TCP Congestion Window Validation

Mark Handley, Jitendra Padhye, Sally Floyd
mjh@aciri.org, padhye@aciri.org, floyd@aciri.org

September 23, 1999

Abstract

TCP's congestion window controls the number of packets a TCP flow may have in the network at any time. However, long periods when the sender is idle or application-limited can lead to the invalidation of the congestion window, in that the congestion window no longer reflects current information about the state of the network. In this paper we propose a simple modification to TCP's congestion control algorithms to decay the congestion window *cwnd* after the transition from a sufficiently-long application-limited period, while using the slow-start threshold *ssthresh* to save information about the previous value of the congestion window.

An invalid congestion window also results when the congestion window is increased (i.e., in TCP's slow-start or congestion avoidance phases) during application-limited periods, when the previous value of the congestion window might never have been fully utilized. We propose that the TCP sender should not increase the congestion window when the TCP sender has been application-limited (and therefore has not fully used the current congestion window). This note illustrates these algorithms both with simulations and with experiments from an implementation in FreeBSD.

1 Introduction

TCP's congestion window controls the number of packets a TCP flow may have in the network at any time. The congestion window is set using an Additive-Increase, Multiplicative-Decrease (AIMD) mechanism that probes for available bandwidth, dynamically adapting to changing network conditions. This AIMD mechanism works well when the sender continually has data to send, as is typically the case for TCP used for bulk-data transfer. In contrast, for TCP used with telnet applications, the data sender often has little or no data to send, and the sending rate is often determined by the rate at which data is generated by the user. With the advent of the web, including developments such as TCP senders with dynamically-created data and HTTP 1.1 with persistent-connection TCP, the interaction between application-limited periods (when the sender sends less than is allowed by the congestion or receiver windows) and network-limited periods (when the sender is limited by the TCP window[1]) becomes increasingly important.

Long periods when the sender is application-limited can lead to the invalidation of the congestion window. During periods when the TCP sender is network-limited, the value of the congestion window is repeatedly "revalidated" by the successful transmission of a window of data without loss. When the TCP sender is network-limited, there is an incoming stream of acknowledgements that "clocks out" new data, giving concrete evidence of recent available bandwidth in the network. In contrast, during periods when the TCP

[1] We define a network-limited period as any period when the sender is sending a full window of data.

sender is application-limited, the estimate of available capacity represented by the congestion window may become steadily less accurate over time. In particular, capacity that had once been used by the network-limited connection might now be used by other traffic.

Current TCP implementations have a range of behaviors for starting up after an idle period. Some current TCP implementations slow-start after an idle period longer than the RTO estimate, as suggested in the appendix of [3], while other implementations don't reduce their congestion window after an idle period. An alternate proposal for TCP's slow-start after idle has been discussed in [4].

To address the revalidation of the congestion window after a application-limited period, we propose a simple modification to TCP's congestion control algorithms to decay the congestion window *cwnd* after the transition from a sufficiently-long application-limited period (i.e., at least one roundtrip time) to a network-limited period. When the congestion window is reduced, the slow-start threshold *ssthresh* remains as "memory" of the recent congestion window. Specifically, *ssthresh* is never decreased when *cwnd* is reduced after a application-limited period; before *cwnd* is reduced, *ssthresh* is set to the maximum of its current value and 3/4 *cwnd*. Thus, a TCP sender increasing its sending rate after a application-limited period can slow-start back up to recover most of the previous value of the congestion window.

An invalid congestion window also results when the congestion window is increased (i.e., in TCP's slow-start or congestion avoidance phases) during application-limited periods, when the previous value of the congestion window might never have been fully utilized. As far as we know, all current TCP implementations increase the congestion window when an acknowledgement arrives, if allowed by the receiver's advertised window and the slow-start or congestion avoidance window increase algorithm, without checking to see if the previous value of the congestion window has in fact been used. This draft proposes that the window increase algorithm not be invoked during application-limited periods. In particular, the TCP sender should not increase the congestion window when the TCP sender has been application-limited (and therefore has not fully used the current congestion window). This restriction prevents the congestion window from growing arbitrarily large, in the absence of evidence that the congestion window can be supported by the network.

The issue of validation of congestion information during idle periods has also been addressed in contexts other than TCP and IP, for example in "Use-it or Lose-it" mechanisms for ATM networks [5, 6].

