY)`
	`  cwnd = ssthresh`
	`  state = DEFAULT`

third or more duplicate ACK	`if (state == DEFAULT)`
	`  ssthresh = max(min(rwnd, cwnd) / 2, 2 * MSS)`
	`  cwnd = ssthresh + 3 * MSS`
	`  state = FAST_RECOVERY`
	`else if (state == FAST_RECOVERY)`
	`  cwnd = cwnd + MSS`

data segment	`cwnd = 2 * MSS`
	`state = DEFAULT`

Figure 2.11: Pseudo code for `cwnd` handling in TCP Reno

2.6 Integration

2.6.1 Command Line Tool that Reads *tcpdump* Traces

tcpdump is a very common open source tool to capture, filter and process network traffic [26]. It can process live traffic as well as record traffic to a file or process previously recorded traffic from a file.

We wrote a software that can read and parse files generated by *tcpdump* and that passes the

```
reception of      action
---------------------------------------------------------------------
pure ACK          if (state == DEFAULT)
                    if (cwnd <= ssthresh)
                      cwnd = cwnd + MSS
                    else
                      cwnd = MSS * MSS / cwnd + MSS / 8
                  else if (state == FAST_RECOVERY)
                   if (ack >= recover)  // ACK is new
                      cwnd = ssthresh
                      recover = 0
                      state = DEFAULT
                   else  // ACK is partial
                      cwnd = cwnd - amount of ack'd data + MSS
---------------------------------------------------------------------
third or more     if (state == DEFAULT)
duplicate ACK       ssthresh = max(min(rwnd, cwnd) / 2, 2 * MSS)
                    cwnd = ssthresh + 3 * MSS
                    recover = next expected sequence number
                    state = FAST_RECOVERY
                  else if (state == FAST_RECOVERY)
                    cwnd = cwnd + MSS
---------------------------------------------------------------------
data segment      cwnd = 2 * MSS
                  if (state == FAST_RECOVERY)
                    recover = 0
                  state = DEFAULT
---------------------------------------------------------------------
```

Figure 2.12: Pseudo code for `cwnd` handling in TCP NewReno

TCP segments of the first connection encountered along to the classes described in the previous sections.

The *pcap* file format generated by *tcpdump* is very basic: A `pcap_file_header` structure at the beginning of a file gives general information about the captured trace such as timestamp accuracy and captured packet length. The captured Ethernet frames are saved sequentially, each preceded by a `pcap_pkthdr` structure depicted in figure 2.13[14].

Every TCP segment (transport layer) is embedded in an IP packet, and every IP packet (network layer) is embedded in an Ethernet frame (data link layer). So the actual payload is preceded by three protocol headers as depicted in figure 2.14.

[14] `len` is the actual length of the frame, whereas `caplen` is the length captured to the file. Usually one would limit the packet size so that only the headers and not the transmitted data are saved.

```
struct pcap_pkthdr {
        struct timeval ts;     /* timestamp */
        bpf_u_int32 caplen;    /* length of portion present */
        bpf_u_int32 len;       /* length this packet (off wire) */
}
```

Figure 2.13: The pcap_pkthdr structure from pcap.h

protocol unit	**size**
Ethernet header	14 bytes
IP header	20 bytes
TCP header	20 bytes (plus options)
payload	variable

Figure 2.14: Headers and payload for TCP over IP over Ethernet

The software instantiates a `Connection` object, opens the pcap file, skips the `pcap_file_header` structure and steps from packet to packet.

The software stores the quadruple specifying the first TCP connection — IP address and port number of source and destination — and passes all segments belonging to that connection to the `Connection` object. No preprocessing is necessary because the `Connection` class understands the raw TCP header and options format.

The notification handler exports the pair of timestamp and round trip time estimate to the standard output every time it is called.

2.6.2 Command Line Tool with Database Access

M. Siekkinen et al. [28] have proposed "to use a database management system (DBMS) that provides the infrastructure for the analysis and management of data from measurements, related metadata, and obtained results".

The main advantages compared to keeping the traces in files is that the raw data can be annotated and stored in a well-organized way together with the results of analysis, that the analytic cycle can be shortened because iterative analysis can easily re-use previous results, and that the issue of scalability can be left to the DBMS.

The department where this software has been developed uses PostgreSQL 7.4 [32], an open-source rational DBMS with a widespread user community.

The layout of the tables that hold TCP traces is depicted in figure 2.15. The *connection ID* specifies a connection and replaces the quadruple of IP address and port number of source and destination. The *trace ID* specifies a trace which is a collection of connections. Thus a tuple of trace ID and connection ID specifies one TCP connection in the database. One table holds one trace. For a detailed explanation and the full Entity Relationship Diagram please refer to [28].

column name	**type**	**description**
`ts`	timestamp	timestamp
`flags`	character varying(5)	flags in *tcpdump* syntax
`startseq`	bigint	sequence number
`endseq`	bigint	sequence number plus data length
`nbbytes`	smallint	data length
`ack`	bigint	acknowledgement number
`win`	integer	receiver advertised window
`urgent`	integer	urgent pointer
`options`	character varying(100)	TCP options in *tcpdump* format
`cnxid`	integer	connection ID
`reverse`	bit(1)	direction of packet
`tid`	smallint	trace ID

Figure 2.15: Database table layout

We wrote a command line tool that takes a table name and a connection ID as parameters, queries the database for the packets, reconstructs the raw TCP header and options and passes them to the classes described in section 2.5.

The libpq library “is the C application programmer’s interface to PostgreSQL. libpq is a set of library functions that allow client programs to pass queries to the PostgreSQL backend server and to receive the results of these queries.” [32]

The software established a connection to the PostgreSQL server using the libpq function `PQconnectdb` and calls `PQexec` to declare a cursor selecting all packets from the given TCP connection ordered by their timestamps and fetching the returned rows in blocks. The connection is terminated by the `PQfinish` function. For a detailed explanation of how to access a PostgreSQL database from a C program, please refer to [32].

The rows in the database have been generated from the *tcpdump* text output. Hence, certain *tcpdump* peculiarities are still visible and the following transformation must be conducted in order to reconstruct the raw TCP header and options format:

- The data length must be calculated by subtracting `startseq` from `endseq`.
- The sequence (and acknowledgement) numbers have been shifted so that both directions of every connections start with the sequence number 0. The SYN and SYN-ACK segments have kept their real sequence numbers. This shift must be reversed by adding the initial sequence number (per direction) to the numbers from the database.

- Sequence numbers of pure acknowledgements are stored as NULL values. The latest seen sequence number in that direction must be inserted.
- Each set flag is stored as one letter. The acknowledgement flag is always omitted. The original bitfield must be reconstructed and the acknowledgement bit set for every segment except for the pure SYN segment.
- The options are stored as a human readable string. The raw format specified in [25] must be reconstructed.

2.6.3 PostgreSQL Module

PostgreSQL is highly extensible. Developers have several possibilities to write functions: A *Query Language Function* is an arbitrary sequence of SQL statements, returning the result of the last query. *Procedural Language Functions* are offered by loadable modules. There are currently four procedural languages available in the standard PostgreSQL distribution: PL/pgSQL, PL/Tcl, PL/Perl, and PL/Python. Finally, *C-Language Functions* (or functions in a language that can be made compatible to C, such as C++) can be compiled to a *shared object* file — also known as *dynamic library* — and loaded by the server on demand.

We chose to write a C language function that encapsulates the classes described in section 2.5. This decision results from the fact that the classes have been written in C++. Furthermore, since the C code is compiled into native machine code whereas the procedural languages must be interpreted, this is the best-performing way to extend the PostgreSQL database.

The main advantage over the command line tool that queries the database is a substantially increased performance since the data is not accessed through network communication, but the code is running in the same environment as the database server itself. Furthermore, the output can be used in an SQL statement like a regular relation. At best that means that it can be stored in the database as an intermediate result as proposed by Siekkinen et al. [28].

Parameters for C language functions can be base types, composite types or sets.

