

High Bandwidth TCP Queuing

John W. Heffner
Peter Steenkiste, advisor

July 24, 2002

Abstract

The available bandwidth on some Internet paths have increased by several orders of magnitude over the past decade. This trend is likely to continue in the future. Consequently, connections on a single host can and will vary in bandwidth by several orders of magnitude. TCP, now the ubiquitous protocol for delivering reliable data across the Internet, requires queues proportional in size to a path's bandwidth. Traditional mechanisms for allocating and limiting TCP queue space fall short on today's Internet, and often limit throughput to only a small fraction of available bandwidth. It is this project's aim to modify the TCP implementation of Linux 2.4 in such a way as to remove these barriers while using memory as efficiently as possible.

1 Introduction

Over the past two decades, TCP has emerged as the ubiquitous transport protocol for delivering data reliably over the Internet. All Web (HTTP) traffic, most bulk file transfers, and a variety of other network applications use TCP.

Also over the past decades, bandwidths of Internet paths over which TCP runs have been following a variation on Moore's law — that is, available bandwidth has been increasing exponentially with time. TCP, however, has not kept pace with other networking improvements. One reason for this is that its queues require space proportional in size to path bandwidth, but space has historically been allocated for these queues statically. As bandwidths have grown, the static queue sizes have proven to be significant bottlenecks.

It is possible to increase the static allocation with parameters made available by the operating system; however this has proven ineffective for a number of reasons. First, setting these correctly requires networking expertise which an

©2002 John Heffner. All rights reserved.

end user or application programmer should not be expected to have. Also, correctly setting space requires per-connection knowledge of the network, which may vary quickly with time and widely across connections, making it impossible for a user to generally set parameters correctly.

A user may try to solve the problem by simply “throwing memory at it,” and setting allocation sizes very large. However, this does not work correctly as it may waste a large amount of memory *per connection*, and also breaks TCP’s flow control mechanism.

1.1 Related work

An “auto-tuning” mechanism is proposed in [4], which sizes the sender’s buffer based on the congestion window. This approach works fairly well, as the send buffer size will not limit window size. Some memory may be wasted, but not more than a constant factor of the window size. This was approximately implemented in Linux 2.4. However, the Linux implementation still sets a per-connection buffer size limit, so this, in the default configuration, acts only as a memory-saving device.

For that paper, the receive buffer was simply set very large, which breaks flow control, so is not a viable solution for the general case. It only works when the receiver is always able to consume data at network rates. In [5], a method called Dynamic Right-Sizing was presented which solves this problem by sizing the receive buffer based on observed network properties.

1.2 Contributions

The goal of this paper is to present a framework for clearly understanding how TCP buffer sizes and queues interact to affect performance, and using this understanding to solve problems in current implementations. In particular, the flow control problem is solved using Dynamic Right-Sizing techniques with an alternate round-trip time measurement, and by using a buffer size based on data sequence rather than memory. The send buffer problem is solved by separating the application buffer from the retransmit queue. All solutions were implemented in Linux 2.4 and tested to show utility.

1.3 Organization

In Section 2, relevant characteristics of TCP are explained. In Section 3, the bandwidth-delay product (BDP) is defined and its relevance explained. In Section 4, the problem caused by the flow control mechanism is explained and a solution is proposed. In Section 5, the problem of the retransmit queue size is explained and a solution is proposed. In Section 6, the details of implementation of the proposed solutions in the Linux kernel is described. Section 7 describes the testing environment used for evaluation of the implementation, and Section 8 describes the tests performed and presents the results of those tests. Conclusions are presented in Section 9.

2 A brief review of TCP

This is a brief explanation of the basic characteristics of TCP relevant to this paper. This should be entirely review for a reader familiar with the workings of TCP. Details on most of this information may be found in [1] and [2], or in a more easily readable yet detailed description [3].

2.1 Reliable delivery

TCP's primary function as a transport protocol is ensuring reliable in-order delivery of data from one host to another over a packet network which may drop or reorder packets. In order to meet this requirement, TCP creates connections, sets of state kept at both end hosts containing information about the progress of the data transfer. A data stream sent through a TCP connection is assigned a sequence space so that each 8-bit byte is numbered sequentially. The stream is divided into segments so that it can be put in packets. A receiving host will cumulatively acknowledge data it has received by sending the sequence number of the last in-order byte it has received. A receiver should queue out-of-order data until the holes in the stream are filled. This data is stored in the *reassembly queue*. A sender must be prepared to retransmit any unacknowledged data since the network may have dropped any of its outstanding packets. This data is stored in the *retransmit queue*.

2.2 Round-trip time

A round-trip time (RTT) between two hosts is defined as the length of time between when an end host sends a packet and when it receives a reply from the other end host. This total time consists of four parts.