A somewhat-orthogonal problem associated with maintaining a large congestion window after an application-limited period is that the sender, with a sudden large amount of data to send after a quiescent period, might immediately send a full congestion window of back-to-back packets. This problem of sending large bursts of packets back-to-back can be effectively handled using rate-based pacing (RBP, [1]), or using a maximum burst size control [2]. We would contend that, even with mechanisms for limiting the sending of back-to-back packets, an old congestion window that has not been fully used for some time can not be trusted as an indication of the bandwidth currently available for that flow. Thus, we would contend that the mechanisms to pace out packets allowed by the congestion window are largely orthogonal to the algorithms used to determine the appropriate size of the congestion window.

2 Description

When a TCP sender has sufficient data available to fill the available network capacity for that flow, *cwnd* and *ssthresh* get set to appropriate values for the network conditions. When a TCP sender stops sending, the flow stops sampling the network conditions, and so the value of the congestion window may become inaccurate.

We believe the correct conservative behavior under these circumstances is to decay the congestion window by half for every RTT that the flow remains inactive. The value of half is a very conservative figure based on how quickly multiplicative decrease would have decayed the window in the presence of loss.

Another possibility is that the sender may not stop sending, but may become application-limited rather than network-limited, and offer less data to the network than the congestion window allows to be sent. In this case the TCP flow is still sampling network conditions, but is not offering sufficient traffic to be sure that there is still sufficient capacity in the network for that flow to send a full congestion window. Under these circumstances we believe the correct conservative behavior is for the sender to keep track of the maximum amount of the congestion window used during each RTT, and to decay the congestion window each RTT to midway between the current *cwnd* value and the maximum value used.

Before the congestion window is reduced, *ssthresh* is set to the maximum of its current value and 3/4 *cwnd*. If the sender then has more data to send than the decayed *cwnd* allows, the TCP will slow-start (perform exponential increase) until the *cwnd* reaches the old value of *ssthresh*.

For the separate issue of the increase of the congestion window in response to an acknowledgement, we believe the correct behavior is for the sender to increase the congestion window only if the window was full when the acknowledgment arrived.

We term this set of modifications to TCP *Congestion Window Validation* (CWV) because they are related to ensuring the congestion window is always a valid reflection of the current network state *as probed by the connection*.

2.1 The basic algorithm for reducing the congestion window

A key issue in the algorithm for reducing the congestion window is to determine how to apply the guideline of reducing the congestion window once for every roundtrip time that the flow is application-limited. We use TCP's retransmission timer (RTO) as a reasonable upper bound on the roundtrip time, and reduce the congestion window once per RTO.

This basic algorithm could be implemented in TCP as follows: After TCP sends a packet, it checks to see if that packet filled the congestion window. If so, the sender is network-limited, and sets the variable T_{prev} to the current TCP clock time, and a variable W_{used} to zero. T_{prev} will be used to determine the elapsed time since the sender last was network-limited. When the sender is application-limited, W_{used} holds the maximum congestion window actually used since the sender was last network-limited.

If the transmitted packet did not fill the congestion window and the TCP send queue is empty, then the sender is application-limited. The sender checks to see if the amount of unacknowledged data is greater than W_{used}; if so, W_{used} is set to the amount of unacknowledged data. In addition TCP checks to see if the elapsed time since T_{prev} is greater than RTO. If so, then the TCP has been application-limited rather than network-limited for an entire RTO interval. In this case, TCP sets *ssthresh* to the maximum of *3/4 cwnd* and the current value of *ssthresh*, and reduces its congestion window to (cwnd+W_{used})/2. W_{used} is then set to zero, T_{prev} is set to the current time, so a further reduction will not take place until another RTO period has elapsed.

After TCP sends a packet, it also sets the variable T_{last} to the current time. When TCP sends a new packet it also checks to see if more than RTO seconds have elapsed since the previous packet was sent. If RTO has elapsed, *ssthresh* is set to the maximum of *3/4 cwnd* and the current value of *ssthresh*, and then the congestion window is halved for every RTO that elapsed since the previous packet was sent. In addition,

T_{prev} is set to the current time, and W_{used} is reset to zero. This last mechanism could also be implemented by using a timer that expires every RTO after the last packet was sent instead of a check per packet - efficiency constraints on different operating systems may dictate which is more efficient to implement.

