Congestion Control in Linux TCP

Pasi Sarolahti
University of Helsinki, Department of Computer Science
pasi.sarolahti@cs.helsinki.fi
Alexey Kuznetsov
Institute of Nuclear Research at Moscow
kuznet@ms2.inr.ac.ru

Abstract

The TCP protocol is used by the majority of the network applications on the Internet. TCP performance is strongly influenced by its congestion control algorithms that limit the amount of transmitted traffic based on the estimated network capacity and utilization. Because the freely available Linux operating system has gained popularity especially in the network servers, its TCP implementation affects many of the network interactions carried out today. We describe the fundamentals of the Linux TCP design, concentrating on the congestion control algorithms. The Linux TCP implementation supports SACK, TCP timestamps, Explicit Congestion Notification, and techniques to undo congestion window adjustments after incorrect congestion notifications.

In addition to features specified by IETF, Linux has implementation details beyond the specifications aimed to further improve its performance. We discuss these, and finally show the performance effects of Quick acknowledgements, Rate-halving, and the algorithms for correcting incorrect congestion window adjustments by comparing the performance of Linux TCP implementing these features to the performance achieved with an implementation that does not use the algorithms in question.

1 Introduction

The *Transmission Control Protocol (TCP)* [Pos81b, Ste95] has evolved for over 20 years, being the most commonly used transport protocol on the Internet today. An important characteristic feature of TCP are its congestion control algorithms, which are essential for preserving network stability when the network load increases. The TCP congestion control principles require that if the TCP sender detects a packet loss, it should reduce its transmission rate, because the packet was probably dropped by a congested router.

Linux is a freely available Unix-like operating system that has gained popularity in the last years. The Linux source code is publicly available[1], which makes Linux an attractive tool for the computer science researchers in various research areas. Therefore, a large number of people have contributed to Linux development during its lifetime. However, many people find it tedious to study the different aspects of the Linux behavior just by reading the source code. Therefore, in this work we describe the design solutions selected in the TCP implementation of the Linux kernel version 2.4. The Linux TCP implementation contains features that differ from the other TCP implementations used today, and we believe that the protocol designers working with TCP find these features interesting considering their work.

The Internet protocols are standardized by the *Internet Engineering Task Force (IETF)* in documents called *Request For Comments (RFC)*. Currently there are thousands of RFCs, of which tens are related to the TCP protocol. In addition to the mandatory TCP specifications, there are a number of experimental and informational specifications of TCP enhancements for improving the performance under certain conditions, which can be im-

[1] The Linux kernel source can be obtained from `http://www.kernel.org/`.

plemented optionally.

Building up a single consistent protocol implementation which conforms to the different RFCs is not a straightforward task. For example, the TCP congestion control specification [APS99] gives a detailed description of the basic congestion control algorithm, making it easier for the implementor to apply it. However, if the TCP implementation supports SACK TCP [MMFR96], it needs to follow congestion control specifications that use a partially different set of concepts and variables than those given in the standard congestion control RFC [FF96, BAF01]. Therefore, strictly following the algorithms used in the specifications makes an implementation unnecessarily complicated, as usually both algorithms need to be included in the TCP implementation at the same time.

In this work we present the approach taken in Linux TCP for implementing the congestion control algorithms. Linux TCP implements many of the RFC specifications in a single congestion control engine, using the common code for supporting both SACK TCP and NewReno TCP without SACK information. In addition, Linux TCP refines many of the specifications in order to improve the TCP efficiency. We describe the Linux-specific protocol enhancements in this paper. Additionally, our goal is to point out the details where Linux TCP behavior differs from the conventional TCP implementations or the RFC specifications.

This paper is organized as follows. In Section 2 we first describe the TCP protocol and its congestion control algorithms in more detail. In Section 3 we introduce the main concepts of the Linux TCP congestion control engine and describe the main algorithms governing the packet retransmission logic. In Section 4 we describe a number of Linux-specific features, for example concerning the retransmission timer calculation. In Section 5 we discuss how Linux TCP conforms to the IETF specifications related to TCP congestion control, and in Section 6 we illustrate the performance effects of selected Linux-specific design solutions. In Section 7 we conclude our work.

2 TCP Basics

We now briefly describe the TCP congestion control algorithms that are referred to throughout this paper. Because the congestion control algorithms play an important role in TCP performance, a number of further enhancements for the TCP algorithms have been suggested. We describe here the ones considered most important. Finally, we point out a few details considered problematic in the current TCP specifications by IETF as a motivation for the Linux TCP approach.

2.1 Congestion control

The TCP protocol basics are specified in RFC 793 [Pos81b]. In order to avoid the network congestion that became a serious problem as the number of network hosts increased dramatically, the basic algorithms for performing congestion control were given by Jacobson [Jac88]. Later, the congestion control algorithms have been included in the standards track TCP specification by the IETF [APS99].

The TCP sender uses a *congestion window (cwnd)* in regulating its transmission rate based on the feedback it gets from the network. The congestion window is the TCP sender's estimate of how much data can be outstanding in the network without packets being lost. After initializing `cwnd` to one or two segments, the TCP sender is allowed to increase the congestion window either according to a *slow start* algorithm, that is, by one segment for each incoming acknowledgement (ACK), or according to *congestion avoidance*, at a rate of one segment in a round-trip time. The *slow start threshold (ssthresh)* is used to determine whether to use slow start or congestion avoidance algorithm. The TCP sender starts with the slow start algorithm and moves to congestion avoidance when `cwnd` reaches the `ssthresh`.

The TCP sender detects packet losses from incoming duplicate acknowledgements, which are generated by the receiver when it receives out-of-order segments. After three successive duplicate ACKs, the sender retransmits a segment and sets `ssthresh` to half of the amount of currently outstanding data. `cwnd` is set to the value of `ssthresh` plus three segments, accounting for the segments that have already left the network according to the arrived duplicate ACKs. In effect the sender halves its transmission rate from what it was before the loss event. This is done because the packet loss is taken as an indication of congestion, and the sender needs to reduce its transmission rate to alleviate the network congestion.

The retransmission due to incoming duplicate ACKs is called *fast retransmit*. After fast retransmit the TCP sender follows the *fast recovery* algorithm until all segments in the last window have been acknowledged. During fast recovery the TCP sender maintains the number of outstanding segments by sending a new segment for each incoming acknowledgement, if the congestion window allows. The TCP congestion control specification temporarily increases the congestion window for each incoming duplicate ACK to allow forward transmission of a segment, and deflates it back to the value at the beginning of the fast recovery when the fast recovery is over.

