Performance Problems in BSD4.4 TCP

Lawrence S. Brakmo Larry L. Peterson

{brakmo,llp}@cs.arizona.edu
Department of Computer Science
University of Arizona
Tucson, AZ 85721

Abstract

This paper describes problems in the BSD 4.4-Lite version of TCP (some of which are also present in earlier versions, such as the Net2 implementation of TCP) and proposes fixes that result in a 21% increase in throughput under realistic conditions.

1 Introduction

As part of our work with TCP Vegas [1], we ported the BSD 4.4-Lite version of TCP (TCP Lite) to the x-kernel [3] with the goal of comparing its performance to that of our implementation of TCP Reno and TCP Vegas.[1] Early results from our simulations showed that TCP Lite performed significantly worse than our version of TCP Reno which was used to measure the gains resulting from using TCP Vegas. In one simple simulation scenario—when no other traffic is present—TCP Reno sends one MByte of data in 8.2 seconds (123 KB/s) and TCP Vegas sends one MByte in 6.2 seconds (166 KB/s), while TCP Lite takes 14.6 seconds (70 KB/s). TCP Lite's performance degradation is also seen when the data transfers are sharing the bottleneck router with background traffic.

We analyzed TCP Lite to find the reasons for its reduced throughput. The first step of the analysis involved examining the graphical and textual traces of the simulations (described in the next section), which allowed us to find the problem areas quickly. This was followed by a detailed inspection of the code to find the exact problems and to ascertain that the problems were not created during the porting of the code to the x-kernel. Finally, the code was modified to fix the exposed problems, and tested to gauge the effect of the fixes. By fixing the problems found in TCP Lite, we were able to increase its throughput by up to 21% under realistic conditions.

Some problems discovered during the analysis of transfers with no background traffic turned out to have very little effect on the tests with background traffic. We describe these problems because they may affect throughput under other scenarios. We also played with some parameters, such as ACKing frequency, to see the effect on both throughput and losses.

[1] Our TCP Lite implementation is based on a version retrieved from ftp.cdrom.com, dated 4/10/94. Our TCP Reno implementation is based on the Reno distribution of BSD Unix, and contains the Fast Recovery and Fast Retransmit mechanisms described in [5].

This paper is organized as follows: Section 2 outlines the tools we used to measure and analyze TCP. Section 3 then describes the problems found in TCP Lite and outlines the fixes, and Section 4 compares the performance of the original and improved versions of TCP Lite. Finally, Section 5 makes some concluding remarks.

2 Tools

This section briefly describes the tools used to implement and analyze the different versions of TCP. Reading this section is not required for understanding the rest of the paper, but it will allow the reader to interpret the graphs used to deduce the problems in TCP Lite. All of the protocols were developed and tested under the University of Arizona's x-kernel framework. Our implementations of TCP Reno and TCP Lite were derived by retrofitting the BSD implementations into the x-kernel.

2.1 Simulator

The results reported in this paper were obtained from a network simulator based on the x-kernel. In this environment, actual x-kernel protocol implementations run on a simulated network. Specifically, the simulator supports multiple hosts, each running a full protocol stack (TEST/TCP/IP/ETH), and several abstract link behaviors (point-to-point connections and ethernets). Routers can be modeled either as a network node running the actual IP protocol code, or as an abstract entity that supports a particular queuing discipline (e.g., FIFO). All the simulations reported in this paper simulate FIFO-based (tail drop) routers.

One of the most important protocols available in the simulator is a protocol called TRAFFIC—it generates TCP Internet traffic based on *tcplib* [2]. TRAFFIC starts conversations with interarrival times given by an exponential distribution. Each conversation can be of type TELNET, FTP, NNTP, or SMTP, each of which expects a set of parameters. For example, FTP expects the following parameters: number of items to transmit, control segment size, and the item sizes. All of these parameters are based on probability distributions obtained from traffic traces. Finally, each of these conversations runs on top of its own TCP connection.

2.2 Trace Facility

We have added code to the x-kernel and its protocols to trace the relevant changes in the connection state. We then developed various tools to analyze and display the tracing information. One of the tools provides excruciating detail of the trace information in textual form. Due to the detailed information shown by this tool, its usefulness at finding problems is limited unless there is some knowledge of the problem, such as when it occurs. In other words, if all we are given is the output of this tool, the chance that we will find any problems in the behavior of the protocol that created the traces is small, as any strange or uncommon behavior is lost under the pages of detailed information. However, once we know that something looks strange t seconds into an experiment, then by looking into this tool's output we can find out if there is really a problem, and if so, what the problem is.

There is another tool whose output is very useful at finding possible problems with a given protocol, and for developing intuition and a deeper understanding of protocol behavior. The rest of this section describes this tool, which graphically represents relevant features of the state of the TCP connection as a function of time. This tool outputs multiple graphs, each focusing on a specific set of characteristics of the connection state. Figure 3 gives an example. Since we use graphs like this throughout the paper, we now explain how to read the graph in some detail.

First, all TCP trace graphs have certain features in common, as illustrated in Figure 1. The circled numbers in this figure are keyed to the following explanations:

1. Hash marks on the x-axis indicate when an ACK was received.
2. Hash marks at the top of the graph indicate when a segment was sent.
3. The numbers on the top of the graph indicate when the n^{th} kilobyte (KB) was sent.
4. Diamonds on top of the graph indicate when the periodic coarse-grained timer fires. This does not imply a TCP timeout, just that TCP checked to see if any timeouts should happen.
5. Circles on top of the graph indicate that a coarse-grained timeout occurred, causing a segment to be retransmitted.
6. Solid vertical lines running the whole height of the graph indicate when a segment that is eventually retransmitted was originally sent, presumably because it was lost. Notice that several consecutive segments are retransmitted in the example.