Base types are the most basic form: They can have fixed or variable size and can be passed by value or by reference. Our function `calc_rtt` takes three base type parameters as input: The table name as a string, the connection ID as an integer and an integer number that specifies which RTT estimation methods shall be applied.

Composite types correspond to a table row in a relational database. Unlike a C structure a composite type does not have a fixed layout but is a set of tuples, that is a set of name/value pairs. This type would be sufficient for our `calc_rtt` function if we wanted to return only one round trip time estimate together with its timestamp.

A *set* is a set of composite types and thus corresponds to a database table. Since our function should return a number of round trip time estimates together with their timestamp, we had to choose this most complicated type as return type. If a function returns a set it is actually called continuously by the database and returns either a composite type every time it is called or signals to the database that it has no more rows to output. Maintenance of state between the calls of the function is provided by PostgreSQL in form of the `FuncCallContext` structure. Another way to maintain state would be the use of global variables.

The declaration of the C function is always the same: The return type is `Datum` and the only input parameter is the macro `PG_FUNCTION_ARGS`, both provided by the PostgreSQL framework. Macros for accessing the input parameters by their number and returning values are provided as well.

The *Server Programming Interface* (SPI) enables the developer of a user-defined C function to run SQL commands. We had to use the SPI to query the table containing the packets of the TCP connection for which the round trip time estimates should be calculated. The Server Programming Interface's API is very similar to the one for accessing an SQL database from an external application. For details please refer to [32].

At the first call our user-defined function `calc_rtt` calculates the round trip time estimates using the classes described in section 2.5. Therefore, the values from the database must be transformed into raw TCP header and options as it was the case for the command line tool described in the previous subsection. The notification handler function that is called for every RTT estimate stores the time/RTT pairs into a linked list that is stored as a global variable. The first raw is returned immediately to the database. For every subsequent call `calc_rtt` will return the next time/RTT pair in the linked list. If the end of the linked list is reached `calc_rtt` cleans up the allocated memory and signals to the database that it has no more rows to return.

Once the function is written and compiled to a shared object file it must be copied to the host where the PostgreSQL server runs, the return types must be defined using the `CREATE TYPE` command and the function must be loaded into the database using the `CREATE FUNCTION` command. The function can then be used like a regular SQL function. An example is given in figure 2.16.

```
SELECT * FROM calc_rtt('honeypot_2005_04_05', 47, 3)
WHERE RTT > 30 ORDER BY RTT;

       ts        |   RTT
-----------------+---------
 12:33:23.118020 |  33.667
 12:33:22.676327 |  45.039
 12:33:24.157109 | 60.3209
 12:33:24.217835 |  60.726
 12:33:24.300945 |   83.11
[...]
```

Figure 2.16: The function `calc_rtt` can be used like a regular SQL function.

2.7 Deployment and Future Use

The output of the two command line tools can be used to generate RTT diagrams, for example by using it as input to the open source software gnuplot [34]. Figure 2.3 for example has been created this way.

However, the far more important result for future research at Institut Eurécom is the database module that has been deployed at the PostgreSQL server used by fellow researchers. One area of application will be presented in the following.

The techniques described in section 2.3.5 will be applied in analysis of limitation cause for obtained throughput of a TCP connection described in [29]. More specifically, RTT samples are required in the test for receiver window limitation where they are used for estimating the congestion window size of the TCP sender at a given time instant. In this way, the RTT samples are not computed with the help of current congestion window value as in the work of Jaiswal et al. [12], which was shown in this thesis to suffer from severe limitations, but contrariwise the congestion window is calculated by reconstructing for every packet at what time it has left the sender respectively for every acknowledgement when it has reached the data sender. Therefore, the time measured at the measurement point must be shifted by half the round trip time.

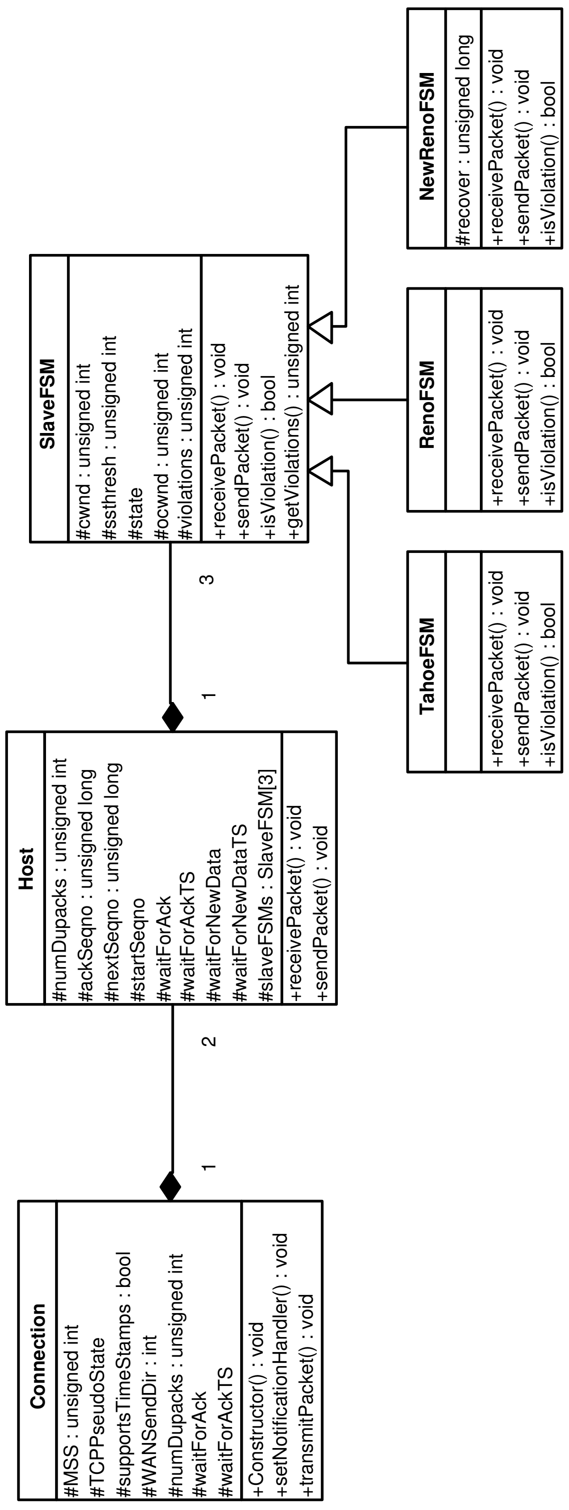


Figure 2.17: UML class diagram for processing of TCP traces

Chapter 3

Access to TCP State Variables

3.1 Introduction

The previous chapter has shown that it is anything but trivial to infer variables such as the round trip time or the congestion window from TCP traces. One self-evident idea to obtain these values is to rather read them out from the operating system of the two hosts involved in the data transmission.

Unfortunately, no operating system allows access to its TCP variables by default. Therefore, we will evaluate the *Web100 Toolkit* that allows access to these variables under Linux.

3.2 The Web100 Toolkit

The *Web100 Toolkit* is a software that allows to query the operating system's kernel for a large number of per-connection TCP state variables. In the current version it comes as a patch for the open source Linux kernel together with a C library to access the values from user space.

The kernel code exports the structure containing the TCP state variables via the Linux */proc* file system. The `libweb100` library contains an API for accessing the variables from within an application. While the interface between the kernel and the library may change on other platforms, the API of `libweb100` is supposed to persist so that applications using Web100 will be portable.

The authors of the Web100 toolkit — mainly researchers from the Pittsburgh Supercomputing Center — see the following use cases for their software [17]: *TCP based measurement* is what will be detailed later in this section. The variables give detailed insight into the remote host, the network and even the communicating applications. *Testing experimental algorithms* is facilitated since writable variables are provided to trigger experimental TCP algorithms or workarounds in the kernel on a per-connection basis. Web100 is furthermore useful for *education* because it illustrates the behavior of TCP and allows to study the changes in the behavior upon changes of certain parameters such as buffer sizes.[1] Finally, the toolkit simplifies *network diagnostics*. The factor that limits throughput — the receiver's advertised window, congestion or the sending application — for example can be read directly from the variables. More complicated questions might be answered by interpreting the large number of available performance variables.