Transmission time The length of time the interface takes to send the packet. This will likely have a linear dependence on packet size.

Propagation delay The time taken for a signal to propagate through the network medium. This is equal to distance between endpoints divided by propagation speed (bounded by and often close to the speed of light).

Routing queues Time spent in routing queues along the path. This will fluctuate over time depending on congestion in the path.

Processing time The amount of time spent by end hosts and routers doing processing necessary for packet delivery. On current hardware, this is usually in the sub-microsecond range.

It is important for the TCP's retransmission and congestion control algorithms to have a good estimate of a connection's RTT. It is sampled by measuring the length of time between transmission of a data segment and reception of its acknowledgment. A host whose end of a TCP connection does not send data will not have a measurement of the RTT.

2.3 Windows

A *window* of data is the amount of data transmitted on a connection during one round-trip time. A TCP sender will have one window of outstanding (unacknowledged) data. Throughput is equal to $window/RTT$. Since RTT should be relatively constant over a given path, the throughput is approximately proportional to the window size. There are a number of bounds on the window size:

Receive window It may be that a receiving application is not able to process data as quickly as the sending application or the network can transmit it. In this case, it is necessary to have a flow control mechanism to slow the transmission rate of the sender. TCP implements flow control by announcing a *receive window* in each segment header. This window announced by the receiver is an upper bound on the sender's window size.

A classical TCP implementation maintains a buffer of a static size between itself and the receiving application. It then announces its receive window as the amount of space available in that buffer. If the application reads slower than data arrives, the buffer will start to fill. As the buffer fills, the window size and therefore throughput will fall (eventually reaching zero when the buffer fills entirely). Throughput will quickly match the application's consumption rate.

Retransmit queue As stated above, the sender must keep all outstanding data in a retransmit queue. If the amount of memory available for this queue is an upper bound on window size.

Congestion window In many cases, the amount of traffic coming into a router is greater than its outgoing bandwidth. A finite quantity of packets may be queued in the router, but in a steady state a certain percentage must be dropped. In this case, the router is said to be a bottleneck and experiencing congestion. One of TCP's functions is to detect this congestion (usually by observing lost packets) and limit its transmission rate accordingly. A TCP sender maintains a *congestion window*, an upper bound on window size based on the observed properties of the network.

Sending application The amount of data the sender produces per round-trip time is yet another bound on window size. If a sender has little or no data to send, the window will be accordingly small. For example, an interactive application such as telnet will likely send between zero and a few segments per round-trip time.

3 The bandwidth-delay product

If a connection's throughput is limited only by the network, its window size will be limited by the congestion window (*cwnd*). In this case, we ideally

have: $window = cwnd$ and $throughput = bandwidth$. Since $throughput = window / RTT$, we have $bandwidth = cwnd / RTT$. Therefore, we have:

$$cwnd = bandwidth * RTT. \tag{1}$$

This quantity is known as the bandwidth-delay product, or BDP. (It should be noted that this $cwnd$ is the ideal congestion window. In practice, bandwidth is not known and may change over time, so the congestion control mechanism grows and shrinks $cwnd$ as it drives the network into congestion.)

The RTT between two endpoints is fundamentally limited by the distance between them and the speed of light. The highest bandwidths, however, have been growing exponentially with time (similar to the way processors follow Moore's "law"). Consequently, BDPs across the Internet are also increasing over time. Further, connection BDPs may vary by many orders of magnitude on a single host. For example, a host connected with 100 Mbps fast Ethernet may connect to a similar host at the same time as it connects to one with a 56 kbps modem.

4 Flow control

The typical flow control mechanism described above suffers from one serious problem — the largest possible receive window is the less than the fixed size of the receive buffer. If this maximum window is less than the BDP, the connection can never reach the throughput of which the network is capable. The size of this receive buffer is settable in most implementations both globally by an administrator, and per-connection by the application; however, as described above, these solutions are undesirable in the general case.

4.1 Using a large receive buffer

Setting the receive buffer size to effectively infinite as proposed in [4] works in the case when the receiving application is able to process data as fast as the network can deliver it, since the buffer will never fill. However, if the application is slower than the network, the buffer will continue to grow indefinitely, consuming memory indefinitely.

Further, a buffer too big will have an undesirable effect in interactive applications. For example, a telnet user whose server starts to output a large amount of text may wish to hit Control-C to stop the output. The telnet application will have to display a full window of data sent after the Control-C is sent.

It might be possible to improve the large receive window approach by using the congestion control mechanism to do flow control as well. The receiver's buffer could be viewed as a router queue, and it could trigger congestion events (drop packets or use ECN) to pull down the sender's congestion window.