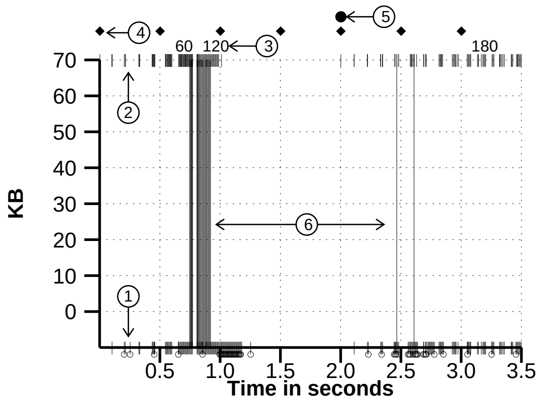

Figure 1: Common Elements in TCP Trace Graphs.

In addition to this common information, each graph depicts more specific information. The most complex of these gives the size of the different windows TCP uses for flow and congestion control. Figure 2 shows these in more detail, again keyed by the following explanations:

1. The dashed line gives the threshold window. It is used during slow-start, and marks the point at which the congestion window growth changes from exponential to linear.
2. The dark gray line gives the send window. It is the minimum of the sender's buffer size and receiver's advertized window, and defines an upper limit to the number of bytes sent but not yet acknowledged.
3. The light gray line gives the congestion window. It is used for congestion control, and is also an upper limit to the number of bytes sent but not yet acknowledged.
4. The thin line gives the actual number of bytes in transit at any given time, where by in transit we mean sent but not yet acknowledged. In the text we use the term UNACK-COUNT to refer to this line or

its value.

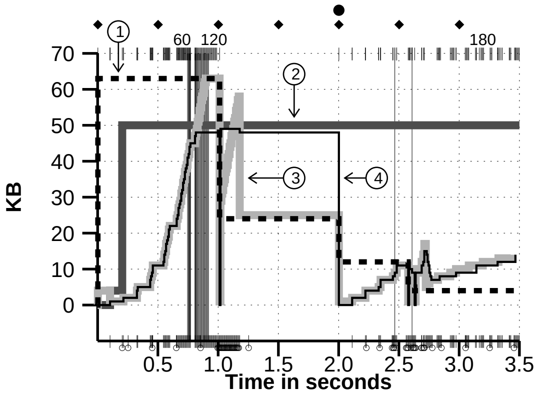

Figure 2: TCP Windows Graph.

Since the window graph presents a lot of information, it is easy to get lost in the detail. To assist the reader in developing a better understanding of this graph, the Appendix presents a detailed description of the behavior depicted in Figure 2.

3 Analysis of TCP Lite

This section describes and analyzes problems found in TCP Lite,[2] as well as suggests fixes for these problems. The effect these problems have on throughput depends on the particular details of the connection, such as RTT, available bandwidth, and so on. For example, these problems may have no effect on a connection over a local area network, yet have a dramatic impact on an Internet-wide connection.

The main simulation configuration consisted of two Ethernet LANs connected by two gateways through a 200 KB/s line with a 50ms delay. Except for the TCP connections that are part of the simulated background traffic, we set the send buffer in each TCP connection to 50KB. Using a size that is too small would have limited the transfer rate. Note that even though there is an optimal buffer size which maximizes the throughput (by minimizing the losses), it is not fair to use it since it is a function of the available bandwidth, which is not known under normal circumstances.

Figure 3 shows the trace graph of an isolated TCP Lite connection transferring one Mbyte of data, and Figure 4 shows the trace graph of a TCP Reno connection under the same conditions. In both figures, the topmost graph gives the window information, as described in Section 2; the middle graph shows the average sending rate, calculated from the last 12 segments; and the bottom graph shows the average queue length at the bottleneck router.

In these tests, the TCP Lite connection performs more than 70% worst than the TCP Reno connection. A detailed analysis was carried out to find the cause of TCP Lite's performance problems, and to ascertain that they were not introduced during the porting of the BSD code to the x-kernel. The performance problems

[2] Some of the problems in TCP Lite also appear in earlier BSD versions of TCP such as the Net2 version of TCP

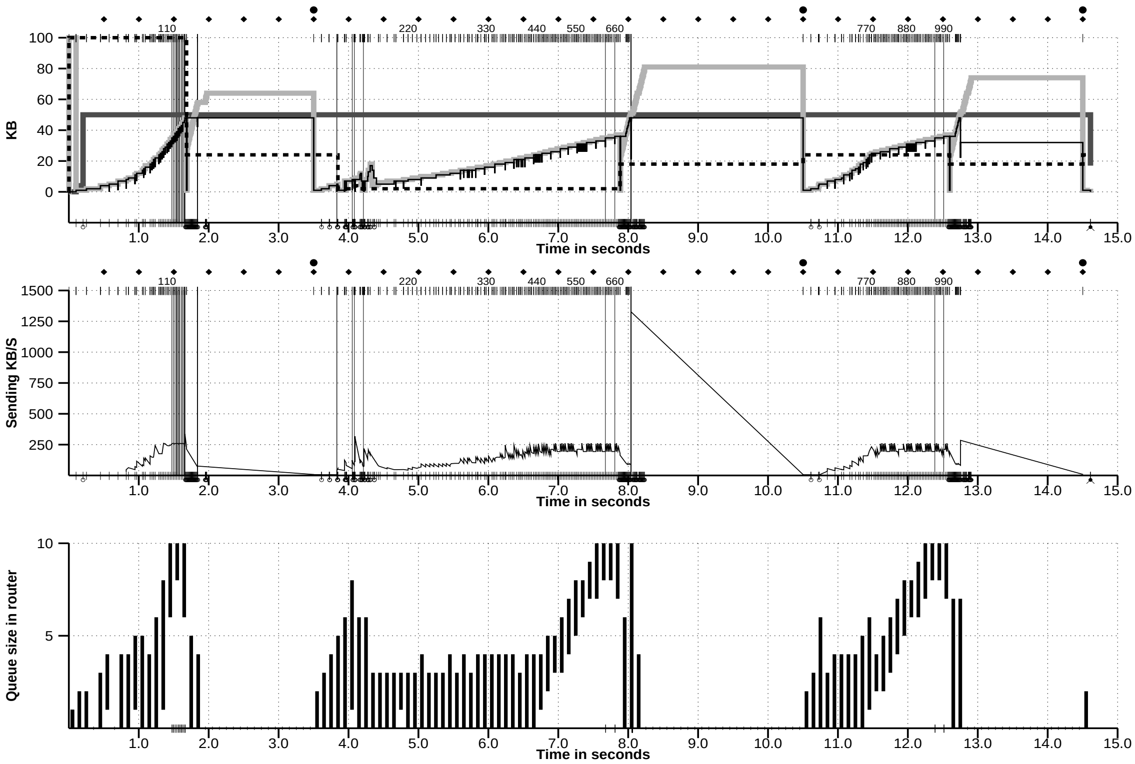

Figure 3: TCP Lite with No Other Traffic (Throughput: 70 KB/s).

found in TCP Lite also manifest themselves under other simulation configurations. For example, when we have a distinguished TCP connection sharing the network with traffic generated from *tcplib*, the distinguished connection does 14% worst when it is running TCP Lite than when it is running TCP Reno.