Two variants of the fast recovery algorithm have been suggested by the IETF. The standard variant exits the fast recovery when the first acknowledgement advancing the window arrives at the sender. However, if there is more than one segment dropped in the same window, the standard fast retransmit does not perform efficiently. Therefore, an alternative called *NewReno* was suggested [FH99] to improve the TCP performance. NewReno TCP exits the fast recovery only after all segments in the last window have been successfully acknowledged.

Retransmissions may also be triggered by the retransmission timer, which expires at the TCP sender when no new data is acknowledged for a while. *Retransmission timeout (RTO)* is taken as a loss indication, and it triggers retransmission of the unacknowledged segments. In addition, when RTO occurs, the sender resets the congestion window to one segment, since the RTO may indicate that the network load has changed dramatically.

The TCP sender estimates packet round-trip times (RTT) and uses the estimator in determining the RTO value. When a segment arrives at the TCP sender, the IETF specifications instruct it to adjust the RTO value as follows [PA00]:

$$\begin{aligned}
&\texttt{RTTVAR <-}\ \tfrac{3}{4} * RTTVAR + \tfrac{1}{4} * |SRTT - R| \\
&\texttt{SRTT <-}\ \tfrac{7}{8} * SRTT + \tfrac{1}{8} * R \\
&\texttt{RTO <-}\ max(SRTT + 4 * RTTVAR, 1s.)
\end{aligned}$$

where `R` is the measured round-trip time, `RTTVAR` is variation of the recent round-trip times, and `SRTT` is the smoothed mean round-trip time based on the recent measurements.

2.2 Enhancements

Recovery from the packet losses is inefficient in the standard TCP because the cumulative acknowledgements allow only one retransmission in a round-trip time. Therefore, *Selective Acknowledgements (SACK)* [MMFR96] were suggested to make it possible for the receiver to acknowledge scattered blocks of incoming data instead of a single cumulative acknowledgement, allowing the TCP sender to make more than one retransmission in a round-trip time. SACK can be used only if both ends of the TCP connection support it.

Availability of the SACK information allows the TCP sender to perform congestion control more accurately. Instead of temporarily adjusting the congestion window, the sender can keep track of the amount of outstanding data and compare it against the congestion window when deciding whether it can transmit new segments [BAF01]. However, the unacknowledged segments can be treated in different ways when accounting for outstanding data. The conservative approach promoted by IETF is to consider all unacknowledged data to be outstanding in the network. The *Forward Acknowledgements (FACK)* algorithm [MM96] takes a more aggressive approach and considers the unacknowledged holes between the SACK blocks as lost packets. Although this approach often results in better TCP performance than the conservative approach, it is overly aggressive if packets have been reordered in the network, because the holes between SACK blocks do not indicate lost packets in this case.

The SACK blocks can also be used for reporting spurious retransmissions. The *Duplicate-SACK (D-SACK)* enhancement [FMMP00] allows the TCP receiver to report any duplicate segments it gets by using the SACK blocks. Having this information the TCP sender can conclude in certain circumstances whether it has unnecessarily reduced its congestion control parameters, and thus revert the parameters to the values preceding the retransmission. For example, packet reordering is a potential reason for unnecessary retransmissions, because out-of-order segments trigger duplicate ACKs at the receiver.

The *TCP Timestamp option* [BBJ92] was suggested to allow more accurate round-trip time measurements, especially on network paths with high bandwidth-delay product. A timestamp is attached to each TCP segment, which is then echoed back in the acknowledgement for the segment. From the echoed timestamp the TCP sender can measure exact round-trip times for the segments and use the measurement for deriving the retransmission timeout estimator. In addition to the more exact round-trip time measurement, use of TCP timestamps allows algorithms for protecting against old segments from the previous incarnations of the TCP connection.

The timestamp option also allows detection of unnecessary retransmissions. The *Eifel Algorithm* [LK00] suggests that if an acknowledgement for a retransmitted segment echoes a timestamp earlier than the timestamp of the retransmission stored at the sender, the original segment has arrived at the receiver, and the retransmission was unnecessarily made. In such a case, the TCP sender can continue by sending new data and revert the recent changes made to the congestion control parameters.

Instead of inferring congestion from the lost packets, *Explicit Congestion Notification (ECN)* [RFB01] was suggested for routers to explicitly mark packets when they arrive to a congested point in the network. When the TCP sender receives an echoed ECN notification from the receiver, it should reduce its transmission rate to mitigate the congestion in the network. ECN allows the TCP senders to be congestion-aware without necessarily suffering from packet losses.

2.3 Criticism

Some details in IETF specifications are problematic in practice. Although many of the RFCs suggest a general algorithm that could be applied to an implementation, combining the algorithms from several RFCs may be inconvenient. For example, combining the congestion control requirements for SACK TCP and NewReno TCP turns out to be problematic due to different variables and algorithms used in the specifications.

The TCP congestion control specifications artificially increase the congestion window during the fast recovery in order to allow forward transmissions to keep the number of outstanding segments stable. Therefore, the congestion window size does not actually reflect the number of segments allowed to be outstanding during the fast recovery. When fast recovery is over, the congestion window is deflated back to a proper size. This procedure is needed because the congestion window is traditionally evaluated against the difference of the highest data segment transmitted (`SND.NXT`) and the first unacknowledged segment (`SND.UNA`). By taking a more flexible method for evaluating the number of outstanding segments, the congestion window size can be constantly maintained at a proper level corresponding to the network capacity.

Adjusting the congestion window consistently becomes an issue especially when SACK information can be used by the TCP sender. By using the selective acknowledgements, the sender can derive the number of packets with a better accuracy than by just using the cumulative acknowledgements. In order to make a coherent implementation of the congestion control algorithms, it is desirable to have common variables and routines both for SACK TCP and for the TCP variant to use when the other end does not support SACK.