The following groups of variables are provided by the Web100 toolkit [17]:

- connection state (the state of the TCP state machine and flags indicating negotiated protocol features such as TCP timestamps and Selective Acknowledgements)
- IP traffic (number of bytes and segments sent and received)
- average throughput
- triage (instruments that characterize the protocol events that limit TCP sending rate)
- congestion (`cwnd`, `ssthresh`, congestion algorithm, timeouts, duplicate ACKs)
- network path properties (loss, timeouts, duplications, retransmissions, segment re-ordering, round trip time, retransmission timeout, MSS)
- API usage (buffer occupacy)

A complete description of all variables can be found in [1]. The variables that we used in the software presented below will be described later.

3.3 Requirements

We have written a web server that accepts connections from web clients and performs diagnostics on the connection while bulk data is being up or downloaded. The great advantage of using the Web100 Toolkit this way is that only the server needs to be equipped with the modified kernel and library and any client can benefit from it without being modified.

For this web server we have established the following requirements.

It must support diagnostics during selectively up or download of bulk data. For the upload the user has to choose a file from his local filesystem and send it via the HTTP POST method [7]. The bulk download must be realized by including a large HTML comment in the requested web page. The user must be able to choose the size of the bulk data.

The first web page gives a short explanation of the diagnostics conducted and contains one web form for entering the amount of data to download and one web form for choosing a local file to upload.

The second web page contains the large HTML comment if the user has chosen the bulk download and the results of the diagnostics consisting of graphs and textual information.

[1]For example with Web100 we were able to identify a bug in the Linux 2.4 kernel that causes `cwnd` to deflate below `ssthresh` during Fast Retransmit.

3.3.1 Graphs and Values for Bulk Download

The following graphs must be generated from the values obtained during the bulk transfer from web server to client.

- round trip time (RTT)
- retransmission timeout (RTO)
- congestion window (`cwnd`)
- slow start threshold (`ssthresh`)

The following values obtained during the bulk download must be displayed.

- features and parameters negotiated during three-way handshake: Maximum Segment Size (MSS), Selective Acknowledgements option (SACK), TCP Timestamp option, Explicit Congestion Notification (ECN)
- minimum, median and maximum of the receiver's advertised window
- the reason for bandwidth limitation[2]
- throughput
- median, average and root mean square deviation of the round trip time
- amount of data retransmitted

3.3.2 Graphs and Values for Bulk Upload

The following graphs must be generated from the values obtained during the bulk transfer from web client to server.

- the next expected sequence number (*time sequence diagram*)
- instantaneous throughput

The following values obtained during the bulk download must be displayed.

- features and parameters negotiated during three-way handshake: Maximum Segment Size (MSS), Selective Acknowledgements option (SACK), Timestamps option, Explicit Congestion Notification (ECN)
- throughput
- amount of data that arrived duplicated

[2]the receiver's advertised window, congestion or the sending application

3.4 Design and Implementation

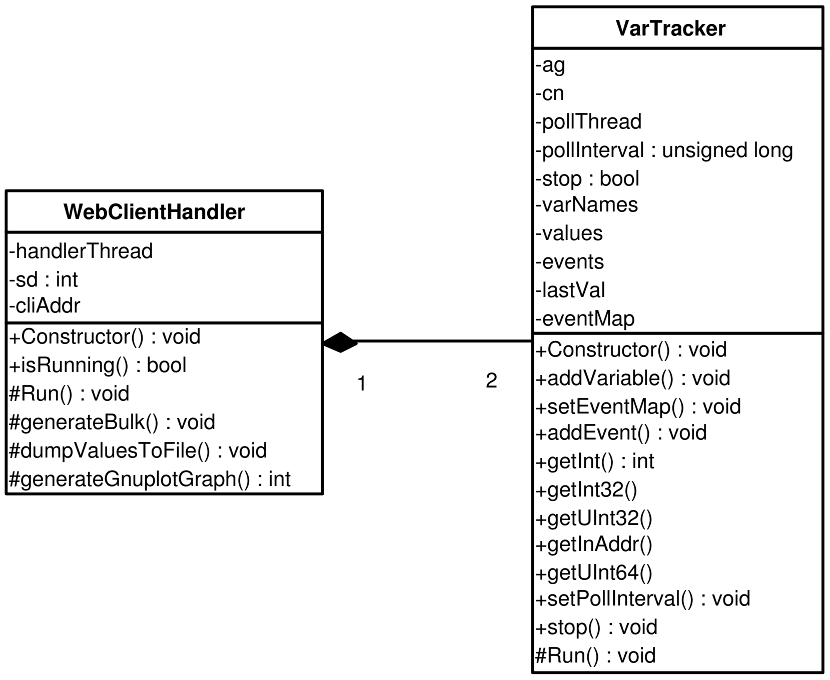


Figure 3.1: UML class diagram for web server

3.4.1 The VarTracker class — a wrapper for Web100

The functions provided by the `libweb100` library are not very convenient to use. First a connection must be selected either by iterating over all current connections, by providing source and destination addresses and port numbers or by providing a Unix *socket descriptor*. Then variables must be looked up before their value can be accessed.

What is completely missing is a notification mechanism that informs the application when a certain value has changed. Consequently a software that uses the Web100 toolkit can either risk to miss values if it does not poll frequently enough or poll too often and thus read the same value more often and consume more resources than necessary.

Furthermore *events* — for example a timeout or the fast retransmission of a segment — are only reported indirectly as a counter will be increased that must be polled so that the application finds out about the event.

We wrote the `VarTracker` class to facilitate access to the variables provided by Web100.

The constructor creates and starts a new thread that performs the polling regularly. This is necessary because reading from or writing to a Linux socket is *blocking*. That means that the according system call will not return until it has completed its task, which may take a certain amount of time in which variable changes may happen that would be missed otherwise.

Getter methods enable access to the current value of a Web100 variable.

The `VarTracker` class can also track Web100 variables and store their sequence of values into a *list*. Therefore, the `addVariable` method must be called with the name of the Web100 variable and a pointer to a *list* data structure.

It can also track events by applying the heuristics described earlier. Therefore, `set-EventMap` must be called once to pass a pointer to a *map* data structure holding a mapping of timestamps to event names. For every event to be tracked the `addEvent` method must be called with the Web100 variable name counting the event and an event name chosen by the developer as parameters.

The method executed by the poll thread is `Run`. It consists of a loop that is repeated until the `stop` method is called. In this loop the designated variables are polled in an ongoing way.

To overcome the missing notification feature and to avoid storing a needlessly high number of values only the Web100 variable `PktsIn` is polled in every cycle. The other designated variables are only polled and stored to the corresponding list if `PktsIn` has changed, that means if a new segment — regardless whether it is a data segment or an acknowledgement — has been received. Since the variables relevant for our application only change upon reception of a segment, this is a very suitable heuristic that allows us to create easy to read graphs that contain time/value points only where they are intuitively expected.

The feature that tracks events works in a similar way. The variable corresponding to the event is polled in every cycle. If it has changed, a time/event name pair is stored in the event map.

3.4.2 The WebClientHandler class

The `WebClientHandler` class handles connections from the web browser, that is it reads the HTTP request from the socket and writes the HTML page that will be displayed by the browser to the socket. At the same it conducts the polling of the Web100 variables using the `VarTracker` class.

HTTP Requests and Responses

Every HTTP request consists of a *request line*, a *header* and optionally a *body*. The request line consists of the *method*, the *URL*[3] and the HTTP version number. The two methods relevant for this software are the GET method that requests a web page and the POST method that allows the user to upload data to the web server. The header of a request contains meta information such as the media types that the browser can handle. In case of the POST method the data which is to be uploaded is transmitted in the HTTP body.