The format of the analysis will be to first deduce the problem from the graphical traces, then to further analyze it through the textual traces and code inspection. For each problem area, we also include the fix, and the effect of the cumulative fixes on TCP Lite's performance.

3.1 Error in Header Prediction Code

One of the first things to notice in the trace graphs of TCP Lite is the high number of retransmit timeouts. These timeouts are represented by the black circles at the top of the first graph in Figure 3 and occur at 3.5, 10.5 and 14.5 seconds. The Fast Retransmit and Fast Recovery mechanisms seem unable to do their job, which is to prevent retransmit timeouts and to keep the pipe full. If we look at the trace graphs of TCP Reno in Figure 4, we see that after the losses associated with the initial slow-start, Reno recovers from losses without a retransmit timeout, unlike TCP Lite.

The source of the problem can be observed by looking at the top graph of Figure 3 at around 8 seconds. Right before the 8 second mark, we see the two thin vertical lines indicating future retransmissions, and right after that we see the congestion window and the UNACK-COUNT lines suddenly going down and up (top graph, light gray and black lines). This is the signal that 3 duplicate ACKs were received and the Fast

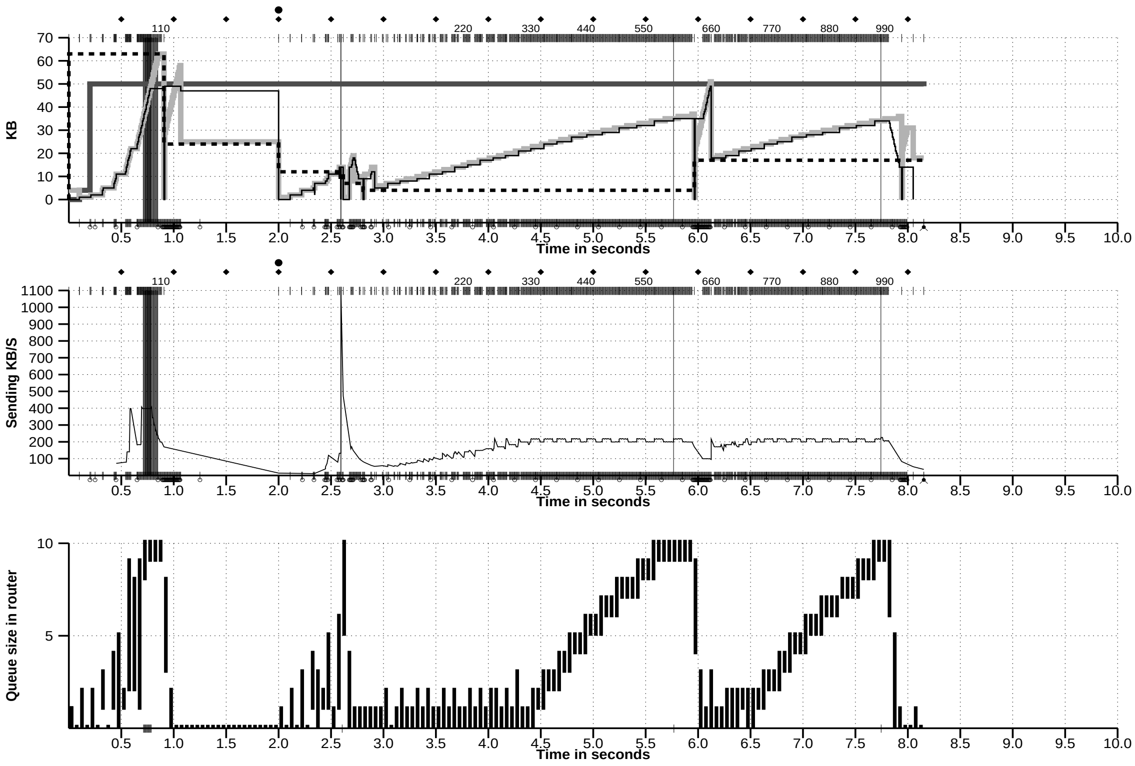

Figure 4: TCP Reno with No Other Traffic (Throughput: 123 KB/s).

Retransmit and Fast Recovery mechanisms went into action (a detailed description of the window behavior is given in the Appendix).

A little after 8 seconds we see the UNACK-COUNT line go straight down and up again. The fact that it went down signifies that a packet was received acknowledging some data, and the length of the lines tells us that it acknowledged more than 20KB. The congestion window should have been fixed at this point (made equal to the threshold window), since it was inflated to allow the pipe to stay full. But the congestion window wasn't fixed, and more than 20KB of data were sent at that time (the amount of data acknowledged) creating a huge spike in the sending rate (middle graph).

From the textual traces, we observe that the packet whose acknowledgment should have triggered the reduction of the congestion window was handled by the header prediction code. This is the root of the problem. The header prediction code is only supposed to handle packets that involve little work, and it doesn't check for inflated congestion windows, which are, after all, a rare occurrence.