Finally, the details of the RTO algorithm presented above have been questioned. Since many networks have round-trip delays of a few tens of milliseconds or less, the RTO algorithm details may not have a significant effect on TCP performance, since the minimum RTO value is limited to one second. However, there are high-delay networks for which the effectiveness of the RTO calculation is important. It has been pointed out that the RTO estimator results in overly large values due to the weight given for the variance of the round-trip time, when the round-trip time suddenly drops for some reason. On the other hand, when the congestion window size increases at a steady pace during the slow start, it is possible that the RTO estimator is not increased fast enough due to small variance in the round-trip times. This may result in spurious retransmission timeouts. Alternative RTO estimators, such as the *Eifel Retransmission Timer* [LS00], have been suggested to overcome the potential problems in the standard RTO algorithm. However, although the Eifel Retransmission Timer is efficient in avoiding the problems of the standard RTO algorithm, it introduces a rather complex set of equations compared to the standard RTO. Therefore, evaluating the possible side effects of different network scenarios on Eifel RTT dynamics is difficult.

3 Linux Approach

Although Linux conforms to the TCP congestion control principles, it takes a different approach in carrying out the congestion control. Instead of comparing the congestion window to the difference of `SND.NXT` and `SND.UNA`, the Linux TCP sender determines the number of packets currently outstanding in the network. The Linux TCP sender then compares the number of outstanding segments to the congestion window when making decisions on how much to transmit. Linux tracks the number of outstanding segments in units of full-sized packets, whereas the TCP specifications and some implementations compare `cwnd` to the number of transmitted octets. This results in different behavior if small segments are used: if the implementation uses a byte-based congestion window, it allows several small segments to be injected in the network for each full-sized segment in the congestion window. Linux, on the other hand, allows only one packet to be transmitted for each segment in the congestion window, regardless of its size. Therefore, Linux is more conservative compared to the byte-based approach when the TCP payload consists of small segments.

The Linux TCP sender uses the same set of concepts and functions for determining the number of outstanding packets with the NewReno recovery and with the two flavors of SACK recovery supported. When the SACK information can be used, the sender can either follow the *Forward Acknowledgements (FACK)* [MM96] approach considering the holes between the SACK blocks as lost segments, or a more conservative approach similar to the ongoing work under IETF [BAF01]. In the latter alternative the unacknowledged segments are considered outstanding in the network. As a basis for all recovery methods the Linux TCP sender uses the equations:

```
left_out <- sacked_out + lost_out
in_flight <- packets_out -
             left_out + retrans_out
```

in defining the number of segments outstanding in the network. In the equation above, `packets_out` is the number of originally transmitted segments above `SND.UNA`, `sacked_out` is the number of segments acknowledged by SACK blocks, `lost_out` is an estimation of the number of segments lost in the network, and `retrans_out` is the number of retransmitted segments. Determining the `lost_out` parameter depends on the selected recovery method. For example, when FACK is in use, all unacknowledged segments between the highest SACK block and the cumulative acknowledgement are counted in `lost_out`. The selected approach makes it easy to add new heuristics for evaluating which segments are lost.

In the absence of SACK information, the Linux TCP sender increases `sacked_out` by one for each incoming duplicate acknowledgement. This is in conformance with the TCP congestion control specification, and the resulting behavior is similar to the *NewReno* algorithm with its forward transmissions. The design chosen in Linux does not require arbitrary adjusting of the congestion window, but `cwnd` holds the valid number of segments allowed to be outstanding in the network throughout the fast recovery.

The counters used for tracking the number of outstanding, acknowledged, lost, or retransmitted packets require additional data structures for supporting them. The Linux sender maintains the state of each outstanding segment in a scoreboard, where it marks the known state of the segment. The segment can be marked as outstanding, acknowledged, retransmitted, or lost. Combinations of these bits are also possible. For example, a segment can be declared lost and retransmitted, in which case the sender is expecting to get an acknowledgement for the retransmission. Using this information the Linux sender knows which segments need to be retransmitted, and how to adjust the counters used for determining `in_flight` when a new acknowledgement arrives. The scoreboard also plays an important role when determining whether a segment has been incorrectly assumed lost, for example due to packet reordering.

The scoreboard markings and the counters used for determining the `in_flight` variable should be in consistent state at all times. Markings for outstanding, acknowledged and retransmitted segments are straightforward to maintain, but when to place a *lost* mark depends on the recovery method used. With NewReno recovery, the first unacknowledged packet is marked lost when the sender enters the fast recovery. In effect, this corresponds to the fast retransmit of the IETF congestion control specifications. Furthermore, when a partial ACK not acknowledging all the data outstanding at the beginning of the fast recovery arrives, the first unacknowledged segment is marked lost. This results in retransmission of the next unacknowledged segment, as the NewReno specification requires.

When SACK is used, more than one segment can be marked lost at a time. With the conservative approach, the TCP sender does not count the holes between the acknowledged blocks in `lost_out`, but when FACK is enabled, the sender marks the holes between the SACK blocks lost as soon as they appear. The `lost_out` counter is adjusted appropriately.

The Linux TCP sender is governed by a state machine that determines the sender actions when acknowledgements arrive. The states are as follows:

- **Open**. This is the normal state in which the TCP sender follows the fast path of execution optimized for the common case in processing incoming acknowledgements. When an acknowledgement arrives, the sender increases the congestion window according to either slow start or congestion avoidance, depending on whether the congestion window is smaller or larger than the slow start threshold, respectively.
- **Disorder**. When the sender detects duplicate ACKs or selective acknowledgements, it moves to the *Disorder* state. In this state the congestion window

is not adjusted, but each incoming packet triggers transmission of a new segment. Therefore, the TCP sender follows the packet conservation principle [Jac88], which states that a new packet is not sent out until an old packet has left the network. In practice the behavior in this state is similar to the *limited transmit* proposal by IETF [ABF01], which was suggested to allow more efficient recovery by using fast retransmit when congestion window is small, or when a large number of segments are lost in the last window of transmission.

- **CWR**. The TCP sender may receive congestion notifications either by Explicit Congestion Notification, *ICMP source quench* [Pos81a], or from a local device. When receiving a congestion notification, the Linux sender does not reduce the congestion window at once, but by one segment for every second incoming ACK until the window size is halved. When the sender is in process of reducing the congestion window size and it does not have outstanding retransmissions, it is in *CWR (Congestion Window Reduced)* state. CWR state can be interrupted by *Recovery* or *Loss* states described below.