[3]Uniform Resource Locator, here the address of a web page

Every HTTP response contains a *status line*, a header and in most cases a body. The status line consists of the HTTP version number and a status code. The header contains meta information such as media type and content length. If the requested page is available it is transferred in the body of the HTTP response.

The full specification of the HTTP protocol can be found in [7].

A web browser connects to the web server for every web page the user requests and for every image and other media embedded in the web page. Therefore, the `WebClientHandler` class has to distinguish between four kinds of requests:

- The start page: It is read from the file system and dumped to the socket with the appropriate status line and header. The start page contains two HTML *forms* into which the user can enter the size of the bulk data to download and the file to upload respectively. The submission of the form by the user will cause the web browser to request the URL specified in the form.

- The bulk data download: In this case the class dumps the HTTP header and a web page that consists of an HTML comment of the requested size directly followed by the results of the diagnostics including references to graphs as well as textual information.

- The bulk data upload: Here, the request contains the file that the user chose to upload in the body of the request. The returned web page contains the results of the diagnostics, consisting of graphs and textual information as well.

- An embedded image: The class reads the graph that has been generated during the measurement and dumps it to the socket preceded by the appropriate status line and header.

The Diagnostics Using Web100

For conducting the measurement the `WebClientHandler` class instantiates the `VarTracker` class once before reading the HTTP request and — if necessary — once before writing the large HTML comment. The values of the measurement during the reading of the request will be discarded if the user has not requested the upload measurement.

A list of the Web100 variables relevant to this application together with a short description is given in figure 3.2.

The round trip time (RTT) and retransmission timeout (RTO) graphs for the bulk data download are simply plots of the variables `SampledRTT` and `CurrentRTO`. The congestion window (`cwnd`) and slow-start threshold (`ssthresh`) diagrams are plots of the variables `CurrentCwnd` and `CurrentSsthresh` where timeouts and Fast Retransmissions are marked with `TO` respectively `FR`.

The time sequence diagram for the bulk data upload is a plot of `RcvNxt` minus the initial sequence number `RecvISS` so that the values start at zero. The instantaneous throughput diagram is generated by plotting the difference of two `ThruBytesReceived` values with a constant time difference.

The textual information is directly assembled as listed in the requirements from the corresponding Web100 variables.

If the minimum of the receiver's advertised window is less than half of its maximum, the application assumes that the user's browser was not able to read the bulk data fast enough and is thus responsible for slowing down the transmission. In that case a warning is printed above the diagrams.

Generation of the Graphs

The graphs are generated with the open source software gnuplot that is very popular among scientists [34].

Gnuplot expects commands from the standard input. These commands load values from a file and set up parameters for formatting the graph and for the output. Therefore, the method `dumpValuesToFile` writes the values that the `VarTracker` class has stored as a list to a file. The `generateGnuplotGraph` method forks[4] and executes the gnuplot binary, passes the necessary commands to its standard input and waits until it exits.

3.4.3 The Web Server Application

The `main` function of the web server applications is simple: It opens a socket and binds it to the port number given as command line argument. Whenever a web client connects the connection is accepted and the corresponding socket descriptor is passed to a new instance of the `WebClientHandler` class.

3.5 Deployment and Future Use

The web server has been installed on a computer at Institut Eurécom that is accessible over the Internet. It can be reached via the URL http://metrojeu2.eurecom.fr.

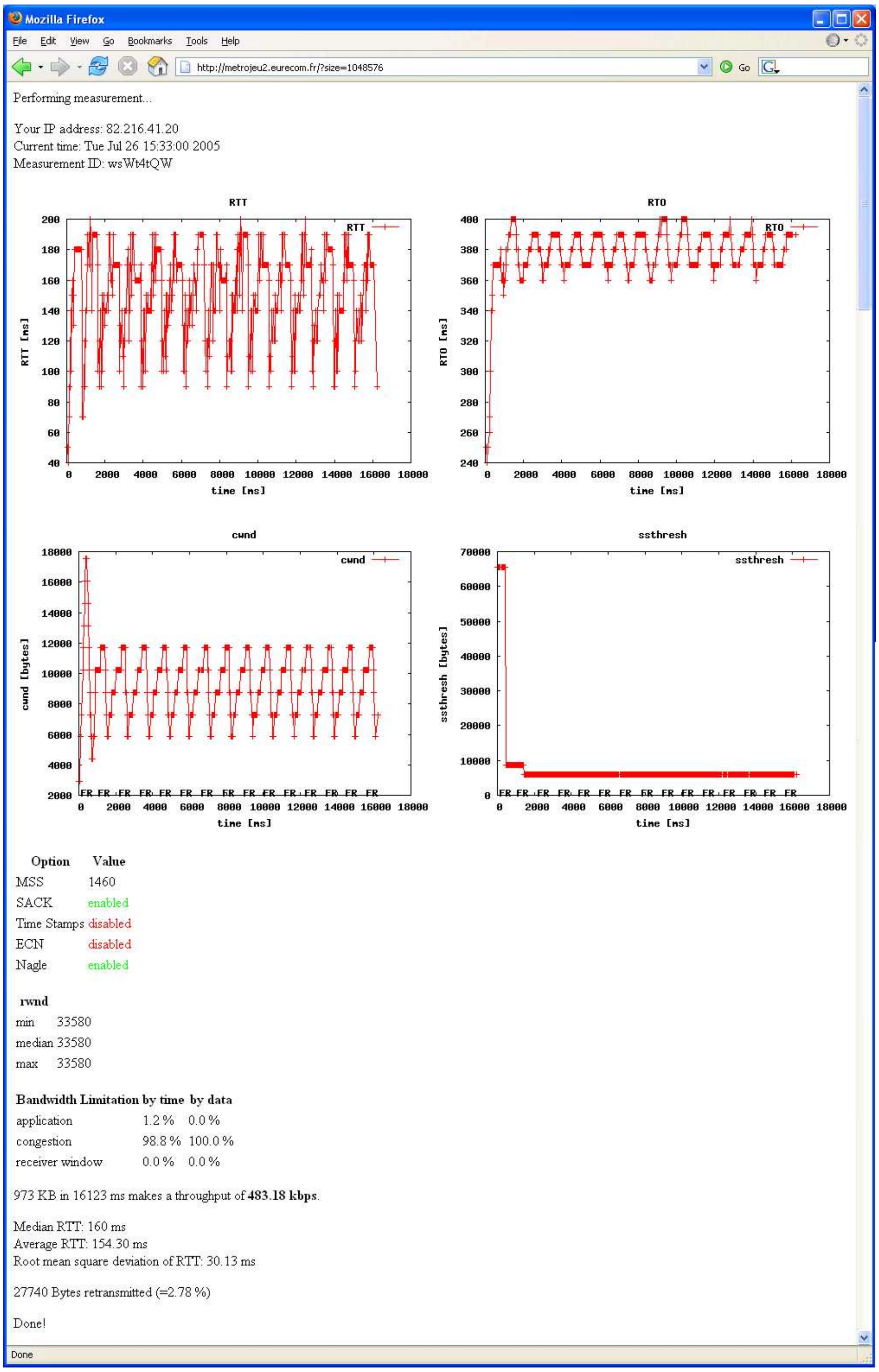
An example screen shot taken on a Windows computer connected via a cable modem (512 kbit/sec) is given in figure 3.3.

The web server will be used in a lecture taught at Institut Eurécom in the future to illustrate TCP's congestion avoidance behavior. Furthermore it will be used in chapter 4 to validate the values deduced by the algorithms presented in chapter 2 that infer the round trip time from recorded traces.