This method has some serious problems, however. For one, if the receiving application completely stops, there will be no way to announce a zero window. If the receiver drops all incoming packets, the sender will try to retransmit until the connection times out and resets. Another problem is that this method

would not solve the interactive application problem. A third undesirable property is that it would not respond quickly if the receiving application changed its consumption rate quickly. The congestion control mechanism is designed to conservatively estimate network properties, which are not likely to change rapidly like an application might.

4.2 Appropriately sizing the receive buffer

If we are to use the receive window for flow control as intended, the announced receive window will ideally be the minimum of the amount of data a receiving application would like per round trip and the amount of data the sender may send per round trip. To calculate such a window, however, a receiver must know the connection's RTT, information not generally available to a receiver. RTT is traditionally observed by the sending side by measuring the time between when a data segment is sent and its corresponding acknowledgment is received. If one end host never sends data on a particular connection, it will have no knowledge of the RTT.

4.2.1 Dynamic Right-Sizing

A method for measuring a connection's RTT and appropriately announcing the receive window was proposed by M. Fisk and W. Feng in [5]. They call their method Dynamic Right-Sizing, or DRS. This measures the amount of data received in a round trip and sizes the receive buffer to fit twice that amount. This measurement and resizing is done once per RTT.

If no RTT sample is available from sending data, Fisk and Feng propose a new technique for estimating RTT. They use the fact that a sender may not send more than an announced window of data within one RTT of its announcement. Therefore, the time between when an acknowledgment for sequence number s announcing receive window w is sent and a data segment containing sequence number $s + w + 1$ is received is an upper bound on RTT. They keep a minimum of these sampled upper bounds. If a smoothed RTT from sending is available, that is used; otherwise, the minimum upper bound is used.

Assuming an accurate RTT measurement, DRS will ensure that in a steady state, the receive window will not limit window size if the receiver consumes all data quickly. If the receive window limits window for one round trip, it will be doubled on the next round trip. (It is worth noting that this doubling matches "slow start," the congestion control state which most quickly grows the congestion window.)

In the case where the receiver consumes data slowly, flow control will work. As the receive buffer fills with unread data, the announced window size will shrink, so less data will be sent per RTT. With less data sent, the target buffer size will be calculated smaller by DRS until it is twice the size of the amount of data read by the receiver per round trip. Then, in the steady state, the receive window will be the amount of data read per round trip by the application, and an equal amount will be buffered to be read.

4.2.2 Receiver RTT measurement with timestamps

A peculiarity of the RTT measurement technique used by DRS is that it only accurately measures the RTT if the receive window controls flow. Otherwise, the measurement will be an overestimate.

There is a way to measure RTT more accurately using timestamps [6]. Most modern TCP implementations use timestamps unless header compression is used. When using timestamps, each segment header contains a 32-bit timestamp with the time it was sent, and a 32-bit echo of the most recent timestamp received. Subtracting the echo on an incoming data segment with the current time yields an upper bound on the RTT. As RTT is different depending on packet size, the RTT should only be bounded if the received segment is a full maximum segment size (MSS).

Either one of these estimates may be very high if the sending TCP does not always have data to send. However, the buffer will not grow very large in this situation, because the amount of data in the RTT will not be large.

The effects of an incorrect RTT measurement should be noted. If measured RTT is different than actual RTT, the buffer will be proportionally larger or smaller than it would be with an accurate measurement. The larger buffer will not limit throughput, but may under some circumstances waste memory. This should not be a problem with a well-behaved TCP, but a malicious TCP may be able to attack a receiver by inflating the RTT, and attempting to drain its resources. Defense against such an attack may be a subject of future work.

If the RTT measured is small, the buffer may grow slowly, or control flow in steady state if the measured RTT is less than half the true RTT. Since DRS uses the minimum of all sampled RTTs, this case will in fact arise frequently, as routing queue sizes vary significantly during connections. When a connection controls the queue sizes (it is causing the congestion), this actually has a positive effect, as documented in [5]. When the congestion occurs, the RTT increases, slowing transmission since window size will not increase as well. However, if cross-traffic is causing congestion, queue sizes, and hence RTT, may vary significantly, causing the connection to slow its transmission rates despite the fact it may not be driving congestion. It may be possible to track RTT variations at the receiver due to varying queue sizes with timestamps. This is also a possible subject of future work.

5 The retransmit queue size

The majority of TCP implementations today are at least conceptually based on Berkeley sockets. These implementations all use the socket send buffer as space to copy data from the application to the kernel, and keep it there until it can be freed when acknowledged. Typically, the size of this buffer has been limited to a relatively small fixed size — usually between 4 KB and 64 KB. Since the entire retransmit queue, whose size limits window, is kept in this buffer, a small buffer size will limit the connection's throughput. A buffer too large will waste

memory because a sending application which always has data to send will fill the buffer and keep it full through the life of the connection.