- **Recovery**. After a sufficient amount of successive duplicate ACKs arrive at the sender, it retransmits the first unacknowledged segment and enters the *Recovery* state. By default, the threshold for entering *Recovery* is three successive duplicate ACKs, a value recommended by the TCP congestion control specification. During the *Recovery* state, the congestion window size is reduced by one segment for every second incoming acknowledgement, similar to the *CWR* state. The window reduction ends when the congestion window size is equal to ssthresh, i.e. half of the window size when entering the *Recovery* state. The congestion window is not increased during the recovery state, and the sender either retransmits the segments marked lost, or makes forward transmissions on new data according to the packet conservation principle. The sender stays in the *Recovery* state until all of the segments outstanding when the *Recovery* state was entered are successfully acknowledged. After this the sender goes back to the *Open* state. A retransmission timeout can also interrupt the *Recovery* state.

- **Loss**. When an RTO expires, the sender enters the *Loss* state. All outstanding segments are marked lost, and the congestion window is set to one segment, hence the sender starts increasing the congestion window using the slow start algorithm. A major difference between the *Loss* and *Recovery* states is that in the *Loss* state the congestion window is increased after the sender has reset it to one segment, but in the *Recovery* state the congestion window size can only be reduced. The *Loss* state cannot be interrupted by any other state, thus the sender exits to the *Open* state only after all data outstanding when the *Loss* state began have successfully been acknowledged. For example, fast retransmit cannot be triggered during the *Loss* state, which is in conformance with the NewReno specification.

Linux TCP avoids explicit calls to transmit a packet in any of the above mentioned states, for example, regarding the fast retransmit. The current congestion control state determines how the congestion window is adjusted, and whether the sender considers the unacknowledged segments lost. After the TCP sender has processed an incoming acknowledgement according to the state it is in presently, it transmits segments while `in_flight` is smaller than `cwnd`. The sender either retransmits earlier segments marked lost and not yet retransmitted, or new data segments if there are no lost segments waiting for retransmission.

There are occasions where the number of outstanding packets decreases suddenly by several segments. For example, a retransmitted segment and the following forward transmissions can be acknowledged with a single cumulative ACK. These situations would cause bursts of data to be transmitted into the network, unless they are taken into account in the TCP sender implementation. The Linux TCP sender avoids the bursts by limiting the congestion window to allow at most three segments to be transmitted for an incoming ACK. Since burst avoidance may reduce the congestion window size below the slow start threshold, it is possible for the sender to enter slow start after several segments have been acknowledged by a single ACK.

When a TCP connection is established, many of the TCP variables need to be initialized with some fixed values. However, in order to improve the communication efficiency at the beginning of the connection, the Linux TCP sender stores in its destination cache the slow start threshold, the variables used for the RTO estimator, and an estimator measuring the likeliness of reordering after each TCP connection. If another connection is established to the same destination IP address that is found in the cache, the cached values can be used to get adequate initial values for the new TCP connection. If the network conditions between the sender and the receiver change for some reason, the values in the destination cache could get momentarily outdated. However, we consider this a minor disadvantage.

4 Features

We now list selected Linux TCP features that differ from a typical TCP implementation. Linux implements a number of TCP enhancements proposed recently by IETF, such as *Explicit Congestion Notification* [RFB01] and *D-SACK* [FMMP00]. To our knowledge, these features are not yet widely deployed in TCP implementations, but are likely to be in the future because they are promoted by the IETF.

4.1 Retransmission timer calculation

Some TCP implementations use a coarse-grained retransmission timer, having granularities up to 500 ms. The round-trip time samples are often measured once in a round-trip time. In addition, the present retransmission timer specification requires that the RTO timer should not be less than one second. Considering that most of the present networks provide round-trip times of less than 500 ms, studying the feasibility of the traditional retransmission timer algorithm standardized by IETF has not excited much interest.

Linux TCP has a retransmission timer granularity of 10 ms and the sender takes a round-trip time sample for each segment. Therefore it is capable of achieving more accurate estimations for the retransmission timer, if the assumptions in the timer algorithm are correct. Linux TCP deviates from the IETF specification by allowing a minimum limit of 200 ms for the RTO. Because of the finer timer granularity and the smaller minimum limit for the RTO timer, the correctness of the algorithm for determining the RTO is more important than with a coarse-grain timer. The traditional algorithm for retransmission timeout computation has been found to be problematic in some networking environments [LS00]. This is especially true if a fine-grained timer is used and the round-trip time samples are taken for each segment.

In Section 2 we described two problems regarding the standard RTO algorithm. First, when the round-trip time decreases suddenly, RTT variance increases momentarily and causes the RTO value to be overestimated. Second, the RTT variance can decay to a small value when RTT samples are taken for every segment while the window is large. This increases the risk for spurious RTOs that result in unnecessary retransmissions.

The Linux RTO estimator attacks the first problem by giving less weight for the measured mean deviance (`MDEV`) when the measured RTT decreases significantly below the smoothed average. The reduced weight given for the `MDEV` sample is based on the multipliers used in the standard RTO algorithm. First, the `MDEV` sample is weighed by $\frac{1}{8}$, corresponding to the multiplier used for the recent RTT measurement in the `SRTT` equation given in Section 2. Second, `MDEV` is further multiplied by $\frac{1}{4}$ corresponding to the weight of 4 given for the `RTTVAR` in the standard RTO algorithm. Therefore, choosing the weight of $\frac{1}{32}$ for the current `MDEV` neutralizes the effect of the sudden change of the measured RTT on the RTO estimator, and assures that RTO holds a steady value when the measured RTT drops suddenly. This avoids the unwanted peak in the RTO estimator value, while maintaining a conservative behavior. If the round-trip times stay at the reduced level for the next measurements, the RTO estimator starts to decrease slowly to a lower value. In summary, the equation for calculating the `MDEV` is as follows:

```
if (R < SRTT and |SRTT - R| > MDEV) {
      MDEV <- 31/32 * MDEV + 1/32 * |SRTT - R|
} else {
      MDEV <- 3/4 * MDEV + 1/4 * |SRTT - R|
}
```

where `R` is the recent round-trip time measurement, and `SRTT` is the smoothed average round-trip time. Linux does not directly modify the `RTTVAR` variable, but makes the adjustments first on the `MDEV` variable which is used in adjusting the `RTTVAR` which determines the RTO. The `SRTT` and `RTO` estimator variables are set according to the standard specification.