[4]A *fork* is when a process creates a copy of itself, which then acts as a *child* of the original process, now called the *parent*.

variable	**description**
`CurrTime`	The time elapsed between StartTime to the most recent protocol event (packet sent or received).
`PktsIn`	The total number of packets received.
`SampledRTT`	The most recent raw round trip time measurement, in milliseconds, used in calculation of the RTO.
`CurrentRTO`	The current value of the retransmit timer RTO, in milliseconds, not scaled by the RTO backoff multiplier. See RFC 2988 [24].
`CurrentCwnd`	The current congestion window, in bytes.
`CurrentRwinRcvd`	The most recent window advertisement received, in bytes.
`CurrentSsthresh`	The current slow start threshold in bytes.
`Timeouts`	The number of times the retransmit timeout has expired when the RTO backoff multiplier is equal to one. See RFC 2988 [24].
`FastRetran`	The number of invocations of the Fast Retransmit algorithm. See RFC 2581 [4].
`CurrentMSS`	The current maximum segment size (MSS), in bytes.
`SACKEnabled`	True if SACK has been negotiated on, else false. See RFC 2018 [18].
`TimestampsEnabled`	True if timestamps have been negotiated on, else false. See RFC 1323 [11].
`ECNEnabled`	True if ECN has been negotiated on, else false.
`DataBytesIn`	The number of bytes contained in received data segments (without TCP headers), including retransmitted data.
`ThruBytesSent`	The number of bytes for which cumulative acknowledgments have been received.
`BytesRetrans`	The number of bytes retransmitted.
`SndLimTimeSender` `SndLimBytesSender`	The cumulative time (in milliseconds) spent respectively bytes sent in the “Sender Limited” state.
`SndLimTimeCwnd` `SndLimBytesCwnd`	The cumulative time (in milliseconds) spent respectively bytes sent in the “Congestion Limited” state. When there is a retransmission timeout, it should be counted in SndLimTimeCwnd (and not the cumulative time for some other state.)
`SndLimTimeRwin` `SndLimBytesRwin`	The cumulative time (in milliseconds) spent respectively bytes sent in the “Receiver Limited” state.
`RcvNxt`	The value of RCV.NXT from RFC 793 [25]. The next sequence number expected on an incoming segment, and the left or lower edge of the receive window.
`ThruBytesReceived`	The number of bytes for which cumulative acknowledgments have been sent.
`RecvISS`	Initial receive sequence number.

Figure 3.2: Relevant Web100 variables, taken from [1]



Performing measurement...

Your IP address: 82.216.41.20
Current time: Tue Jul 26 15:33:00 2005
Measurement ID: wsWt4tQW

Option	Value
MSS	1460
SACK	enabled
Time Stamps	disabled
ECN	disabled
Nagle	enabled

rwnd	
min	33580
median	33580
max	33580

Bandwidth Limitation	by time	by data
application	1.2 %	0.0 %
congestion	98.8 %	100.0 %
receiver window	0.0 %	0.0 %

973 KB in 16123 ms makes a throughput of **483.18 kbps**.

Median RTT: 160 ms
Average RTT: 154.30 ms
Root mean square deviation of RTT: 30.13 ms

27740 Bytes retransmitted (=2.78 %)

Done!

Figure 3.3: Browser screen shot

Chapter 4

Validation and Assessment

In this chapter we will validate the correctness of the algorithms gathered in chapter 2.3 and show their limitations. We will focus on the algorithms for inferring the round trip time since the exactness of the congestion window algorithm by Jaiswal et al. has already shown to be insufficient for universal use.

4.1 Method

The purpose of most previous work in this field was to infer only one average round trip time per TCP connection and to generate statistics over a large number of connections. With that objective, most authors let their algorithms calculate the RTTs for a large number of sample connections either from "real" Internet traces or from a network simulator and compared them to RTTs calculated by a method they assumed to be correct. Jiang et al. [14] for example use the network delay measured by the *ping* tool right before the start of the connection as well as the result of the SYN-ACK method as reference values. Jaiswal et al. [12] use the average of the RTT values measured by the kernel of the sender.

Since the focus of the software developed in this thesis is to track the round trip time over the lifetime of a TCP connection, we decided to rather analyze a small number of traces in detail and to understand which factors influence the exactness.

Therefore, we accessed the web server described in chapter 3 at Institut Eurécom, France, from four different sites:

- the data center of the German ISP *1&1*[1]
- a 1 Mbit/s DSL link over a wireless router to the German ISP *Arcor*
- a 33.6 kbit/s modem link to the *Leibniz Computing Centre* (LRZ), Germany

[1]a host that is connected with a 100 MBit/s Ethernet interface

- a 512 kbit/s cable link to the French ISP *Numéricable*.

Traces were recorded both at the server and the client side. The web server stored the RTT samples exported by the kernel to a file. For one experiment per site the bulk download from our web server was exclusive. In a second experiment per site the recorded download had to compete with three concurrent FTP downloads for the client side access link, which was the bottle neck in the three cases of dial-up connections.[2]

The client host in the *1&1* data center runs under Linux and supports the TCP Timestamp option by default. The client host for the three connections over ISPs was an *Acer* notebook running Windows XP. To enable support for the TCP Timestamp option a registry key had to be added [20].

4.2 Validation

We ran the Jaiswal (2.3.4) and the timestamp algorithm (2.3.5) on the traces taken at the client side. Our first idea was to use only the Web100 values as reference but since the resolution is only 10 ms, we also ran the algorithm from 2.3.3, that emulates the sender's behavior for measuring the round trip time (recall figure 2.4), on the traces taken at the server side.[3]

The results of the exclusive *1&1* download and the concurrent downloads from the other three sites are given as graphs in the figures 4.2 to 4.5 at the end of this chapter and will be discussed in the following.

4.2.1 Characteristics of the Transfers

The *1&1* transfer (figure 4.2) was mainly limited by the receiver window of 128 KB, and no loss occurred. There was no noticeable queueing involved. Although the tightest link between the client and the server is 100 Mbit/s the TCP connection only reached about 30 Mbit/s throughput, which corresponds to the fraction of receiver window and round trip time:[4]

$$\frac{1024 \text{ kbit}}{0.033 \text{ s}} = 31 \text{ Mbit/s}$$

This receiver window limitation is the reason why the results for the concurrent download was similar: The other downloads were also receiver window limited and did altogether not fully consume the full bandwidth of the tightest link.

[2]From the *1&1* data center we processed only the trace from the exclusive transfer since the concurrent transfer was very similar.

[3]Note, however, that the algorithm does *not* use TCP timestamps unlike the Web100 kernel. Thus during loss recovery the algorithm must interrupt because acknowledgement numbers do not relate to the sequence number of the received data segment whereas Web100 still returns RTT values because it can make the association with help of the TCP timestamps. We will refer to this difference later in this chapter.

[4]The problem of the receiver window limiting TCP throughput is discussed for example in [11].

The traffic of the *Arcor* transfer had to pass a 1 Mbit/s DSL link, which is much tighter than the server's uplink. So we expected that the last router at the ISP would drop packets, which the server would perceive as congestion and deflate the window. However, the queue of the router just before the DSL link is big enough to buffer the packets for all four concurrent downloads. With the filling of the queue the round trip time increases as one can see in figure 4.3. As soon as the receiver window limits the throughput the queue does not fill any more and prevents loss. The round trip time of about 950 ms is thus the sum of network *and* queueing delay.

This behavior is a very strong indication that approaches from previous work that only take one RTT sample at the beginning of a connection — such as the SYN-ACK method — cannot be taken as a representative value for the whole connection because it would underestimate it in cases where the receiver's link is tighter than the sender's link and queueing is involved, which should be the case for most private end users.

In the *Leibniz Computing Centre* trace without concurrent downloads we experienced the same queueing effect as with the *Arcor* trace above. In the experiment with concurrent downloads, however, the queue was overflowed, and loss — perceived as congestion by the sender — finally occurred. This is the more challenging case for the Jaiswal algorithm because the correctness of the round trip time depends on the correct estimation of the congestion window. The increase of RTT in figure 4.4 corresponds again to the queue filling up. The abrupt decrease in RTT is a signal of loss: The sender backs off exponentially and the queue can flush. This observation could be confirmed by the values obtained from Web100.

The router of *Numéricable* dropped packets in both experiments, the one where the transfer was exclusive as well as the one shown in figure 4.5 with concurrent downloads, as we were able to see from the Web100 data exported by the web server.