The fix is to add one more test to the part of the header prediction that handles pure ACKs for outstanding data, replacing

```
if (tlen == 0) {
        if (SEQ_GT(tHdr.th_ack, tp->snd_una) &&
            SEQ_LEQ(tHdr.th_ack, tp->snd_max) &&
            tp->snd_cwnd >= tp->snd_wnd) {
```

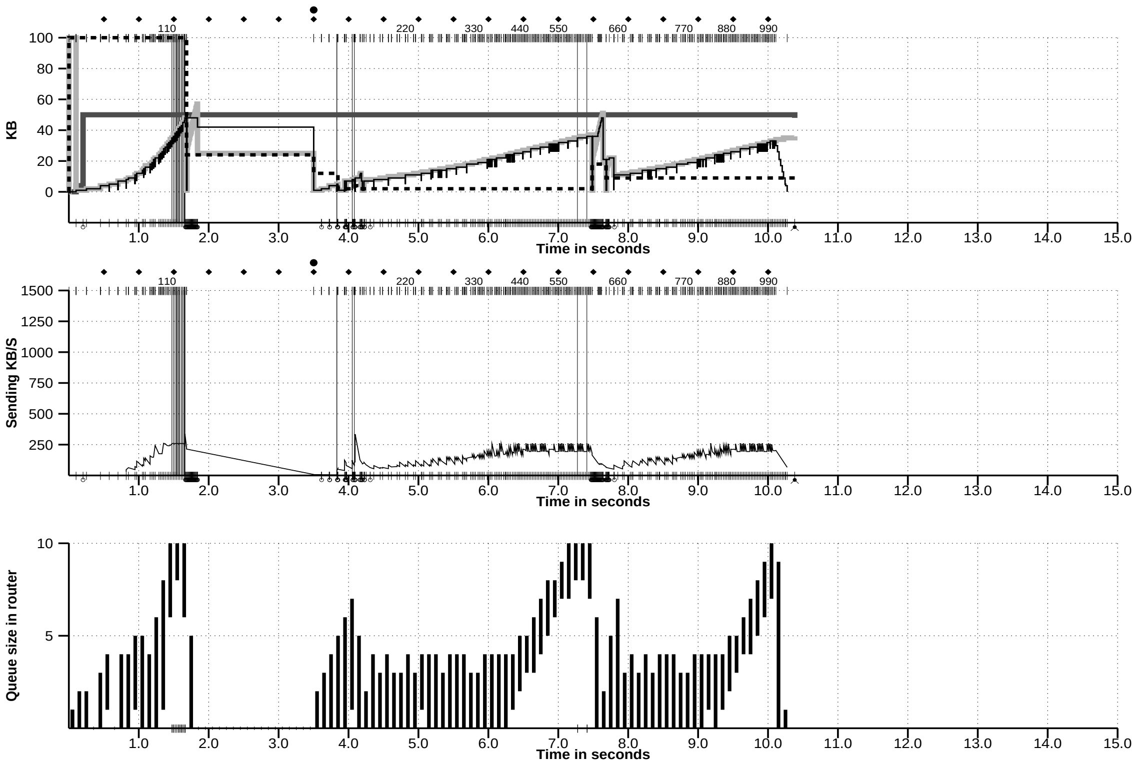

Figure 5: TCP Lite.1 with No Other Traffic (Throughput: 99 KB/s).

with

```
if (tlen == 0) {
        if (SEQ_GT(tHdr.th_ack, tp->snd_una) &&
            SEQ_LEQ(tHdr.th_ack, tp->snd_max) &&
            tp->snd_cwnd >= tp->snd_wnd
            && tp->t_dupacks < tcprexmtthresh) {
```

The behavior of TCP Lite with this fix, which we refer to as TCP Lite.1, is shown in Figure 5. Even though the performance increases considerably for this specific simulation, we see an average 2% decrease in throughput on the more complex simulations which also include *tcplib* traffic. The reason is that the problem we fixed—the congestion window not going down after being inflated to keep the pipe full—results in limiting the amount of unacknowledged data allowed. This problem has a similar effect as using a smaller send buffer, which if happens to be chosen right, can increase the throughput of the TCP connection by preventing the connection from overruning the bottleneck queue. For example, experiments that have heavy background traffic, where we would expect limiting the send buffer would be most beneficial, show that Lite.1 has 13% less throughput that Lite.0 (the original version), while experiments with light background traffic show that Lite.1 has 3% more throughput than Lite.0. These results help to illustrate just how complex and unintuitive TCP behavior can be.

3.2 Suboptimal Retransmit Timeout Estimates

The retransmission timeout value (RTO) calculation in TCP Lite closely follows the code described in the updated version[3] of Jacobson's '88 paper [4], . The RTO is based on a, an average round-trip time (RTT) estimator, and d, a mean deviation estimator of the RTT, as follows:

$$rto \leftarrow a + 4d$$

Given m, a new RTT measurement, the estimators are updated as follows:

$$Err \equiv m - a$$

$$a \leftarrow a + g_0 Err$$

$$d \leftarrow d + g_1(|Err| - d)$$

The values chosen by Jacobson for the gain parameters are $g_0 = .125 = \frac{1}{8}$ and $g_1 = .25 = \frac{1}{4}$, which allow the use of integer arithmetic by keeping scaled versions of a and d. Jacobson multiplies both sides of the equations by the following factors:

$$2^3 a \leftarrow 2^3 a + Err$$

$$2^2 d \leftarrow 2^2 d + (|Err| - d)$$

Then if we define $sa = 2^3 a$ and $sd = 2^2 d$ to be scaled versions of a and d, we have:

$$sa \leftarrow sa + Err$$

$$sd \leftarrow sd + (|Err| - (sd >> 2))$$

The whole algorithm can be expressed in C as:

```
m -= (sa >> 3);
sa += m;
if (m < 0)
        m = -m;
m -= (sd >> 2);
sd += m;
rto = (sa >> 3) + sd;
```

[3] Available at ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z

Where the RTO is in units of clock ticks.

Jacobson further describes how, in general, this computation will correctly round rto. Although this algorithm is a major improvement over the original algorithm described in RFC793 [6], there seems to be a problem with this algorithm based on the large delay observed in Figure 3, at the point before the retransmit timeout fires (indicated by the large black circles at the top of the graph). For the first timeout at 3.5 seconds, there is a delay of 1.7 seconds between the timeout and the previous packet sent. For the second timeout at 10.5 seconds, the delay is 2.5 seconds.