2.2 Pseudo-code for reducing the congestion window

Initially:

$T_{last} = tcp_now, T_{prev} = tcp_now, W_{used} = 0$

After sending a data segment:

```
If tcp_now − T_last ≥ RTO
    (The sender has been idle.)
    ssthresh = max(ssthresh, 3 ∗ cwnd/4)
    For i = 1 To (tcp_now − T_last)/RTO
        win = min(cwnd, receiver's declared max window)
        cwnd = max(win/2, MSS)
    T_prev = tcp_now
    W_used = 0

T_last = tcp_now

If window is full
    T_prev = tcp_now
    W_used = 0
Else
    If no more data is available to send
        W_used = max(W_used, amount of unacknowledged data)
        If tcp_now − T_prev ≥ RTO
            (The sender has been application-limited.)
            ssthresh = max(ssthresh, 3 ∗ cwnd/4)
            win = min(cwnd, receiver's declared max window)
            cwnd = (win + W_used)/2
            T_prev = tcp_now
            W_used = 0
```

3 Simulations

3.1 Simulations of window reduction after an application-limited period

The CWV proposal has been implemented as an option in NS, and the first set of simulations shows the use of CWV to reduce the congestion window after a period when the TCP connection was application-limited.

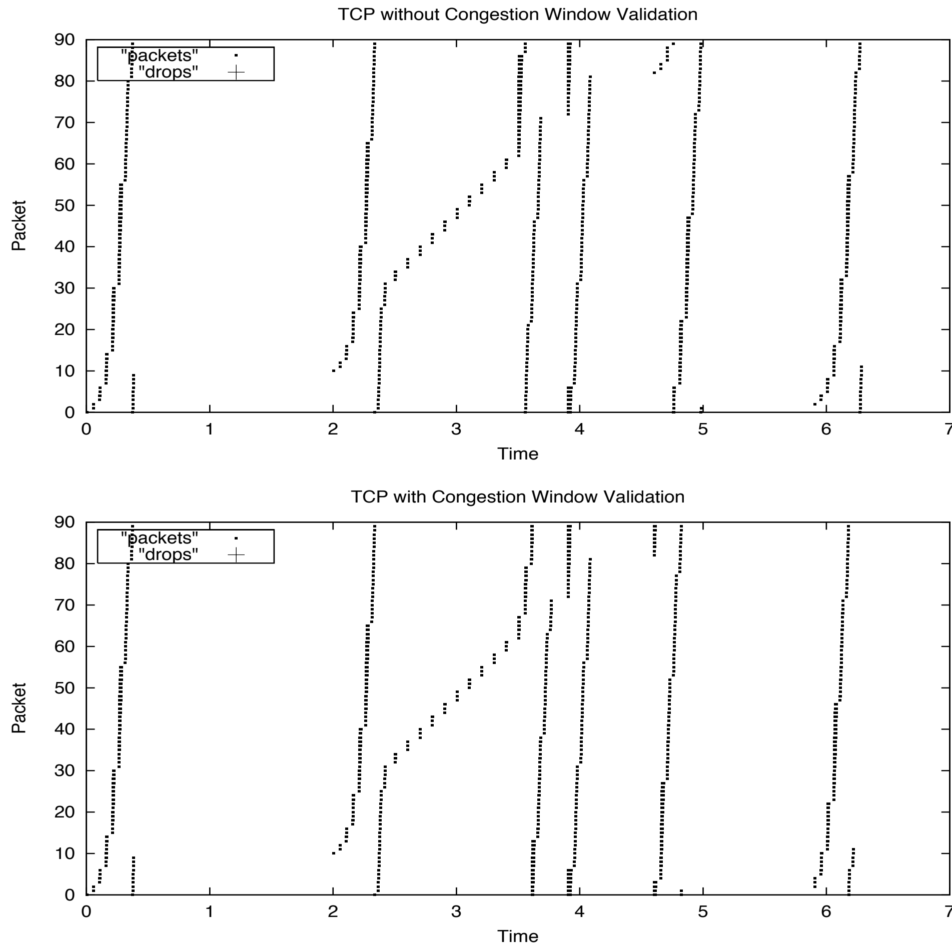

Figure 1: Simulations with and without Congestion Window Validation

The top graph of Figure 1 shows the standard TCP in NS, and the bottom graph shows TCP with the CWV modification. The round trip time in this simulation is 20 ms, and the sending rate is limited by the receiver window rather than network congestion to illustrate the effects more clearly. Both TCPs have a TCP clock tick of 100 ms.

With the regular TCP implementation in NS, the sender slow-starts if the connection has been idle (in both directions) for more than an RTO. This slow-start can be seen in the top graph[2] of Figure 1 at times 2, 4.5

[2]The simulation in the top graph of Figure 1 can be run in NS in the directory tcl/test with the command "ns test-suite-tcp.tcl quiescent_100ms", and the simulation in the bottom graph can be run in NS with the command "ns test-suite-tcp.tcl qui-

and 5.8. Like the non-CWV TCP, the CWV TCP also slow-starts from a one packet congestion window at time 2. However, at times 4.5 and 5.8, the CWV TCP slow-starts from a larger congestion window that that used by the non-CWV TCP, because the connection has not been idle for long enough for the window to decay to one.