A separate `MDEV` variable is needed, because the Linux TCP sender allows decreasing the `RTTVAR` variable only once in a round-trip time. However, `RTTVAR` is increased immediately when `MDEV` gives a higher estimate, thus `RTTVAR` is the maximum of the `MDEV` estimates during the last round-trip time. The purpose of this solution is to avoid the problem of underestimated RTOs due to low round-trip time variance, which was the second of the problems described earlier.

Linux TCP supports the *TCP Timestamp option* that allows accurate round-trip time measurement also for retransmitted segments, which is not possible without using timestamps. Having a proper algorithm for RTO calculation is even more important with the timestamp option. According to our experiments, the algorithm proposed above gives reasonable RTO estimates also with TCP timestamps, and avoids the pitfalls of the standard algorithm.

The RTO timer is reset every time an acknowledgement advancing the window arrives at the sender. The RTO timer is also reset when the sender enters the *Recovery* state and retransmits the first segment. During the rest of the Recovery state the RTO timer is not reset, but a packet is marked lost, if more than RTO's worth of time has passed from the first transmission of the same segment. This allows more efficient retransmission of packets during the *Recovery* state even though the information from acknowledgements is not sufficient enough to declare the packet lost. However, this method can only be used for segments not yet retransmitted.

4.2 Undoing congestion window adjustments

Because the currently used mechanisms on the Internet do not provide explicit loss information to the TCP sender, it needs to speculate when keeping track of which packets are lost in the network. For example, reordering is often a problem for the TCP sender because it cannot distinguish whether the missing ACKs are caused by a packet loss or by a delayed packet that will arrive later. The Linux TCP sender can, however, detect unnecessary congestion window adjustments afterwards, and do the necessary corrections in the congestion control parameters. For this purpose, when entering the *Recovery* or *Loss* states, the Linux TCP sender stores the old `ssthresh` value prior to adjusting it.

A delayed segment can trigger an unnecessary retransmission, either due to spurious retransmission timeout or due to packet reordering. The Linux TCP sender has mainly two methods for detecting afterwards that it unnecessarily retransmitted the segment. Firstly, the receiver can inform by a *Duplicate-SACK (D-SACK)* that the incoming segment was already received. If all segments retransmitted during the last recovery period are acknowledged by D-SACK, the sender knows that the recovery period was unnecessarily triggered. Secondly, the Linux TCP sender can detect unnecessary retransmissions by using the TCP timestamp option attached to each TCP header. When this option is in use, the TCP receiver echoes the timestamp of the segment that triggered the acknowledgement back to the sender, allowing the TCP sender to conclude whether the ACK was triggered by the original or by the retransmitted segment. The *Eifel* algorithm uses a similar method for detecting spurious retransmissions.

When an unnecessary retransmission is detected by using TCP timestamps, the logic for undoing the congestion window adjustments is simple. If the sender is in the *Loss* state, i.e. it is retransmitting after an RTO which was triggered unnecessarily, the *lost* mark is removed from all segments in the scoreboard, causing the sender to continue with transmitting new data instead of retransmissions. In addition, `cwnd` is set to the maximum of its present value and `ssthresh` * 2, and the `ssthresh` is set to its prior value stored earlier. Since `ssthresh` was set to the half of the number of outstanding segments when the packet loss is detected, the effect is to continue in congestion avoidance at a similar rate as when the *Loss* state was entered.

Unnecessary retransmission can also be detected by the TCP timestamps while the sender is in the *Recovery* state. In this case the Recovery state is finished normally, with the exception that the congestion window is increased to the maximum of its present value and `ssthresh` * 2, and `ssthresh` is set to its prior value. In addition, when a partial ACK for the unnecessary retransmission arrives, the sender does not mark the next unacknowledged segment lost, but continues according to present scoreboard markings, possibly transmitting new data.

In order to use D-SACK for undoing the congestion control parameters, the TCP sender tracks the number of retransmissions that have to be declared unnecessary before reverting the congestion control parameters. When the sender detects a D-SACK block, it reduces the number of revertable outstanding retransmissions by one. If the D-SACK blocks eventually acknowledge every retransmission in the last window as unnecessarily made and the retransmission counter falls to zero due to D-SACKs, the sender increases the congestion window and reverts the last modification to `ssthresh` similarly to what was described above.

While handling the unnecessary retransmissions, the Linux TCP sender maintains a metric measuring the observed reordering in the network in variable `reordering`. This variable is also stored in the destination cache after the connection is finished. `reordering` is updated when the Linux sender detects unnecessary retransmission during the *Recovery* state by TCP timestamps or D-SACK, or when an incoming acknowledgement is for an unacknowledged hole in the sequence number space below selectively acknowledged sequence numbers. In these cases `reordering` is set to the number of segments between the highest segment acknowledged and the currently acknowledged segment, in other words, it corresponds to the maximum distance of reordering in segments detected in the network. Additionally, if FACK was in use when reordering was detected, the sender switches to use the conservative variant of

SACK, which is not too aggressive in a network involving reordering.

4.3 Delayed acknowledgements

The TCP specifications state that the TCP receiver should delay the acknowledgements for a maximum time of 500 ms in order to avoid the *Silly Window Syndrome* [Cla82]. The specifications do not mandate any specific delay time, but many implementations use a static delay of 200 ms for this purpose. However, a fixed delay time may not be adequate in all networking environments with different properties. Thus, the Linux TCP receiver adjusts the timer for delaying acknowledgements dynamically to estimate the doubled packet interarrival time, while sending acknowledgements for at least every second incoming segment. A similar approach was also suggested in an early RFC by Clark [Cla82]. However, the maximum delay for sending an acknowledgement is limited to 200 ms.

Using delayed ACKs slows down the TCP sender, because it increases the congestion window size based on the rate of incoming acknowledgements. In order to speed up the transmission in the beginning of the slow start, the Linux TCP receiver refrains from delaying the acknowledgements for the first incoming segments at the beginning of the connection. This is called *quick acknowledgements*.

The number of quick acknowledgements sent by the Linux TCP receiver is at most half of the number of segments required to reach the receiver's advertised window limit. Therefore, using quick acknowledgements does not open the opportunity for the Silly Window Syndrome to occur. In addition, the Linux receiver monitors whether the traffic appears to be bidirectional, in which case it disables the quick acknowledgements mechanism. This is done to avoid transmitting pure acknowledgements unnecessarily when they can be piggybacked with data segments.