An solution was proposed in [4], which “auto-tunes” the buffer size inside TCP. It sets the socket send buffer size to between two and four times *cwnd*, so the send buffer will never limit window size (provided enough memory is available globally), but it will not waste a large amount of memory. In version 2.4 and higher, Linux has adopted this general approach. It sill, however, has a global per-connection upper limit to the socket send buffer size. This limit’s default value often controls throughput on today’s high-performance paths.

A more appealing approach is to separate the socket send buffer from the retransmit queue. These serve two distinctly different purposes. Network events happen in real time and should not have to wait for an application to be scheduled. The socket buffer decouples the application from the network. The retransmit queue, on the other hand, depends only on properties of the network. It will never grow in size beyond exactly what is needed by TCP. Separating the two simply makes the problem disappear. The retransmit queue size will never limit window size if enough memory is available.

6 Implementation

The DRS-style buffer sizing and separation of the retransmit queue described above were implemented in the Linux 2.4.17 kernel. These and some additional related changes are described below.

6.1 Important existing variables

The following are existing per-connection variables.

rcv_nxt The sequence number of the next expected byte. Corresponds to RCV.NXT from [1].

sndbuf The amount of space a socket’s send buffer may use. Its default value is a global sysctl variable. Its value may be changed at any time by an application using a `setsockopt()` system call. If an application has not changed the value, Linux 2.4 dynamically changes this value to be at least $2 * cwnd$.

wmem_queued The amount of memory actually allocated to data in the send buffer.

6.2 New variables

The following per-connection variables were added.

rcv_min_rtt Contains the minimum observed RTT of an MSS-sized segment or a full window.

rcv_rtt_seq The sequence number at which a receiver RTT measurement should end.

rcv_rtt_time The time at at which the RTT measurement begins, in system clock ticks.

rcv_winst_seq The sequence number at the beginning of a window size measurement.

rcv_winst_time The time at which the window size measurement begins, in system clock ticks.

rcv_space The dynamic amount of space available for the receive buffer.

rcv_alloc The amount of receive buffer space allocated to in-order data ready for the application.

ofo_alloc The amount of data in the out-of-order queue.

In addition, a global sysctl (user-settable) variable was added.

tcp_default_wscale A suggestion for the window scale option [6] value.

It should be noted that some of these new variables are similar in definition to existing variables; however, their meanings and uses are significantly different and so are described here.

6.3 Window scale

The field in the TCP header for the window size is 16 bits in length, enough only to advertise a receive window of size 65535. On a connection with a large BDP, it is necessary to announce windows correspondingly large. In [6], a *window scale* option is specified which negotiates at connection synchronization a factor by which announced receive windows will be scaled (left bit shifted). For example, with a window scale of 3, an announced receive window of 65535 will be $65535 * 2^3 = 524280$.

Linux currently sets the window scale so that a receive window corresponding to an empty receive buffer may be advertised. Since the modified implementation has a dynamically changing buffer size, it is necessary to select a window scale a different way.

Selecting a window scale, however, can be difficult, since it must be negotiated before a connection starts sending data, and therefore before any path characteristics are known. If the scale factor is too small, it will restrict the maximum window size. On the other hand, a large scale factor limits the granularity of receive window size. A user-settable default value of 7 is used. This should be enough for cross-continent 100 Mbps traffic ($10^8 * 0.07 = 7 * 10^6 < 8388480 = 65535 * 2^7$), yet small enough to announce an approximately single-segment window with a small MSS (granularity of 256 bytes compared to a 536 byte MSS for a modem or a 1460 byte MSS for Ethernet).

6.4 Receiver RTT measurement

Each time a new RTT sample is taken, this variable is set to $\min(rcv_min_rtt, sample)$. There are three places samples may be taken.

1. A transmitted MSS-sized data segment is acknowledged.
2. An MSS-sized data segment is received with a timestamp echo.
3. A data segment arrives containing sequence number rcv_rtt_seq .

One tick is added to the sample, because a fractional tick will be truncated. It is safer to overestimate than underestimate RTT, as the minimum of all samples is used, and because an underestimate may incorrectly restrict flow whereas an overestimate will at worst cost one bandwidth-tick's worth of memory when the receiving application is controlling flow.

6.5 Window measurement

At the start of the connection, rcv_space is initialized to $4 * MSS$, since a TCP sender's initial congestion window should be no greater than this size [7]. When rcv_min_rtt is first set, rcv_winest_seq is set to rcv_next , and $textit{rcv_winest_time}$ is set to the current system clock time, a global variable $jiffies$ in Linux. When a packet is received and $jiffies - rcv_winest_time \geq rcv_min_rtt$, rcv_space is set to $rcv_next - rcv_winest_seq + 4 * MSS$. It is set to no less than half its previous value to decrease the penalty for brief stalls in transmission.