At time 2.4 the sender enters an application-limited period, and at time 3.5 the sender again becomes network-limited. The regular TCP in Figure 1 dumps a full window of packets into the network at time 3.5, even though network conditions could have changed since the congestion window was last fully used. In contrast, the CWV TCP has decayed the congestion window by time 3.5. Instead of sending a large burst into the network at time 3.5, the TCP with CWV slow-starts from the window that was actually being used.

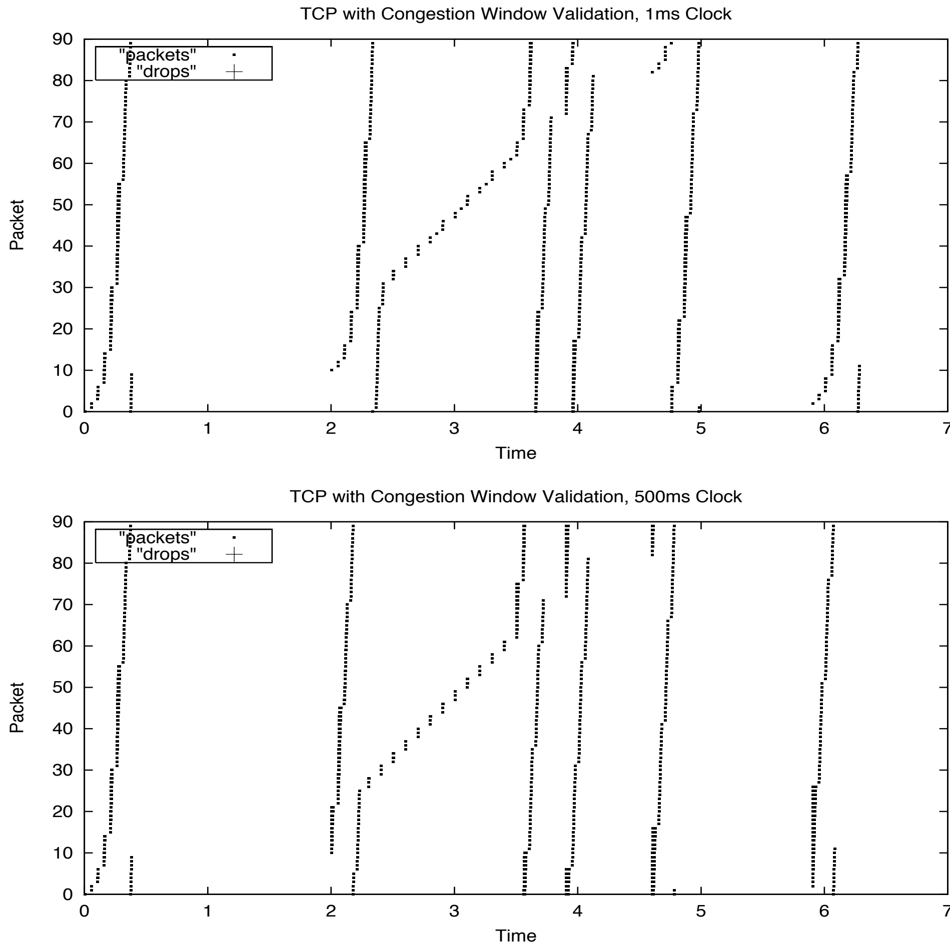

Figure 2: Simulations with Congestion Window Validation with different values for TCP Tick

The two graphs[3] in Figure 2 show the CWV TCP with different values for the TCP clock of 1 ms clock and 500 ms respectively. A large value for the TCP clock results in a large value for the Retransmit TimeOut value RTO, and therefore for a slower decaying of the congestion window after an idle or application-limited

escent_100ms_fine".

[3]The simulation in the top graph in Figure 2 can be run in NS with the command "ns test-suite-tcp.tcl quiescent_1ms_fine", and the simulation in the bottom graph can be run in NS with the command "ns test-suite-tcp.tcl quiescent_500ms_fine".

period. Note that when the 500 ms clock is used, the congestion window at time 3.5 has only decayed part of the way down to the congestion window actually being used.

3.2 Using ssthresh to maintain connection history

The use of ssthresh to maintain connection history is a critical part of the Congestion Window Validation algorithm.