4.4 Congestion Window Validation

The Linux sender reduces the congestion window size if it has not been fully used for one RTO estimate's worth of time. This scheme is similar to the *Congestion Window Validation* suggested by the IETF [HPF00]. The motivation for Congestion Window Validation is that if the congestion window is not fully used, the TCP sender may have an invalid estimate of the present network conditions. Therefore, a network-friendly sender should reduce the congestion window as a precaution.

When the Congestion Window Validation is triggered, the TCP sender decreases the congestion window to half way between the actually used window and the present congestion window. Before doing this, `ssthresh` is set to the maximum of its current value and $\frac{3}{4}$ of the congestion window, as suggested in RFC 2861.

4.5 Explicit Congestion Notification

Linux implements *Explicit Congestion Notification (ECN)* to allow the ECN-capable congested routers to report congestion before dropping packets. A congested router can mark a bit in the IP header, which is then echoed to the TCP sender by an ECN-capable receiver. When the TCP sender gets the congestion signal, it enters the *CWR* state, in which it gradually decreases the congestion window to half of its current size at the rate of one segment for two incoming acknowledgements. Besides making it possible for the TCP sender to avoid some of the congestion losses, ECN is expected to improve the network performance when it is more widely deployed to the Internet routers.

5 Conformance to IETF Specifications

Since Linux combines the features specified in different IETF specifications following certain design principles described earlier, some IETF specifications are not fully implemented according to the algorithms given in the RFCs. Table 1 shows our view of which RFC specifications related to TCP congestion control are implemented in Linux. Some of the features shown in the table can be found in Linux, but they do not fully follow the given specification in all details. These features are marked with an asterisk in the table, and we will explain the differences between Linux and the corresponding RFC in more detail below.

Linux fast recovery does not fully follow the behavior given in RFC 2582. First, the sender adjusts the threshold for triggering fast retransmit dynamically, based on the observed reordering in the network. Therefore, it is possible that the third duplicate ACK does not trigger a fast retransmit in all situations. Second, the Linux sender does not artificially adjust the congestion win-

Table 1: TCP congestion control related IETF specifications implemented in Linux. + = implemented, * = implemented, but details differ from specification.

Specification	Status
RFC 1323 (Perf. Extensions)	+
RFC 2018 (SACK)	+
RFC 2140 (Ctrl block sharing)	+
RFC 2581 (Congestion control)	*
RFC 2582 (NewReno)	*
RFC 2861 (Cwnd validation)	+
RFC 2883 (D-SACK)	+
RFC 2988 (RTO)	*
RFC 3042 (Lim. xmit)	+
RFC 3168 (ECN)	*

dow during fast recovery, but maintains its size while adjusting the `in_flight` estimator based on incoming acknowledgements. The different approach alone would not cause significant effect on TCP performance, but when entering the fast recovery, the Linux sender does not reduce the congestion window size at once, as RFC 2582 suggests. Instead, the sender decreases the congestion window size gradually, by one segment per two incoming acknowledgements, until the congestion window meets half of its original value. This approach was originally suggested by Hoe [Hoe95], and later it was named *Rate-halving* according to an expired Internet Draft by Mathis, et. al. Rate-halving avoids pauses in transmission, but is slightly too aggressive after the congestion notification, until the congestion window has reached a proper size.

As described in Section 4, the round-trip time estimator and RTO calculation in Linux differs from the Proposed Standard specification by the IETF. Linux follows the basic patterns given in RFC 2988, but the implementation differs from the specification in adjusting the `RTTVAR`. A significant difference between RFC 2988 and Linux implementation is that Linux uses the minimum RTO limit of 200 ms instead of 1000 ms given in RFC 2988.

RFC 2018 defines the format and basic usage of the SACK blocks, but does not give detailed specification of the congestion control algorithm that should be used with SACK. Therefore, applying the FACK congestion control algorithm, as Linux does by default, does not violate the current IETF specifications. However, since FACK results in overly aggressive behavior when packets have been reordered in the network, the Linux sender changes from FACK to a more conservative congestion control algorithm when it detects reordering. The IETF currently has a work in progress draft defining a congestion control algorithm to be used with SACK [BAF01], which is similar to the conservative SACK alternative in Linux. Furthermore, Linux follows the D-SACK basics given in RFC 2883.

Linux implements RFC 1323, which defines the TCP timestamp and window scaling options, and the limited transmit enhancement defined in RFC 3042, which is taken care of by the *Disorder* state of the Linux TCP state machine. However, if the `reordering` estimator has been increased from the default of three segments, the Linux TCP sender transmits a new segment for each incoming acknowledgement, not only for the two first ACKs. Finally, the Linux destination cache provides functionality similar to the RFC 2140 that proposes Control Block Interdependence between the TCP connections.

6 Performance Issues

Before concluding our work, we illustrate the performance implications of using quick acknowledgements, rate-halving, and congestion window reverting. We do this by disabling these features, and comparing the time-sequence diagrams of a pure Linux TCP implementation and an implementation with the corresponding feature disabled. We use Linux hosts as connection endpoints communicating over a 256 Kbps link with MTU of 1500 bytes. Between the sender and the 256 Kbps link there is a tail-drop router with buffer space for seven packets, connected to the sender with a high-bandwidth link with small latency. The test setup is illustrated in Figure 1. In addition to the low bandwidth, the link between the router and TCP receiver has a fairly high propagation delay of 200 ms. The slow link and the router are emulated using a real-time network emulator [KGM+01]. With the network emulator we can control the link and the network parameters and collect statistics and log about the network behavior to help the analysis.

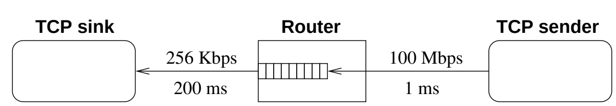
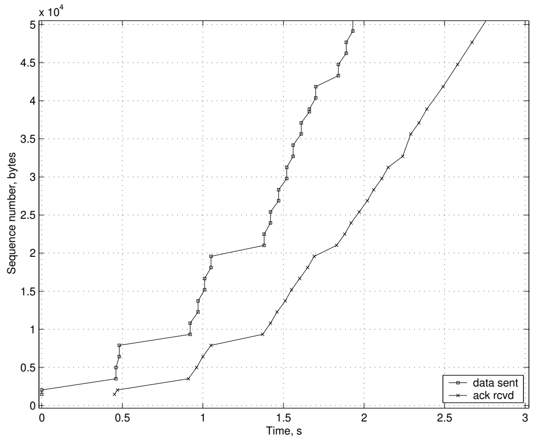
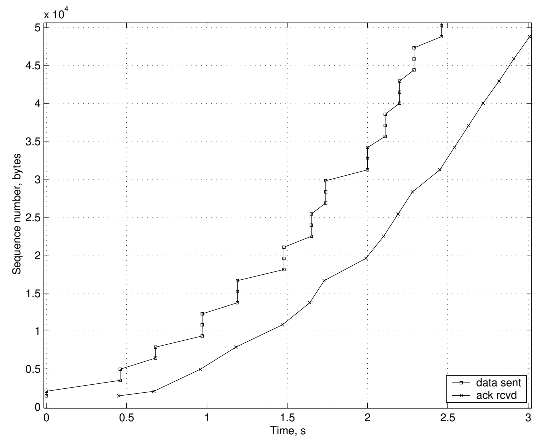


Figure 1: Test setup.