6.6 Window announcement

The Linux 2.4 TCP keeps each packet in a `sk_buff` structure. This is allocated when the packet is first read in from the device and contains all headers as well as the payload. The space is not de-allocated until its data payload is read in by the receiving application. When these `sk_buffs` are kept in the receiving TCP's queues, their full size, including protocol headers and the structure's overhead space, is used for calculating available space in the buffer (and hence the announced receive window). Linux provides a `sysctl` variable, `tcp_adv_win_scale`, to reserve a fraction of the buffer space for overhead.

Also, as in most TCP implementations, the reassembly queue is kept in the receive buffer. The reassembly queue will normally be zero in size, or a few packets in size if there is some reordering occurring in the path. However, it will grow to a full window in size when loss occurs. Using this approach means that the announced receive window will shrink substantially during the round-trip time after a single packet is lost. Linux provides a `sysctl` variable, `tcp_app_win`, which can reserve a fraction of the buffer space which will only be used for application buffer.

This approach is somewhat awkward, requiring user-settable parameters, and arises from the restriction that a socket may not use more memory than its receive buffer limit. This restriction makes little sense, especially when the

receive buffer is resized frequently based on observed network behavior. If window announcement, a flow control device, is separated from memory allocation, correct window announcement becomes relatively easy.

In the modified implementation, *rcv_alloc* is the amount of data (not including headers or overhead) buffered for delivery to the application, and *ofc_alloc* is the amount of data in the out-of-order queue. The target receive window will be $rcv_space - rcv_alloc + ofc_alloc$. This target value is then checked so it may not retract a previously advertised window, scaled by the negotiated window scale, then announced.

6.7 Separating the retransmit queue

Conceptually, separation is fairly simple. At the point at which a segment is transmitted for the first time, its size is subtracted from *wmem_queued*, and the writing process may be notified of free space if it is waiting. Linux’s auto-tuning of *sndbuf* is disabled, and it remains a fixed size.

6.8 Implementation behavior

In Figures 1 and 2, we see time-sequence graphs collected at the sender using `tcpdump` [9] and graphed with `tcptrace` [10]. Both the sender and receiver are using the modified kernel. In Figure 1, we see the receiver doubling its announced receive window each round-trip time while the sender increases its transmission rate, then remaining constant (at twice the BDP) once the sender is limited by the link bandwidth. In Figure 2, we see the receiving application stop reading data for approximately one second, and shrinks the window to zero. When it resumes, we see that *rcv_space* has shrunk, but grows again until it is an appropriate size.

7 Test Environment

Testing these modifications across a path requires machines running custom kernels on both ends of that path. Testing high-BDP paths would require these end-points to be in distant locations, attached to low-loss, high-bandwidth networks. Establishing such a test bed can be difficult administratively and politically.

Since doing tests across a “true” network is problematic, the Linux kernel was modified to be able to artificially delay the output of IP packets by a specified amount of time. When in effect, this increases the delay while the bandwidth remains the same. It becomes possible to simulate the conditions seen on a “long fat pipe” over a LAN.

7.1 Variable delay

Modifying a path’s RTT is non-trivial. Most components of the delay are dependent on the physical network or on network congestion. However, we can add