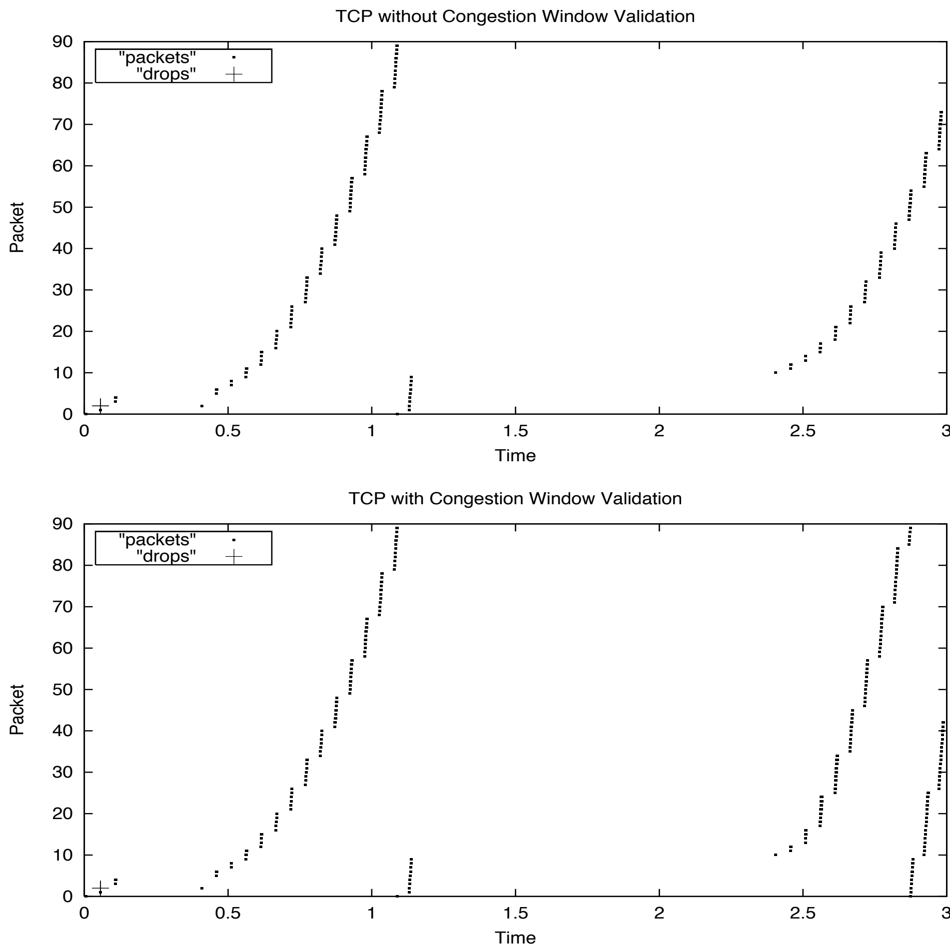

Figure 3: Simulations with and without Congestion Window Validation, with an initial packet drop.

Figure 3 shows two simulations[4], one with and one without CWV, when there is a packet drop early in the connection history. After the packet drop, the slow-start threshold *ssthresh* is set to a very small value. For the non-CWV TCP in the top graph of Figure 3, the congestion window is reset after the first idle period, but *ssthresh* is not increased to represent the congestion window before the idle period. As a result, the TCP at time 2.4 has a very small value of ssthresh when slowstarting, and therefore takes a long time to reopen

[4]The simulation in the top graph of Figure 3 can be run in NS with the command "ns test-suite-tcp.tcl quiescentB", and the simulation in the bottom graph can be run in NS with the command "ns test-suite-tcp.tcl quiescentB_qoption".

its congestion window.

For the TCP with CWV in the lower graph of Figure 3, there is the same small value of *ssthresh* from the initial packet loss, but *ssthresh* was increased along with the decrease of the congestion window after the idle period. As a result, the TCP still slow-starts at time 2.4 from a congestion window of one packet, but because of the larger value of *ssthresh*, the TCP quickly recovers the old value of its congestion window.

3.3 Limiting the congestion window increase

In standard TCP, arriving acknowledgments cause the congestion window to increase. For a transfer that is always network-limited, such as a file transfer, this is not a problem. However, for transfers that are application-limited, this can cause inappropriate increases in the congestion window.