4.2.2 Evaluation

Figure 4.1 lists the mean, standard deviation and median RTT values for all traces and all applied methods.

Before we will analyze the traces one by one let us address one general observations if figure 4.1. The timestamp algorithm always generates less RTT values than all the other measurement methods. That is because of timestamp ambiguity. Recall that the algorithm takes only those ACKs into account whose TCP timestamp it has not seen before. The Jaiswal algorithm in contrast does not have such a limitation and considers approximately as many three-way associations as there are ACKs in a bulk transfer.

For the transfer to the *1&1* data center (figure 4.2) the Web100 values' resolution of 10 ms is too coarse — the values jump between 30 and 40 ms. But since no loss occurred, we can fully rely on the server side measurement as a reference. The timestamp algorithm returns very precise values that contain almost all peaks from the reference graph. Only the number of values is smaller for the reason mentioned previously. Since the congestion window is equal to

trace	method	mean	std dev	median	# samples
1&1	Web100	33.0	4.77[5]	30	1373
100 Mbit/s	server	32.8	0.407	32.7	3756
	timestamps	32.5	0.334	32.4	1385
	Jaiswal	32.8	0.928	32.7	3699
Arcor	Web100	266	38.7	280	360
DSL	server	251	36.2	260	389
exclusive	timestamps	246	43.5	259	85
	Jaiswal	265	28.4	271	377
Arcor	Web100	966	71.8	960	380
DSL	server	949	63.4	949	409
concurrent	timestamps	944	68.9	942	192
	Jaiswal	974	78.8	955	397
LRZ	Web100	2853	1070	3440	78
modem	server	2991	949	3430	109
exclusive	timestamps	2993	827	3415	78
	Jaiswal	3238	775	3576	96
LRZ	Web100	8099	3104	9260	116
modem	server	7828	2896	8750	92
concurrent	timestamps	8231	2771	9354	81
	Jaiswal	9960	5144	9810	79
Numéricable	Web100	155	31.3	160	501
exclusive	server	132	30.2	140	347
	timestamps	126	27.7	122	121
	Jaiswal	204	31.3	212	294
Numéricable	Web100	168	55.5	160	528
concurrent	server	137	48.0	133	325
	timestamps	139	57.2	121	184
	Jaiswal	223	88.1	212	262

Figure 4.1: Measurement results for all traces. The unit for mean, RMS deviation and median is ms.

the receiver window most of the time, the Jaiswal algorithm performs very well. There is only one erroneous outlier at t=300 due to overestimation of `cwnd`.

Something similar can be observed for the *Arcor* trace. Some of the values that the timestamp algorithm misses due to timestamp ambiguity are unfortunately correct outliers so that the graph looks different than the reference graphs. However, mean and median are only 2.3% respectively 1.9% below the corresponding Web100 reference values.

In the *Leibniz Computing Centre* graphs the superiority of the timestamp algorithm over the Jaiswal algorithm gets obvious. While the graph of the timestamp algorithm is very similar to the two reference graphs, the Jaiswal algorithm systematically overestimates `cwnd`, which leads to a lot of outliers and clearly falsifies mean and median.

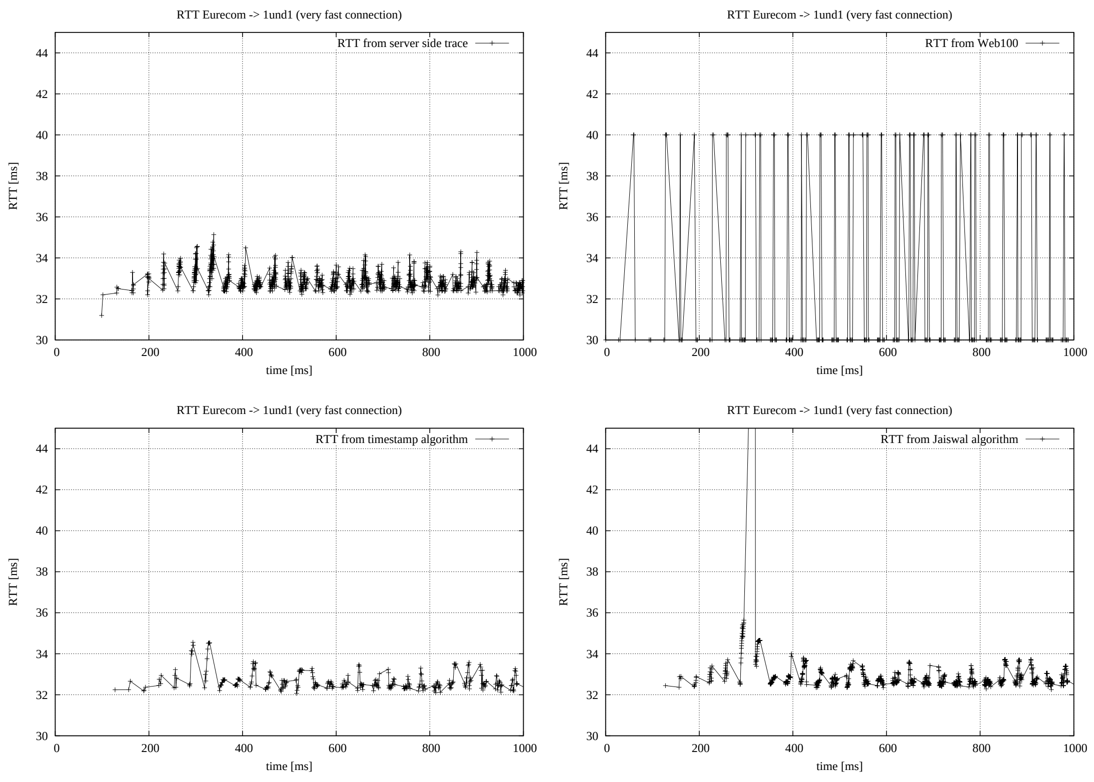


Figure 4.2: Round trip time measurements from *1&1* data center, Germany

The same is true for the *Numéricable* graphs: While after t=10000 the reference values oscillate around 150 ms they oscillate around 200 ms in the Jaiswal graph. Besides, the outliers after t=5000 are obviously incorrect. Consequently the mean and median for the Jaiswal values is clearly too high.

In contrast the timestamp algorithm seems to systematically slightly underestimate the round trip time if compared to the Web100 graph. Since its mean and median are much closer to the corresponding server side values than to the Web100 values, the explanation is that the kernel generates RTT samples even during loss recovery, whereas the other three methods have to interrupt their measurement in that case. One can assume that the ISP's router queue is more filled after a loss than on average and that the additional samples the kernel obtains are consequently higher than the samples outside of recovery available to the three other methods.