Given that the average RTT is about 150ms, the timeout delays seem much longer than necessary. To better understand the problem, we can look at the textual traces shown in Table 1 to see the behavior of the rtt, sa, sd and the rto.

RTT	sa	sd	RTO
0	8	2	3
0	7	3	3
0	7	3	3
1	8	4	5
0	7	4	4
0	7	3	3
0	7	3	3
0	7	3	3
0	7	3	3
0	7	3	3
1	8	4	5

Table 1: Original RTO Related Values

The values shown for sa, sd and rto are taken after they have been updated based on the new rtt measurement. The RTT is measured using a clock that ticks every 500ms; hence it is usually zero except for those packets where the clock ticked in between sending the packet and receiving the acknowledgment. Also note that in practice, the RTO is not allowed to go below 2, the minimum feasible timer (see Figure 6).

Something seems amiss by the fact that the rto stays at 3 regardless of how often the rtt is zero. From the equations we see that when m (new RTT) equals a (averaged RTT, $a \equiv (sa >> 3)$), then sa is not modified. Furthermore, if $(sd >> 2)$ is zero, then the two lower bits of sd are not modified either. The fact that m is usually either zero or one results in the fractional bits of a and d (lower three bits of sa and lower two bits of sd) being set most of the time.

As a consequence, $4sd$ usually contributes 3 ticks to rto rather than the 1.75 described by Jacobson.[4] These three ticks would be of no great importance if m, the measured RTT, was measured with a finer clock, so that it usually had a larger value—3 out of 100 is only 3%, 3 out of 1 is 300%!

Next we describe a simple modification to the RTO algorithm which solves the problem for the conditions of our experiment, where the RTT is either 0 or 1. This modification allows us to test the improvement

[4] The 1.75 is a statistical average assuming a wide range of RTT values, but in practice, the RTT is usually either 0 or 1. See Figure 6 for the reason why the bias is needed.

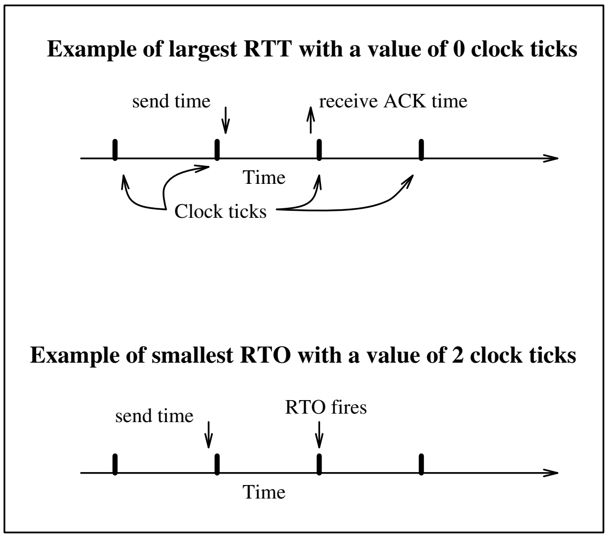

Figure 6: RTT and RTO examples.

in performance that is possible with a better RTO algorithm. However, we are not fully satisfied that this algorithm is the correct one under all possible scenarios. We are currently investigating this issue and hope to have a recommendation in the near future.

Our modification is to use a larger scaling factor:

$$2^2Err \leftarrow 2^2m - 2^2a$$

$$2^5a \leftarrow 2^5a + 2^2Err$$

$$2^4d \leftarrow 2^4d + 2^2|Err| - 2^2d$$

Now, if we define $sErr = 2^2Err$, $sa = 2^5a$ and $sd = 2^4d$ to be scaled versions of Err, a and d, we have:

$$sErr \leftarrow (m << 2) - (sa >> 3)$$

$$sa \leftarrow sa + sErr$$

$$sd \leftarrow sd + (|sErr| - (sd >> 2))$$

Then the low order bits that are not modified (and usually end up set) after m has the same value repeatedly do not affect the RTO calculation.

Since we want to be conservative when setting the RTO, and by conservative we mean that we never want to retransmit as a result of choosing too small an RTO, this implies that the RTO should always be at least two larger than the RTT. For example, the largest possible RTT with a value of x ticks has a real time length of just less than $x + 1$ clock ticks (see Figure 6), but the smallest possible RTO with value $x + 2$ has real time length of just greater than $x + 1$ tics (again see Figure 6).

Note that if there is access to a more accurate clock, as in TCP Vegas, then the coarse RTO can be made more accurate since we do not have to worry about the extreme cases described in the previous paragraph.

For example, if the more accurate RTO is less than 200ms, then as long as the segment is sent within the first 300ms after the coarse grain clock ticked, we can set the coarse rto to 1 (instead of at least 2). We also don't have the large jumps in the size of the RTO which come from the RTT jumping between 0 and 1.

The whole algorithm can be expressed in C as:

```
sdelta = (m << 2) - (sa >> 3);
sa += sdelta;
if (sdelta < 0)
        sdelta = -sdelta;
sdelta -= (sd >> 2);
sd += sdelta;
rto = max( m + 2, ((sa >> 3) + sd) >> 2 );
```

RTT	original RTO	new RTO	float RTO
0	3	3	3.4
0	3	3	3.5
0	3	3	3.5
0	3	3	3.4
1	5	3	3.1
0	4	3	3.1
0	3	3	2.9
0	3	3	2.8
0	3	3	2.5
0	3	2	2.3
0	3	2	2.1
1	5	3	2.5

Table 2: Original and New RTO Values

The new mechanism approximates the real RTO, obtained by using floating point computations, more closely as can be seen in Table 2. The new RTO more closely matches the RTO obtained using floating point computation, and unlike the old RTO, the new RTO can go all the way down to a value of 2 after the RTT has a value of 0 repeatedly.

The version of TCP Lite with both the header prediction and RTO computation fixes, which we refer to as Lite.2, has a throughput of 102.8 KB/s for the 1 MByte transfer when there is no other traffic. The average throughput for the set of tests with *tcplib* background traffic is 55.9 KB/s, an increase over the original of 9%. However, this is still 6% below the throughput of TCP Reno.