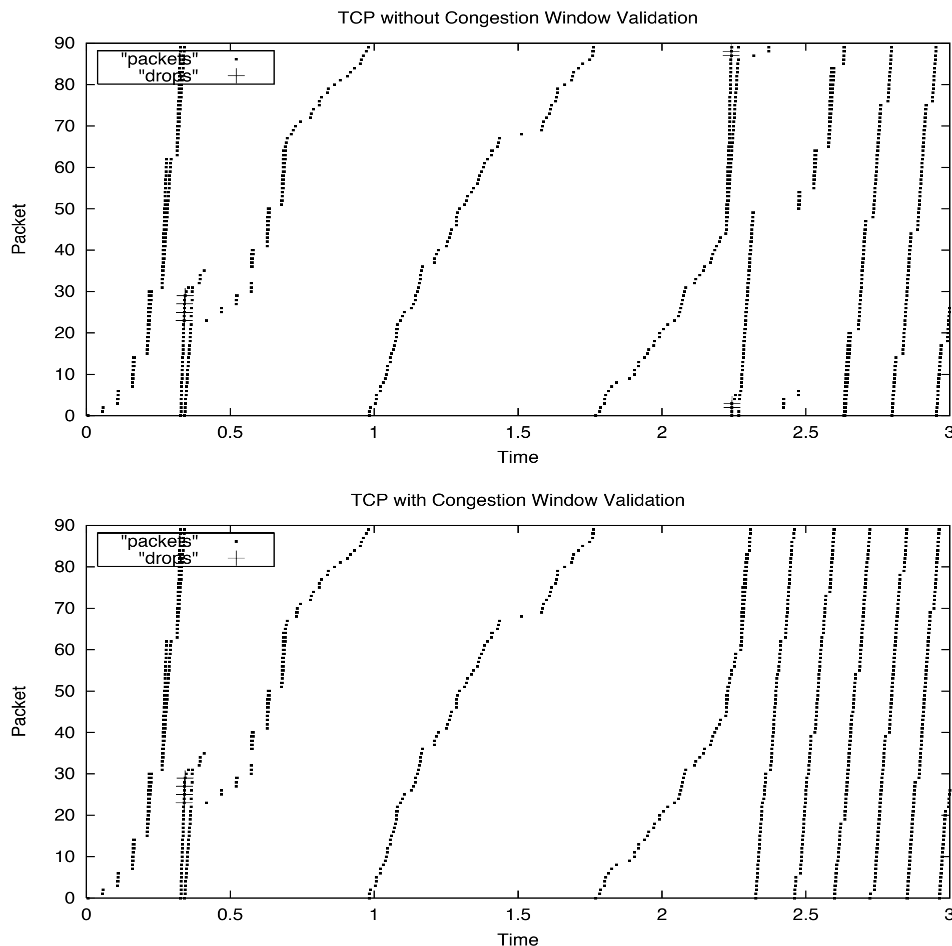

Figure 4: Simulations with and without a limitation on congestion window increase during application-limited periods.

The top graph[5] in Figure 4 illustrates this effect in simulation. The sending TCP is application-limited

[5]The simulation in the top graph of Figure 4 can be run in NS in the directory tcl/test with the command “ns test-suite-tcp.tcl

from time 0.7 to 2.2, and a time 2.2 a large burst of traffic is generated. For the non-CWV TCP, the congestion window continues to increase during the application-limited period. When the large burst of traffic is generated at time 2.2, the non-CWV TCP has a large congestion window, and sends a window of back-to-back packets. However the network does not have sufficient bandwidth for these packets, and many of the packets are dropped. With the CWV TCP (lower graph), the congestion window has not been over-inflated during the application-limited period, and the TCP comes out of the application-limited period with a more appropriate value for the congestion window.

The CWV TCP in the bottom graph of Figure 4 uses two CWV mechanisms, the limitation on the congestion window increase during the application-limited period, and the periodic reduction of the congestion window during the application-limited period. Separate simulations in the NS validation test suite show the effect of each of these two mechanisms.

4 Experimental validation

We have implemented the changes described above in the TCP implementation in FreeBSD 3.2.

Figure 5 shows the effects of the Congestion Window Validation mechanisms for limiting *cwnd* increases during application-limited periods. Both graphs in Figure 5 show the return path of a real ssh connection through a modem link emulated using Dummynet[7]. The link speed is 30Kb/s and the link has five packet buffers available. Today most modem banks have more buffering available than this, but this situation sometimes occurs with older modems. In the first half of both graphs, the user is typing away over the connection (the character echo is shown). About half way through the time, the user lists a moderately large file, which causes a large burst of traffic to be transmitted.

The top graph in Figure 5 shows an unmodified TCP. Every returning ACK has increased *cwnd*, and so the burst is sent as many back-to-back packets, most of which get lost and subsequently retransmitted (marked “R” in the graph).

The bottom graph in Figure 5 shows a modified TCP with Congestion Window Validation. The congestion window has been decreased to be close to what the user used, and is not increased when the window is not full. The burst of traffic is now constrained by the congestion window, resulting in a much better behaved flow with minimal loss. The end result is that the transfer happens approximately 30% faster due to avoiding retransmission timeouts.