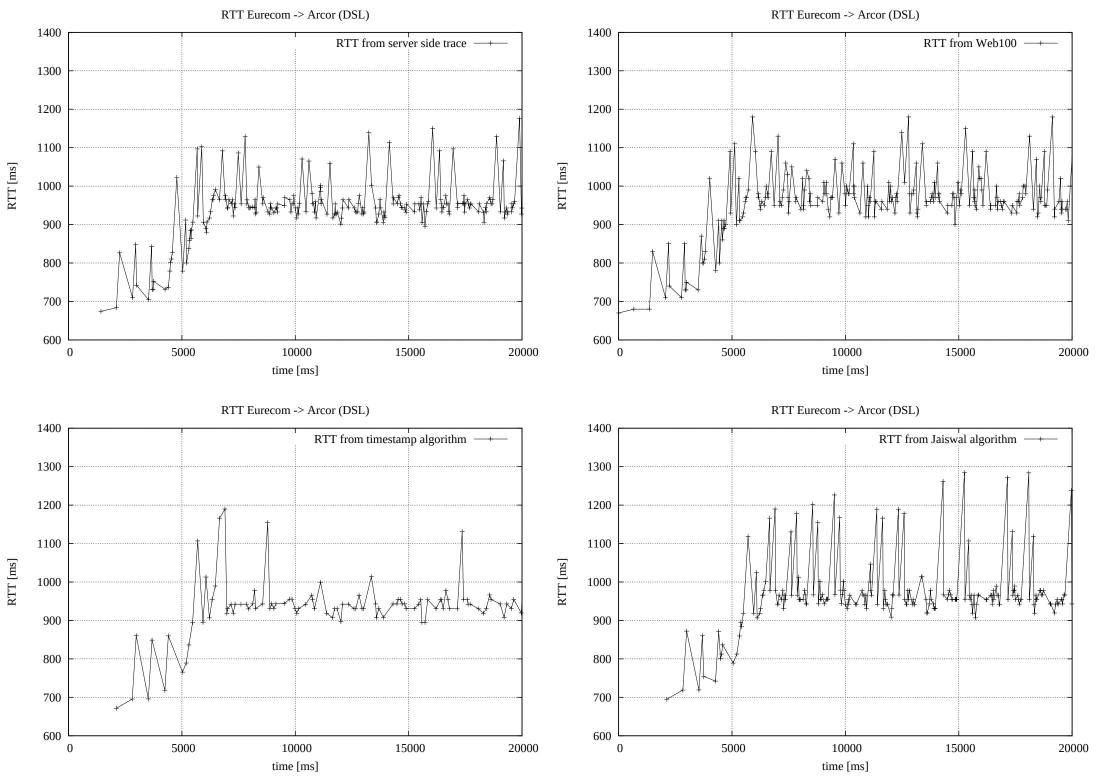


Figure 4.3: Round trip time measurements from DSL connection to the German ISP *Arcor*

4.3 Limitations

In contrast to other authors who have presented their work in this field we clearly state that our experiment is *not* representative for general Internet traffic. Non-application limited bulk transfers are *not* the predominant kind of traffic on the Internet any more.

In network telephony, audio or video streaming or peer-to-peer applications for example rate limitation being imposed by the sending application is prevailing. Our software tries as good as possible not to generate samples at all in that case rather than generating wrong ones. In certain cases like in combination with *delayed acknowledgements,* however, the application limitation is not determinable by a passive approach as we have discussed in section 2.3.4.

4.4 Assessment

The analysis of the example measurement results in this chapter has shown that both the Jaiswal and the timestamp algorithm generate very precise results for receiver window limited transmissions. For congestion limited transfers the timestamp algorithm is clearly superior to the

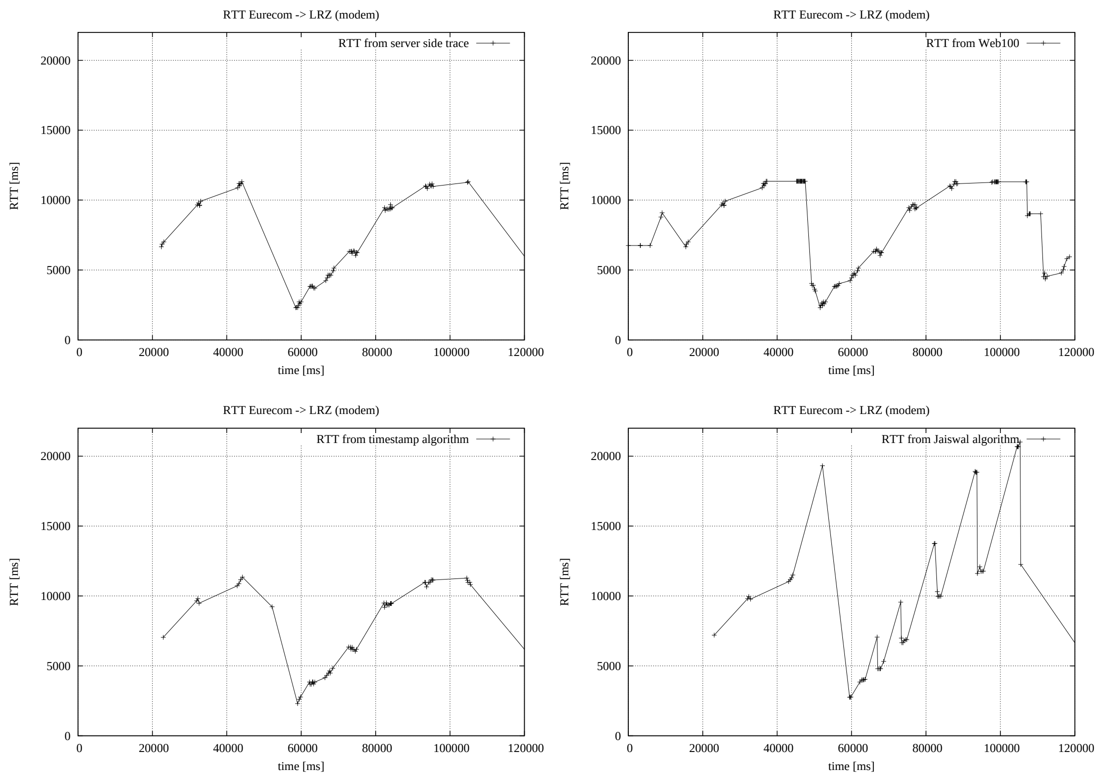


Figure 4.4: Round trip time measurements from modem connection to *Leibniz Computing Centre*, Germany

Jaiswal algorithm since it does not depend on the correct estimation of the congestion window. The Jaiswal algorithm returns more samples since it does not suffer from the timestamp granularity problem. Both have to interrupt their measurements during loss recovery, which makes them less useful for very lossy connections.

A major limitation of the timestamp algorithm is that the necessary TCP option is not entirely deployed on all Internet hosts and that *both* hosts have to support it so that it is used in a connection.

We recommend the use of the software developed in the context of this thesis for analyzing traces where the change in round trip time over the lifetime of the connection — mainly due to queueing effects — matters and where the user is aware of the kind of application that generated the transferred data.

To survey RTT distribution over larger networks or even parts of the Internet on a large scale basis where only network delay and not queueing delay should be taken into account other methods such as *ping* or the SYN-ACK method are more appropriate, though.

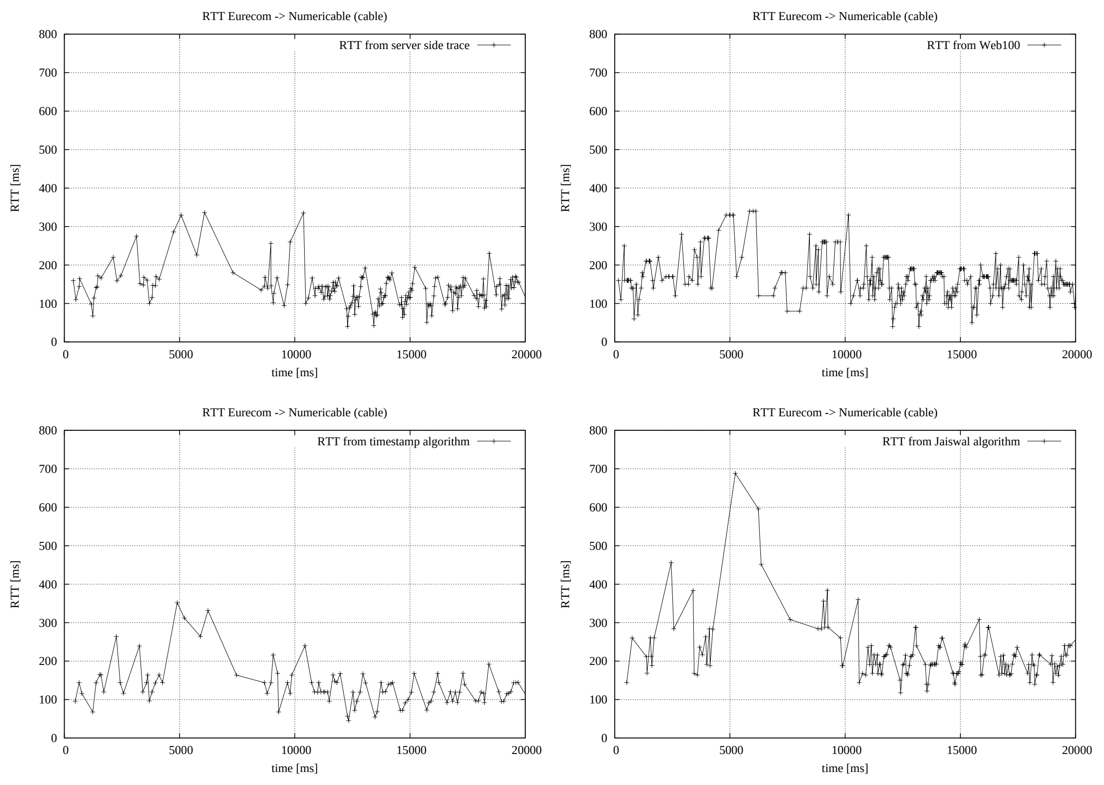


Figure 4.5: Round trip time measurements from cable connection to the French ISP *Numéricable*