3.3 Options and ACKing Frequency

TCP Lite's ACKing frequency is affected by the use of options. When there are no options in use, TCP Lite ACKs every other packet. When options (e.g., timestamp) are used, however, it ACKs every third packet. The reason is that the test which decides to send an ACK is based on how much data has been received, and

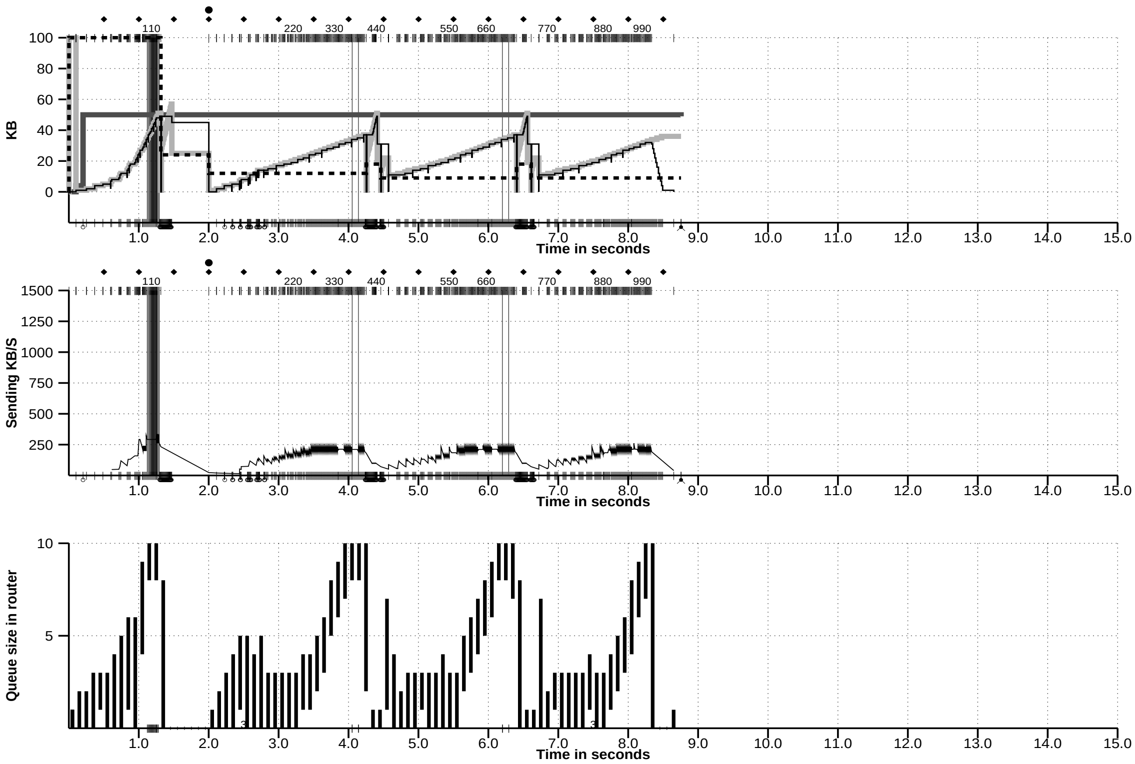

Figure 7: TCP Lite.3 with No Other Traffic (Throughput: 117 KB/s).

if it is greater than or equal to more than twice the maximum segment size, an ACK is sent. However, since TCP options take some of the available payload space, two segments that contain an option no longer hold enough data to trigger sending an acknowledgment.

As discussed below, ACKing frequency affects the growth of the congestion window during both slow-start and the linear increase period. We disabled the timestamp option as a quick way to test the effect of modifying the code which handles the ACKing frequency so it would not be affected by the use of options[5] . We refer to this version as Lite.3, and its throughput for the 1 MByte transfer experiments was 177 KB/s. However, only about half of the throughput gain was due to the higher ACKing frequency; the rest of the gain was due to the fact that the RTO happened to be 2 instead of 3 when the losses occurred, decreasing the delay until the RTO fired by half a second. In the experiment when there is also background traffic, the average throughput increased by 5%, but the losses increased by 28% when compared to TCP Lite.2.

Figure 7 shows the graphical traces of Lite.3 when transferring 1 MByte. The important thing to notice is how the losses are paired when they occur around 4 and 6 seconds, resulting in decreasing the congestion window twice. The reason the losses are paired is that the congestion window is increasing too rapidly—by more than one maximum segment per RTT. Before the first loss is detected, TCP Lite increases the congestion window, again resulting in an extra lost packet.

[5]Disabling the timestamp option is not meant as the solution to this problem, it was only done to examine the effect of the problem. We plan to make a fix available in the near future.

The reason the congestion window is increasing so fast is the extra $1/8^{th}$ of a maximum segment being added to it[6] :

```
if (cw > tp->snd_ssthresh)
        incr = incr * incr / cw + incr / 8;
```

By removing this $1/8^{th}$ increase:

```
if (cw > tp->snd_ssthresh)
        incr = incr * incr / cw;
```

the losses are not paired any more, so the congestion window is decreased only once. In the experiments with background traffic, running TCP Lite.4, which includes the four previous modifications (header prediction fix, RTO fix, no timestamps, and eliminating the extra $1/8^{th}$ increase) results in an average throughput of 62.3 KB/s. This represents a 21% increase over the original TCP Lite, and a 6% increase over the previous one (Lite.3).

3.4 Linear Increase of the Congestion Window and ACKing Frequency

TCP should perform a multiplicative decrease of the congestion window when losses are detected, and a linear increase while there are no losses detected. Due to the bugs described above, TCP Lite follows no specific guideliness on how fast to increase the congestion window: if there are options, the acking frequency is reduced and the window increases more slowly; by using the extra $1/8^{th}$ of a maximum segment size the rate of increase is now a function of the window size (which means the increase is exponential).