Figure 6 shows similar traces of a real ssh connection over a real dialup ppp connection. The top graph shows the unmodified TCP. In this case the modem bank has much more buffering, and the initial burst does not cause loss in this case, but does cause the RTT to increase to approximately 5 seconds, where the connection becomes bounded by the receiver’s window.

The bottom graph of Figure 6 shows the modified TCP with Congestion Window Validation. This flow is much better behaved, and produces no large burst of traffic. In this case linear increase occurs which slowly increases the RTT as the buffer fills.

In this example, both flows finish delivering the data at precisely the same time as can be seen from the lower line which shows the data ACKed. This is because the link has been fully utilized in both cases due to the modem buffer being larger than the receiver window. Clearly a modem buffer of this size is undesirable

underutilized_100ms”, and the simulation in the bottom graph can be run in NS with the command “ns test-suite-tcp.tcl underutilized_100ms_control_Q”.

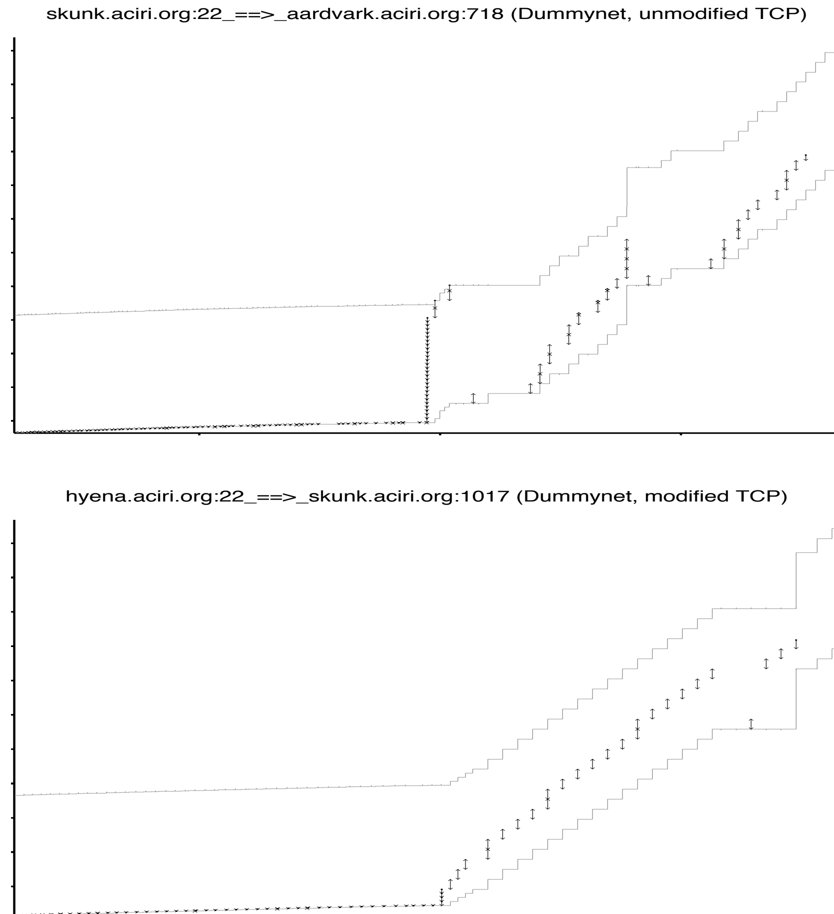

Figure 5: Experiments with FreeBSD TCP over an emulated dialup connection

due to its effect on the RTT of competing flows, but it is necessary with current TCP implementations that produce bursts similar to those shown in the top graph.

Reports on further experiments of these changes will be reported in a future document.

5 Interaction with the Nagle algorithm

The FreeBSD implementation revealed that CWV as described above can interact poorly with the Nagle algorithm. This is because there are checks in both our congestion window increase and decrease code that ensure some actions only occur when the window is full, and under some circumstances the Nagle algorithm can prevent the window from being completely full for significant lengths of time. This problem is easily solved by changing the checks for a full window to be true if the window is less than an MSS below being full. Experimental validation shows this does avoid the problem, and the change does not significantly change the overall behavior of the mechanisms.