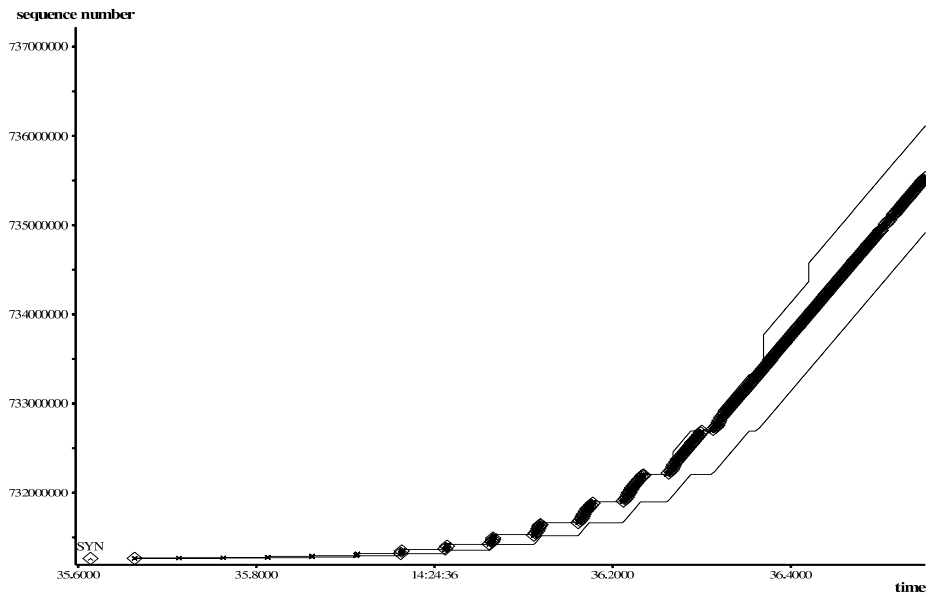


Figure 1: Time-sequence graph of the modified TCP running

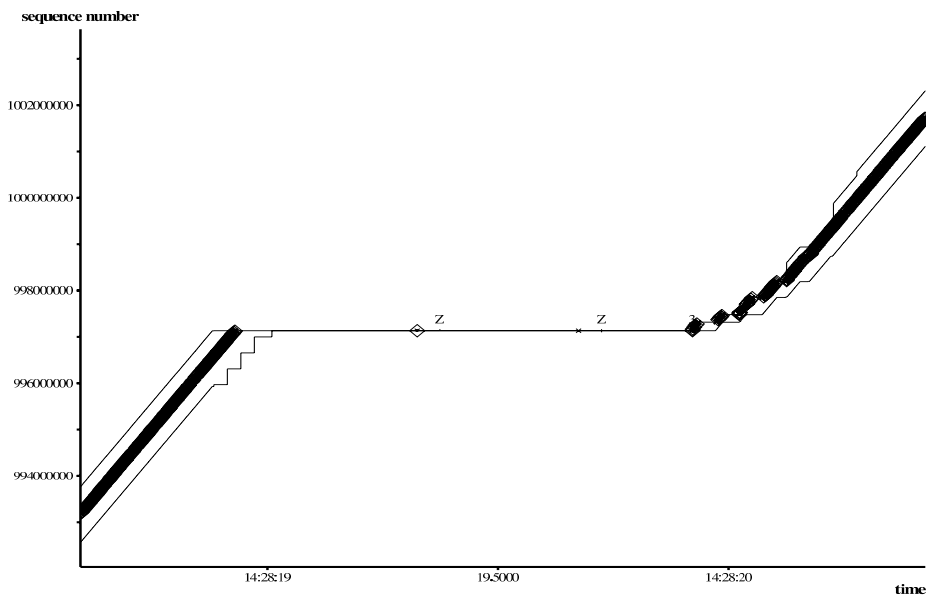


Figure 2: Application stops receiving then resumes

to the effective time spent processing packets by adding a timer delay between the network protocol and the routing queue.

A variable delayer was added to the experimental kernel. It has an interface through the Linux “proc” filesystem, where a user may specify target hosts to which packets will be delayed, and for how many milliseconds to delay those packets. It intercepts packets as they are about to be put into an outbound routing queue, and delays them for the specified interval.

Setting a timer for each packet would be very costly and may cause reordering. Instead, an array of delay queues is created. The array is scanned, one step per clock tick, and the packets in the current delay queue are moved into their target routing queues. To delay a packet by n clock ticks, it is placed in the delay queue n steps ahead of the current scan position in the array.

It should be noted that delaying packets in this manner had some drawbacks. Linux has a fairly coarse system clock compared with network events (100 ticks per second on Intel, 1024 ticks per second on DEC Alpha). The effects of this coarseness should be noted. Packet delays will vary by up to one clock tick. Small packets such as TCP ACK packets may be “compressed.” For example, if ten ACK packets are output by TCP each 1/10th of a clock tick apart, they will be put in the same delay queue, and output to the network in a burst, likely significantly shorter than one clock tick. Such compression may increase TCP’s burstiness, which may cause loss and reduced throughput.

7.2 Bottleneck routing queue size

While in the slow-start congestion control state (at the beginning of a connection), the congestion window will double every round trip. The manner in which TCP self-clocks will cause twice as much data to be transmitted by TCP (put into the local routing queue) over a period of time smaller than RTT as the bottleneck link can send. During each successive round trip, a queue at the bottleneck will be created and drained twice the size of the round trip prior. If this flow is the only one congesting the bottleneck, this occurs until either the time taken for the bottleneck to transmit the window of data becomes greater than RTT, or the router’s queue overflows and it drops packets, forcing the connection into recovery. The former will happen when the path’s BDP is less than the bottleneck router’s maximum queue size; the latter will happen when it is greater. In the first case, a persistent queue will form and continue growing until congestion is signaled to the TCP (by dropping packets or using ECN). The TCP will then recover if necessary and *cwnd* will be a good estimate of BDP. In the second case, however, *cwnd* will be smaller than the BDP, and the connection will spend a lot of time in the congestion avoidance state slowly ramping up *cwnd*.

Further, while in congestion avoidance, *cwnd* will not be able to grow enough to maintain full link bandwidth. Since *cwnd* is incremented by one MSS per round trip and is halved when a congestion event occurs, a TCP flow may use on average only 3/4 of available bandwidth if the router is significantly under-buffered.