We first illustrate the effect of quick acknowledgements on TCP throughput. Figure 2(a) shows the slow start performance of unmodified Linux implementing quick



(a) Quick acknowledgements enabled.



(b) Quick acknowledgements disabled.

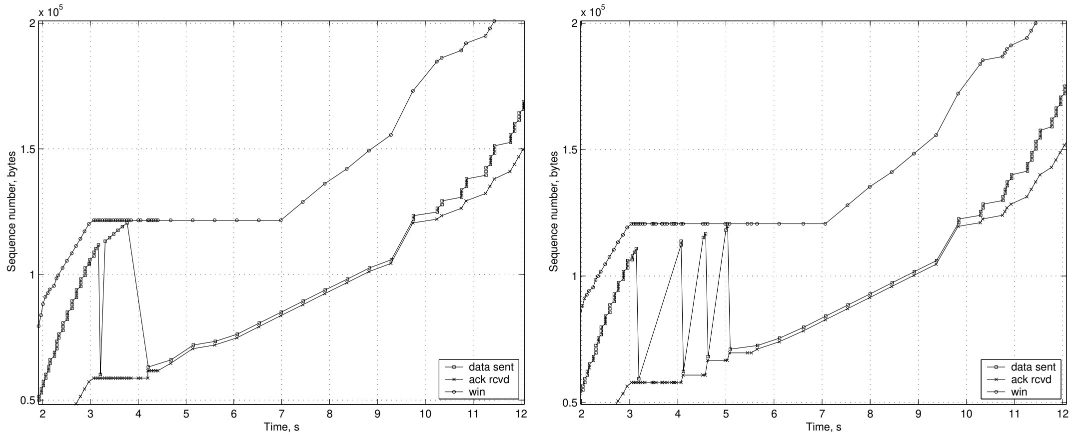
Figure 2: Effect of quick acknowledgements on slow start performance.

acknowledgements, and Figure 2(b) shows the performance of an implementation with the quick acknowledgements mechanism disabled. The latter implementation applies a static delay of 200 ms for every acknowledgement, but transmits an acknowledgement immediately if more than one full-sized segment's worth of unacknowledged data has arrived. One can see that when the link has a high bandwidth-delay product, such as our link does, the benefit of quick acknowledgements is noticeable. The unmodified Linux sender has transmitted 50 KB in 2 seconds, but when the quick acknowledgments are disabled, it takes 2.5 seconds for the sender to transmit 50 KB. In our example, the unmodified Linux receiver with quick acknowledgements enabled sent 109 ACK packets, and the implementation without quick acknowledgements sent 95 ACK packets. Because quick acknowledgements cause more ACKs to be generated in the network than when using the conventional delayed ACKs, the sender's congestion window increases slightly faster. Although this improves the TCP performance, it makes the network slightly more prone to congestion.

Rate-halving is expected to result in a similar average transmission rate as the conventional TCP fast recovery, but it paces the transmission of segments smoothly by making the TCP sender reduce its congestion window steadily instead of making a sudden adjustment. Figure 3(a) illustrates the performance of an unmodified Linux TCP implementing rate-halving, and Figure 3(b) illustrates the performance of an implementation with the conventional fast recovery behavior. These figures also illustrate the receiver's advertised window (the uppermost line), since it limits the fast recovery in our example.

The scenario is the same in both figures: the router buffer is filled up, and several packets are dropped due to congestion before the feedback of the first packet loss arrives at the sender. The packet losses at the bottleneck link due to initial slow start is called slow start overshooting. The figures show that after 12 seconds both TCP variants have transmitted 160 KB. However, the behavior of the unmodified Linux TCP is different from the TCP with rate-halving disabled. With the conventional fast recovery, the TCP sender stops sending new data until the number of outstanding segments has dropped to half of the original amount, but the sender with the rate-halving algorithm lets the number of outstanding segments reduce steadily, with the rate of one segment for two incoming acknowledgements. Both variants suffer from the advertised window limitation, which does not allow the sender to transmit new data, even though the congestion window would.

Finally, we show how the timestamp-based undoing affects TCP performance. We generated a three-second delay, which is long enough to trigger a retransmission timeout at the TCP sender. Figure 4(a) shows a TCP implementation with the TCP timestamp option enabled, and Figure 4(b) shows the same scenario with timestamps disabled. The acknowledgements arrive at the sender in a burst, because during the delay packets queue up in the emulated link receive buffers and are all re-



(a) Rate-halving enabled.

(b) Rate-halving disabled.

Figure 3: Effect of Rate-halving on TCP performance.

leased when the delay is over[2].

The timestamps improve the TCP performance considerably, because the TCP sender detects that the acknowledgement following the retransmission was for the original transmission of the segment. Therefore the sender can revert the `ssthresh` to its previous value and increase congestion window. Moreover, the Linux TCP sender avoids unnecessary retransmissions of the segments in the last window. The ACK burst injected by the receiver after the delay causes 19 new segments to be transmitted by the sender within a short time interval. However, the sender follows the slow start correctly as clocked by the incoming acknowledgements, and none of the segments are transmitted unnecessarily.