It is easy to specify what the upper bound to the rate of increase for the congestion window during the linear increase mode should be: the congestion window should not be increased by more than one maximum segment size per RTT. Increasing the congestion window at a faster rate will result in the loss of two or more segments per RTT. This may result in a timeout since the fast retransmit and fast recovery mechanisms in TCP Lite cannot always recover from the loss of two segments when the losses occur during a one RTT time interval.

ACKing frequency has been seen as a tradeoff between higher congestion and lower latency. As shown here, it also affects the rate of the linear growth of the congestion window in TCP Lite—this is due to implementation, and it could be changed. However, it is not clear that ACKing every two or three packets puts less stress in the network than acking every packet. On the one hand, lower ACKing frequencies imply less packets on the network, but on the other hand, lower ACKing frequencies result in more packets being sent at one time. For example, if the receiver ACKs after receiving three packets, then the sender will transmit 3 or 4 packets at one time (the fourth one if the congestion window linear increase allows one more packet to go out). If two or more connections do it at the same time, then the likelihood of losses increases. Another example resulting in increased stress on the network would be of a server which doesn't reply to requests immediately. This would result in sending all of the delayed acknowledgments at the same time every 200ms.

[6] This problem has already been pointed out by Sally Floyd.

We modified the code, creating Lite.5, so that it would ACK after every packet rather than after every two packets. The effect on the tests with traffic was a slight increase in the throughput (3%) but a much larger increase in the losses (48%). The increase in the losses may be due to our simulation scenario—e.g., number of buffers at the router. We are planning to look into this issue in more detail, since we do not feel it has been settled. Until then, we prefer to take the conservative approach and use delayed ACKs in TCP Lite.

3.5 Handling Big ACKs

The response of TCP Lite (and previous BSD versions) when it receives a packet acknowledging x bytes, is to send x bytes immediately, as long as the windows allow it. If x is a large number, equivalent to 4 or more packets, this behavior usually results in losses due to packets being dropped at the bottleneck.

Two causes for large acknowledgments are (1) losses that result in a retransmit timeout, and (2) packets received out of order. The fast retransmit and fast recovery mechanisms try to keep the pipe full after a loss is detected by receiving three duplicate ACKs. This means that after retransmitting the lost segment, TCP keeps sending (new) data at a rate half of what it was when the loss was detected. The receiver cannot acknowledge the new data since it is missing one or more of the earlier segments. If other packets following the packet originally lost are also lost, or if the retransmitted packet is lost, the pipe will likely empty and a retransmit timeout will be needed to start sending again. After the segment that was lost is retransmitted, the receiver will now be able to acknowledge not only the retransmitted packet, but also all of the segments following the retransmitted segment which were sent to keep the pipe full.

The fix to prevent sending too many segments at once is very simple. When a large acknowledgment is detected, if the threshold window is smaller than the congestion window, set it to the value of the congestion window. Then set the congestion window to a lower value so only 2 or 3 segments are sent at one time. As new acknowledgments are received, the congestion window will then increase exponentially to the correct level, as specified by the threshold window. An example of the fix follows:

```
if (acked >= 3*tp->t_maxseg  &&
     (tp->snd_cwnd - (tp->snd_nxt - tp->snd_una)) > 3*tp->t_maxseg) {
  if (tp->snd_cwnd > tp->snd_ssthresh)
    tp->snd_ssthresh = tp->snd_cwnd;
  tp->snd_cwnd = (tp->snd_nxt - tp->snd_una) + 3*tp->t_maxseg;
}
```

This code needs to be inserted after the window information is updated. Adding this fix to Lite.4 (resulting in Lite.6) did not affect the results very much, indicating that the problem is rare, at least under the conditions of the experiments.

3.6 Final Details

There is one final detail. The test that checks if the congestion window needs to be fixed because it was inflated trying to keep the pipe full, is slightly wrong. The original test

```
tp->dupacks > tcprexmtthresh
```

TCP Version	Fixes
Lite.0	None. This is the original TCP BSD4.4-Lite code
Lite.1	Header prediction fix
Lite.2	Previous + RTO fix
Lite.3	Previous + not using timestamp option
Lite.4	Previous + removing the extra 1/8 increase of the congestion window
Lite.5	Previous + acking on every packet
Lite.6	Lite.4 + fix to prevent sending to much at one time due to large ACKs

Table 3: Description of Different Versions of TCP Lite.

should be changed to

```
tp->dupacks >= tcprexmtthresh
```

However, this will probably not have much of a practical effect.

4 Comparing the Different Versions of TCP Lite

This section compares the average throughput and average losses of the different versions of TCP Lite. The experiment consisted of running 20 simulations in which there is a distinguished 20 second transfer sharing the bottleneck link with *tcplib* generated traffic running over the same version of TCP as the distinguished transfer. The background traffic uses between 30 and 80% of the bottleneck bandwidth.

There are 7 versions of TCP Lite, denoted by Lite.0 to Lite.6, which represent the different cumulative fixes applied to TCP Lite. Table 3 describes the fixes applied to the different versions. The purpose of Lite.3, which does not use the timestamp option, is to see the effect of fixing the code which decides when to send an acknowledgment so the acknowledgment frequency is not affected by the use of options.

	Lite.0	Lite.1	Lite.2	Lite.3	Lite.4	Lite.5	Lite.6	Reno	Vegas
Throughput (KB/s)	51.3	50.4	55.9	58.5	62.3	64.3	61.5	59.3	82.8
Throughput Ratio	1.00	0.98	1.09	1.14	1.21	1.25	1.20	1.16	1.61
Retransmissions	3.7%	4.5%	3.6%	4.6%	3.3%	4.9%	3.8%	4.5%	2.8%

Table 4: 20sec Transfer with *tcplib*-Generated Background Traffic.

Table 4 shows the average throughput, the throughput ratio with respect to Lite.0, and the percent of bytes retransmitted. The numbers shown are the averages of the 20 runs using different background traffic patterns. As can be seen from the table, Lite.4 does 21% percent better in terms of throughput that the original code (Lite.0). Based on the current experiments, we recommend the modifications to TCP Lite corresponding to Lite.4 (keeping in mind the need for more testing of the RTO modifications).