For all tests, the sender’s local interface was the bottleneck. Adaptive routing algorithms for dealing with this problem exist [8]; however, a simple drop-tail algorithm was used. The default interface queue size was increased from 100 to 2000 packets.

8 Test Results

Performance was measured by running a data sink server application on one host, and running a data source application on another. The sink application measures the duration and the total data received on each connection, and reports these figures. The source application sends data as quickly as it can for a user-specified amount of time. The data sent is pseudo-randomized in order to nullify potential lower layer data compression algorithms (although none were present in the testing environment used), and generated prior to the connection so that the CPU is not a bottleneck. These applications were custom written, but were loosely based on `trafficsnd` and `trafficrcv` [11].

The RTT varied from < 1 ms to 200 ms, which is approximately the range of propagation delay seen on terrestrial land lines. (A propagation delay of about 70 ms is common for endpoints on opposite U.S. coasts.)

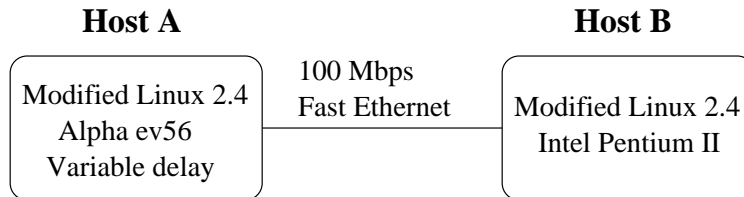


Figure 3: Variable delay testing environment

As noted in Figure 3, the two hosts used for testing were directly connected via full-duplex Fast Ethernet. The sink was run on Host A (the Alpha) and the source on Host B (the Intel). Packet delaying was done using Host A since it has a finer-grained system clock, which minimized the delayer’s bursting effect.

In Figure 4, we see the average throughputs of default and manually-tuned send buffer configurations of the standard Linux 2.4 and the modified implementation with different RTTs. Figure 5 shows performance of default and manually-tuned receive buffer configurations as well as the modified implementation. All data points are the average throughput of an approximately 30-second connection.

The modified implementation performed well in all tested cases. The average throughput decreases with RTT because ramping up the window (doubling per RTT) necessarily takes longer. Maximum throughput achieved, though not measured, should be approximately equal for all cases.

We can clearly see the performance drop in the standard implementation when the buffer space is inadequate. It should be noted that the BDP at