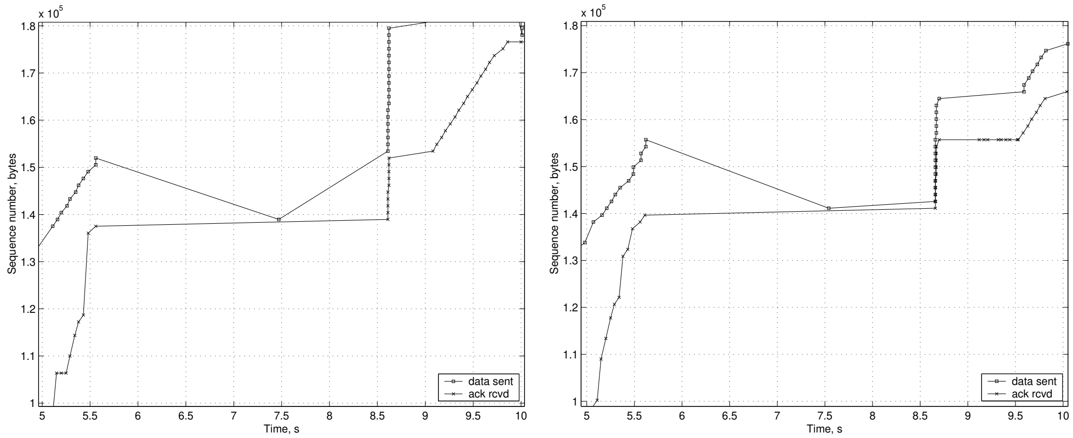
A conventional TCP sender not implementing congestion window reverting retransmits the last window after the first delayed segment unnecessarily. Not only does this waste the available bandwidth, but the retransmitted segments appearing as out-of-order data at the receiver trigger several duplicate acknowledgments. However, since the TCP sender is still in the *Loss* state, the duplicate ACKs do not cause further retransmissions. One can see that, the conventional TCP sender without timestamps has received acknowledgements for 165 KB of data in the 10 seconds after the transmission begun, while the Linux sender implementing TCP timestamps and congestion window reverting has received acknowledgements for 175 KB of data. The Linux TCP sender having TCP timestamps enabled retransmitted 22.6 KB in 16 packets, but the Linux TCP sender without timestamps retransmitted 37.1 KB in 26 packets in the test case transmitting 200 KB. The link scenario was the same in both test runs, having a 3-second delay in the middle of transmission. When TCP timestamps were not used, the TCP sender retransmitted 11 packets unnecessarily.

[2] The delay stands for emulated events on the link layer, for example representing persistent retransmissions of erroneous link layer frames. The link receive buffer holds the successfully received packets until the period of retransmissions is over to be able to deliver them in order for the receiver.

7 Conclusion

We presented the basic ideas of the Linux TCP implementation, and gave a description of the details that differ from a typical TCP implementation. Linux implements many of the recent TCP enhancements suggested by the IETF, some of which are still at a draft state. Therefore Linux provides a platform for testing the interoperability of the recent enhancements in an actual network. The current design also makes it easy to implement and study alternative congestion control policies.

The Linux TCP behavior is strongly governed by the packet conservation principle and the sender's estimate of which packets are still in the network, which are acknowledged, and which are declared lost. Whether to retransmit or transmit new data depends on the markings made in the TCP scoreboard. In most of the cases none of the requirements given by the IETF are violated, al-



(a) TCP timestamps enabled.

(b) TCP timestamps disabled.

Figure 4: Effect of congestion window undoing on TCP performance.

though in marginal scenarios the detailed behavior may be different from what is given in the IETF specifications. However, the TCP essentials, in particular the congestion control principles and conservation of packets, are maintained in all cases.

The selected approach can also be problematic when implementing some features. Because Linux combines the features in different IETF specifications under the same congestion control engine, an uncareful implementation may break some parts of the retransmission logic. For example, if the balance between congestion window and `in_flight` variable is broken, fast recovery algorithm may not work correctly in all situations.

Acknowledgements

We would like to thank Markku Kojo and Craig Metz for the useful feedback given during preparation of this paper.

References

[ABF01] M. Allman, H. Balakrishnan, and S. Floyd. Enhancing TCP's Loss Recovery Using Limited Transmit. RFC 3042, January 2001.

[APS99] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581, April 1999.

[BAF01] E. Blanton, M. Allman, and K. Fall. A Conservative SACK-based Loss Recovery Algorithm for TCP. Internet draft "draft-allman-tcp-sack-08.txt", November 2001. Work in progress.

[BBJ92] D. Borman, R. Braden, and V. Jacobson. TCP Extensions for High Performance. RFC 1323, May 1992.

[Cla82] D. D. Clark. Window and Acknowledgement Strategy in TCP. RFC 813, July 1982.

[FF96] K. Fall and S. Floyd. Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. *ACM Computer Communication Review*, 26(3), July 1996.

[FH99] S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 2582, April 1999.

[FMMP00] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An Extension to the Selective Acknowledgment (SACK) Option for TCP. RFC 2883, July 2000.

[Hoe95] J. Hoe. Start-up Dynamics of TCP's Congestion Control and Avoidance Schemes. Master's thesis, Massachusetts Institute of Technology, June 1995.

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

[Jac88] V. Jacobson. Congestion Avoidance and Control. In *Proceedings of ACM SIGCOMM '88*, pages 314–329, August 1988.

[KGM+01] M. Kojo, A. Gurtov, J. Manner, P. Sarolahti, T. Alanko, and K. Raatikainen. Seawind: a Wireless Network Emulator. In *Proceedings of 11th GI/ITG Conference on Measuring, Modelling and Evaluation of Computer and Communication Systems*, pages 151–166, Aachen, Germany, September 2001. VDE Verlag.

[LK00] R. Ludwig and R. H. Katz. The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions. *ACM Computer Communication Review*, 30(1), January 2000.

[LS00] R. Ludwig and K. Sklower. The Eifel Retransmission Timer. *ACM Computer Communication Review*, 30(3), July 2000.

[MM96] M. Mathis and J. Mahdavi. Forward acknowledgement: Refining TCP Congestion Control. In *Proceedings of ACM SIGCOMM '96*, October 1996.

[MMFR96] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgement Options. RFC 2018, October 1996.

[PA00] V. Paxson and M. Allman. Computing TCP's Retransmission Timer. RFC 2988, November 2000.

[Pos81a] J. Postel. Internet Control Message Protocol. RFC 792, September 1981.

[Pos81b] J. Postel. Transmission Control Protocol. RFC 793, September 1981.

[RFB01] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, September 2001.

[Ste95] W. Stevens. *TCP/IP Illustrated, Volume 1; The Protocols*. Addison Wesley, 1995.