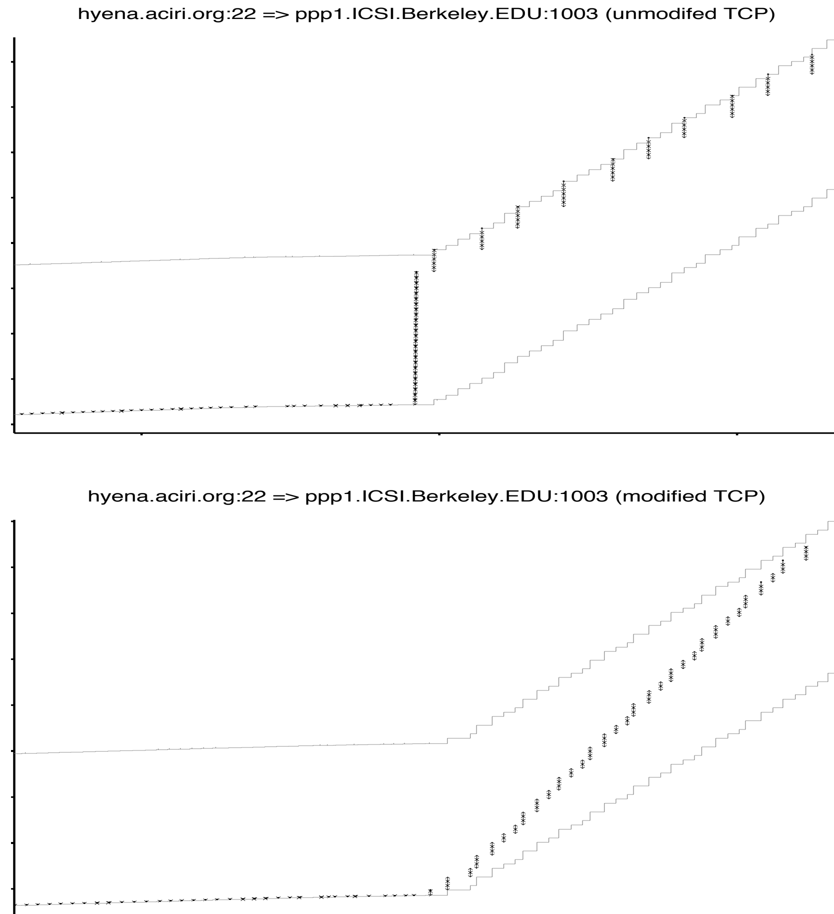

Figure 6: Experiments with FreeBSD TCP over a dialup connection

6 Conclusions

This short paper has presented several TCP algorithms for Congestion Window Validation, to be employed after an idle period or a period in which the sender was application-limited, and before an increase of the congestion window. The goal of these algorithms is for TCP's congestion window to reflect recent knowledge of the TCP connection about the state of the network path, while at the same time keeping some memory (i.e., in *ssthresh*) about the earlier state of the path. We believe that these modifications will be of benefit to both the network and to the TCP flows themselves, by preventing unnecessary packet drops due to the TCP sender's failure to update its information (or lack of information) about current network conditions. Future work will document and investigate the benefit provided by these algorithms, using both simulations and experiments. Additional future work will describe more sophisticated versions of these algorithms for TCP implementations where the sender does not have an accurate estimate of the TCP round-trip time.

References

[1] Vikram Visweswaraiah and John Heidemann. "Improving Restart of Idle TCP Connections", Technical Report 97-661, University of Southern California, November, 1997.

[2] Fall, K., and Floyd, S., Simulation-based Comparisons of Tahoe, Reno, and SACK TCP, Computer Communication Review, V. 26 N. 3, July 1996, pp. 5-21. URL 'http://www.aciri.org/floyd/papers.html'.

[3] Jacobson, V., Congestion Avoidance and Control, Originally from Proceedings of SIGCOMM '88 (Palo Alto, CA, Aug. 1988), and revised in 1992. URL "http://www-nrg.ee.lbl.gov/nrg-papers.html".

[4] Amy Hughes, Joe Touch, John Heidemann, "Issues in TCP Slow-Start Restart After Idle" Work-in-progress. April 1998. URL "ftp://ftp.isi.edu/internet-drafts/draft-ietf-tcpimpl-restart-00.txt".

[5] Raj Jain, Shiv Kalyanaraman, Rohit Goyal, Sonia Fahmy, and Fang Lu, Comments on "Use-it or Lose-it", ATM Forum Document Number: ATM Forum/96-0178, URL 'http://www.netlab.ohio-state.edu/∼jain/atmf/af_rl5b2.htm'.

[6] R.Jain, S.Kalyanaraman, R. Goyal, S.Fahmy, F.Lu, A Fix for Source End System Rule 5, AF-TM 95-1660, December 1995, URL 'http://www.netlab.ohio-state.edu/∼jain/atmf/af_rl52.htm'.

[7] L. Rizzo, "Dummynet - Flexible bandwidth manager and delay emulator", Unix manual page, 1998. URL 'http://www.iet.unipi.it/∼luigi'