Our version of Reno has lower throughput (5%) and higher losses (36%) than Lite.4. The reasons are that our version of Reno does not delay ACKs, and it does not contain the the RTO enhancements described in section 3.2[7] . When comparing to Lite.4, TCP Vegas shows an improvement of 33% under our simulation parameters, and a 15% decrease in the number of retransmitted bytes. The improvement in losses is less than that reported earlier based on our implementation of Reno.

5 Concluding Remarks

We have described code improvements to the BSD4.4 version of TCP (TCP Lite) that result in a 21% throughput increase under our simulation scenarios. One of the major lessons we have learned from this exercise is that TCP's robustness—its ability to complete data transfers under the worst conditions—makes coding errors that much harder to find. It would be much easier to find these problems if they resulted in breaking TCP, but breaking TCP is hard. Instead, sophisticated analysis tools are required to find performance-related bugs in TCP.

We also see that TCP Vegas achieves significantly higher throughput than the improved version of TCP Lite. This increased throughput does not come at the expense of the background traffic, but as a result of better utilization of the bottleneck link. The average bottleneck utilization during the 20 second transfer was 82% when using TCP Lite but 91% when using TCP Vegas.

Acknowledgements

We would like to thank Sally Floyd for her insightful comments.

References

[1] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proceedings of the SIGCOMM '94 Conference*, pages 24–35, Aug. 1994.

[2] P. Danzig and S. Jamin. tcplib: A Library of TCP Internetwork Traffic Characteristics. Technical Report CS-SYS-91-495, Computer Science Department, USC, 1991.

[3] N. C. Hutchinson and L. L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.

[4] V. Jacobson. Congestion Avoidance and Control. In *Proceedings of the SIGCOMM '88 Symposium*, pages 314–32, Aug. 1988.

[5] W. R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley Publishing Co., New York, 1994.

[6] USC. Transmission control protocol. Request for Comments 793, USC Information Sciences Institute, Marina del Ray, Calif., Sept. 1981.

[7] Note that our version of Reno doesn't have the $1/8^{th}$ increase, this is one of the reasons why it did better than the original TCP Lite.

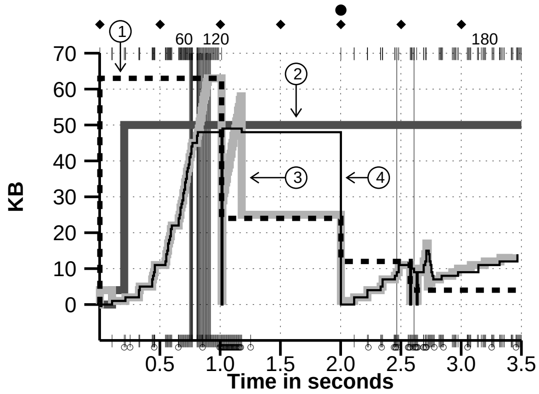

Figure 8: TCP Windows Graph.

A Detailed Graph Description

To assist the reader develop a better understanding of the graphs used thoroughout this paper, and to gain a better insight of Reno's behavior, we describe in detail one of these graphs. Figure 8 is a trace of Reno when there is other traffic through the bottleneck router. The numbers in parenthesis refer to the type of line in the graph.

In general, output is allowed while the UNACK-COUNT (4) (number of bytes sent but not acknowledged) is less than the congestion window (3) and less than the send window (2). The purpose of the congestion window is to prevent substained congestion. The send window is used for flow control; it prevents data from being sent when there is no buffer space availabe at the receiver.

The threshold window (1) is set to the maximum value (64KB) at the beginning of the connection. Soon after the connection is started, both sides exchange information on the size of their respective receive buffers, and the send window (2) is set to the minimum of the sender's send buffer size and the receiver's advertized window size.

The congestion window (3) increases exponentially while it is less than the threshold window (1). At 0.75 seconds, losses start to occur (indicated by the tall vertical lines). More precisely, the vertical lines represent segments that are later retransmited (usually because they were lost). At around 1 second, a loss is detected after receiving 3 duplicate ACKs and Reno's Fast Retransmit and Fast Recovery mechanisms go into action. The purpose of these mechanisms is to detect losses before a retransmit timeout occurs, and to keep the pipe full (we can think of a connection's path as a water pipe, and our goal is to keep it full of water) while recovering from these losses.

The congestion window (3) is set to the maximal allowed segment size (for this connection) and the UNACK-COUNT is set to zero momentarily, allowing the lost segment to be retransmitted. The threshold window (1) is set to half the value that the congestion window had before the losses (it is assumed that this is a safe level, that losses won't occur at this window size).

The congestion window (3) is also set to this value after retransmitting the lost segment, but it increases with each duplicate ACK (segments whose acknowledgement number is the same as previous segments and carry no data or new window information). Since the receiver sends a duplicate ACK when it receives a segment that it cannot acknowledge (because it has not received all previous data), the reception of a duplicate ACK implies that a packet has left the pipe.

This implies that the congestion window (3) will reach the UNACK-COUNT (4) when half the data in transit has been received at the other end. From this point on, the reception of any duplicate ACKs will allow a segment to be sent. This way the pipe can be kept full at half the previous value (since losses occurred at the previous value, it is assumed that the available bandwidth is now only half its previous value). Earlier versions of TCP would begin the slow-start mechanism when losses were detected. This implied that the pipe would almost empty and then fill up again. Reno's mechanism allows it to stay filled.

At around 1.2 seconds, a non-duplicate ACK is received, and the congestion window (3) is set to the value of the threshold window (1). The congestion window was temporarily inflated when duplicate ACKs were received as a mechanism for keeping the pipe full. When a non-duplicate ACK is received, the congestion window is reset to half the value it had when losses occurred.

Since the congestion window (3) is below the UNACK-COUNT (4), no more data can be sent. At 2 seconds, a retransmit timeout occurs (see black circle on top), and data starts to flow again. The congestion window (3) increases exponentially while it is below the threshold window (1). A little before 2.5 seconds, a segment is sent that will later be retransmitted. Skipping to 3 seconds, we notice the congestion window (3) increasing linearly because it is above the threshold window (1).