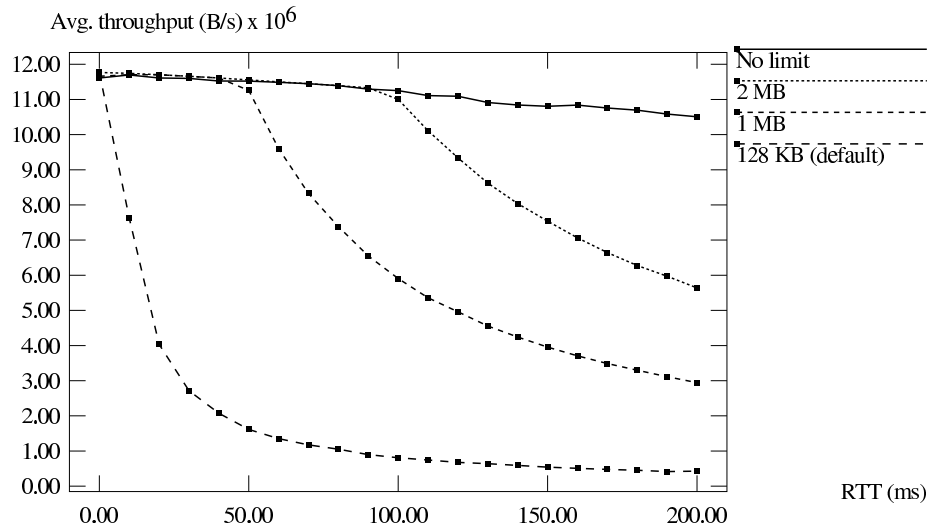


Figure 4: Effect of send buffer size on throughput at different BDPs

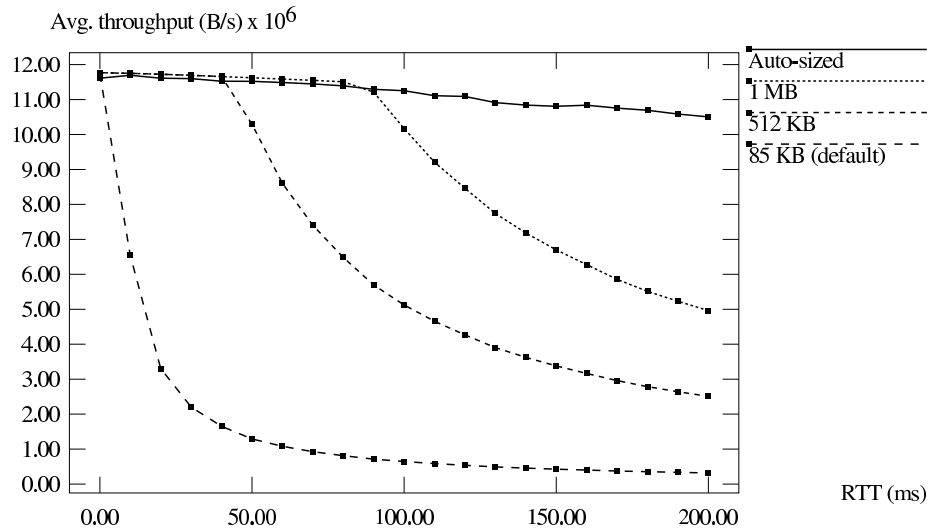


Figure 5: Effect of receive buffer size on throughput at different BDPs

which similarly-sized send and receive buffers limit the transfer is substantially different (about a factor of two) because of the nature of the congestion control mechanism [4], and also because of differing amounts of overhead space.

8.1 Further tests

Some testing was done on a “real” high-BDP path. The two end hosts, one in Pittsburgh and one in Washington, D.C., had GigE interfaces and were connected through an OC-48 network with a delay of approximately 10 ms. The bottleneck in this path should be the sender’s local GigE interface.

However, testing was somewhat problematic. The congestion control mechanism does not work properly on this path, likely because of the high amount of packet reordering caused by parallel routers. The average throughputs observed varied significantly on this path under all buffer configurations. Also, access to the Washington host was restricted to that of a normal user, so the modified kernel was not used, and the interface queue size could not be changed.

	<i>standard</i>	<i>modified</i>
<i>Pittsburgh sender</i>	39.7	42.7
<i>Washington sender</i>	28.7	30.2

Table 1: average throughputs, in MB/sec, over a real network

Table 1 shows the average throughput achieved over a 20-second connection with each of four configurations. The Pittsburgh host was the only one which ran the modified kernel. In the standard configurations, buffers were set very large by the applications. The reduced throughput when the Washington host was sending is likely due to the small interface queue.

It seems likely that in all four configurations, the congestion window rather than the buffers limited the transfer. It seems unlikely that the modified configurations performed better due to the modifications, but that it was randomness of the network interacting with the congestion control mechanism.

9 Conclusion

As networks continue to increase in bandwidth, the queue sizes TCP requires to take advantage of that bandwidth will continue to increase correspondingly. Correctly allocating space for these queues is vital to the scalability and future utility of TCP.

Methods of TCP queue allocation for both sender and receiver queues were proposed and implemented. These methods behave well with arbitrary bandwidth-delay products, and provide a theoretically attractive and practical separation of network and operating system properties in the allocations. Experimental evidence was provided demonstrating the scalable behavior of the implementation at BDPs likely observable today. Future work will include making these algorithms more robust to all network conditions, including deliberate attacks.

10 Acknowledgments

Thanks go to Matt Mathis for many insightful discussions and comments, the Pittsburgh Supercomputing Center for the use of its facilities during development and testing, and Peter Steenkiste for advising on this project.

References

- [1] J. Postel, “RFC 793: Transmission Control Protocol,” September 1981.
- [2] V. Jacobson, “Congestion Avoidance and Control,” *Computer Communication Review*, vol. 18, no. 4, pp. 314-329, August 1988.
- [3] W. Richard Stevens, *TCP/IP Illustrated, Volume 1*, Addison-Wesley, 1994.
- [4] J. Semke, J. Mahdavi, M. Mathis, “Automatic TCP Buffer Tuning,” *ACM SIGCOMM '98*, vol. 28, no. 4, October 1998.
- [5] M. Fisk, W. Feng, “Dynamic Right-Sizing in TCP,” <http://www.lanl.gov/-radiant/website/pubs/drs/lacsi2001.ps>, October 2001.
- [6] V. Jacobson, R. Braden, D. Borman, “RFC 1323: TCP Extensions for High Performance,” May 1992.
- [7] M. Allman, S. Floyd, S. Partridge, “RFC 2414: Increasing TCP’s Initial Window,” September 1998.
- [8] V. Jacobson, K. Nichols, K. Poduri, “RED in a Different Light (Draft),” http://www.cnaf.infn.it/~ferrari/papers/ispn/red_light_9.30.pdf, September 1999.
- [9] “tcpdump,” <http://www.tcpdump.org/>.
- [10] S. Osterman, “tcptrace,” <http://www.tcptrace.org/>.
- [11] “A preconfigured TCP testrig,” <http://ncne.nlanr.net/research/tcp/-testrig/>.