Chapter 5

Conclusion and Outlook

In this thesis, we presented several approaches for inferring congestion window and round trip time samples from traces of TCP transmissions and have implemented some of them. The realization was harder than expected, mostly because many publications in this field conceal problems and limitations of their approaches. The resulting software is a combination of multiple algorithms aiming at combining their advantages depending on characteristics of the connection. We have shown that the software produces precise results for bulk data transfers that are not limited by the sending application.

The results are the basis for further analysis of TCP traces conducted at Institut Eurécom such as determining what is the limiting factor of the throughput of a TCP connection.

Furthermore, we have implemented a web server that presents the development of the congestion window, the round trip time and other connection parameters to the user.

Passive techniques for round trip time estimation also pave the way for further studies of today's Internet. Skitter [15] is an example of utilizing active probing and IP address to location mapping to conduct Internet wide delay measurement. Techniques allowing to compute round trip time samples for connections from packet headers anywhere along the path may accomplish this in a non-intrusive, passive way and without prior knowledge about what hosts and sites exist.

Bibliography

[1] "Web100 Kernel Instrumentation Set", http://web100.org/download/kernel/alpha1.2/tcp-kis.txt, 2002.

[2] "The Web100 Project", http://www.web100.org, September 2005.

[3] M. Allman, S. Floyd, and C. Partridge, "RFC 2414: Increasing TCP's Initial Window", http://www.rfc-editor.org/rfc/rfc2414.txt, September 1998.

[4] M. Allman, V. Paxson, and W. R. Stevens, "RFC 2581: TCP Congestion Control", http://www.rfc-editor.org/rfc/rfc2581.txt, April 1999.

[5] R. Braden, "RFC 1122: Requirements for Internet Hosts – Communication Layers", http://www.rfc-editor.org/rfc/rfc1122.txt, October 1989.

[6] D. E. Comer, *Principles, Protocols, and Architecture*, volume 1 of *Internetworking With TCP/IP*, Prentice-Hall, 4th edition, 2000.

[7] R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. J. Leach, and T. Berners-Lee, "RFC 2616: Hypertext Transfer Protocol — HTTP/1.1", http://www.rfc-editor.org/rfc/rfc2616.txt, June 1999.

[8] S. Floyd and T. Henderson, "RFC 2582: The NewReno Modification to TCP's Fast Recovery Algorithm", http://www.rfc-editor.org/rfc/rfc2582.txt, April 1999.

[9] M. Handley, J. Padhye, and S. Floyd, "RFC 2861: TCP Congestion Window Validation", http://www.rfc-editor.org/rfc/rfc2861.txt, April 1999.

[10] V. Jacobson, "Congestion avoidance and control", In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pp. 314–329, New York, NY, USA, 1988, ACM Press.

[11] V. Jacobson, B. Braden, and D. Borman, "RFC 1323: TCP Extensions for High Performance", http://www.rfc-editor.org/rfc/rfc1323.txt, May 1992.

[12] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley, "Inferring TCP Connection Characteristics Through Passive Measurement (extended version)", Technical Report RR03-ATL-070121, Sprint ATL, July 2003.

[13] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley, "Measurement and classification of out-of-sequence packets in a tier-1 IP backbone", In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pp. 113–114, New York, NY, USA, 2002, ACM Press.

[14] H. Jiang and C. Dovrolis, "Passive estimation of TCP round-trip times", *SIGCOMM Comput. Commun. Rev.*, 32(3):75–88, 2002.

[15] A. Ma, "skitter", http://www.caida.org/tools/measurement/skitter, September 2005.

[16] A. Manion, "Vulnerability Note VU#222750", http://www.kb.cert.org/vuls/id/222750, April 2005.

[17] M. Mathis, J. Heffner, and R. Reddy, "Web100: extended TCP instrumentation for research, education and diagnosis", *SIGCOMM Comput. Commun. Rev.*, 33(3):69–79, 2003.

[18] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "RFC 2018: TCP Selective Acknowledgement Option", http://www.rfc-editor.org/rfc/rfc2018.txt, October 1996.

[19] A. Medina, M. Allman, and S. Floyd, "Measuring the evolution of transport protocols in the internet", *SIGCOMM Comput. Commun. Rev.*, 35(2):37–52, 2005.

[20] Microsoft, "Windows 2000 resource kits", http://www.microsoft.com/resources/documentation/Windows/ 2000/server/reskit/en-us/regentry/58800.asp, September 2005.

[21] S. Ostermann, "tcptrace Official Homepage", http://www.tcptrace.org, September 2005.

[22] J. Pahdye and S. Floyd, "On inferring TCP behavior", In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 287–298, New York, NY, USA, 2001, ACM Press.

[23] V. Paxson, "Automated packet trace analysis of TCP implementations", In *SIGCOMM '97: Proceedings of the ACM SIGCOMM '97 conference on Applications, technologies, architectures, and protocols for computer communication*, pp. 167–179, New York, NY, USA, 1997, ACM Press.

[24] V. Paxson and M. Allman, "RFC 2988: Computing TCP's Retransmission Timer", http://www.rfc-editor.org/rfc/rfc2988.txt, November 2000.

[25] J. Postel, "RFC 793: Transmission Control Protocol", http://www.rfc-editor.org/rfc/rfc793.txt, September 1981.

[26] M. Richardson, "TCPDUMP public repository", http://www.tcpdump.org, September 2005.

[27] S. Shakkottai, R. Srikant, N. Brownlee, A. Broido, and K. C. Claffy, "The RTT Distribution of TCP Flows in the Internet and its Impact on TCP-based Flow Control", CAIDA Tech Report Number tr-2004-02, University of Illinois, Cooperative Association for Internet Data Analysis, San Diego Supercomputer Center and University of California, January 2004.

[28] M. Siekkinen, E. W. Biersack, G. Urvoy-Keller, V. Goebel, and T. Plagemann, "InTraBase: Integrated Traffic Analysis Based on a Database Management System", March 2005.

[29] M. Siekkinen, G. Urvoy-Keller, E. Biersack, and T. En-Najjary, "Disambiguating Network Effects from Edge Effects in TCP Connections", To appear in the Proceedings of CoNEXT'2005, October 2005.

[30] W. R. Stevens, *The Protocols*, volume 1 of *TCP/IP illustrated*, Addison-Wesley, 1994.

[31] W. R. Stevens, "RFC 2001: TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms", http://www.rfc-editor.org/rfc/rfc2001.txt, January 1997.

[32] The PostgreSQL Global Development Group, "PostgreSQL 7.4.8 Documentation", http://www.postgresql.org/docs/7.4/static/index.html, September 2005.

[33] B. Veal, K. Li, and D. K. Lowenthal, "New Methods for Passive Estimation of TCP Round-Trip Times.", In *PAM*, pp. 121–134, 2005.

[34] T. Williams and C. Kelley, "gnuplot homepage", http://www.gnuplot.info, September 2005.

[35] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker, "On the characteristics and origins of internet flow rates", In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 309–322, New York, NY, USA, 2002, ACM Press.