TCP Rate-Halving with Bounding Parameters

Matt Mathis
Jamshid Mahdavi

This technical note describes recent enhancements to FACK TCP. It supplements our SIGCOMM'96 Paper "Forward Acknowledgment: Refining TCP Congestion Control" [MM96]. Due to the urgency of distributing this information to the community, we are releasing drafts still under active revision. The final version of this note is expected to appear in some future journal.

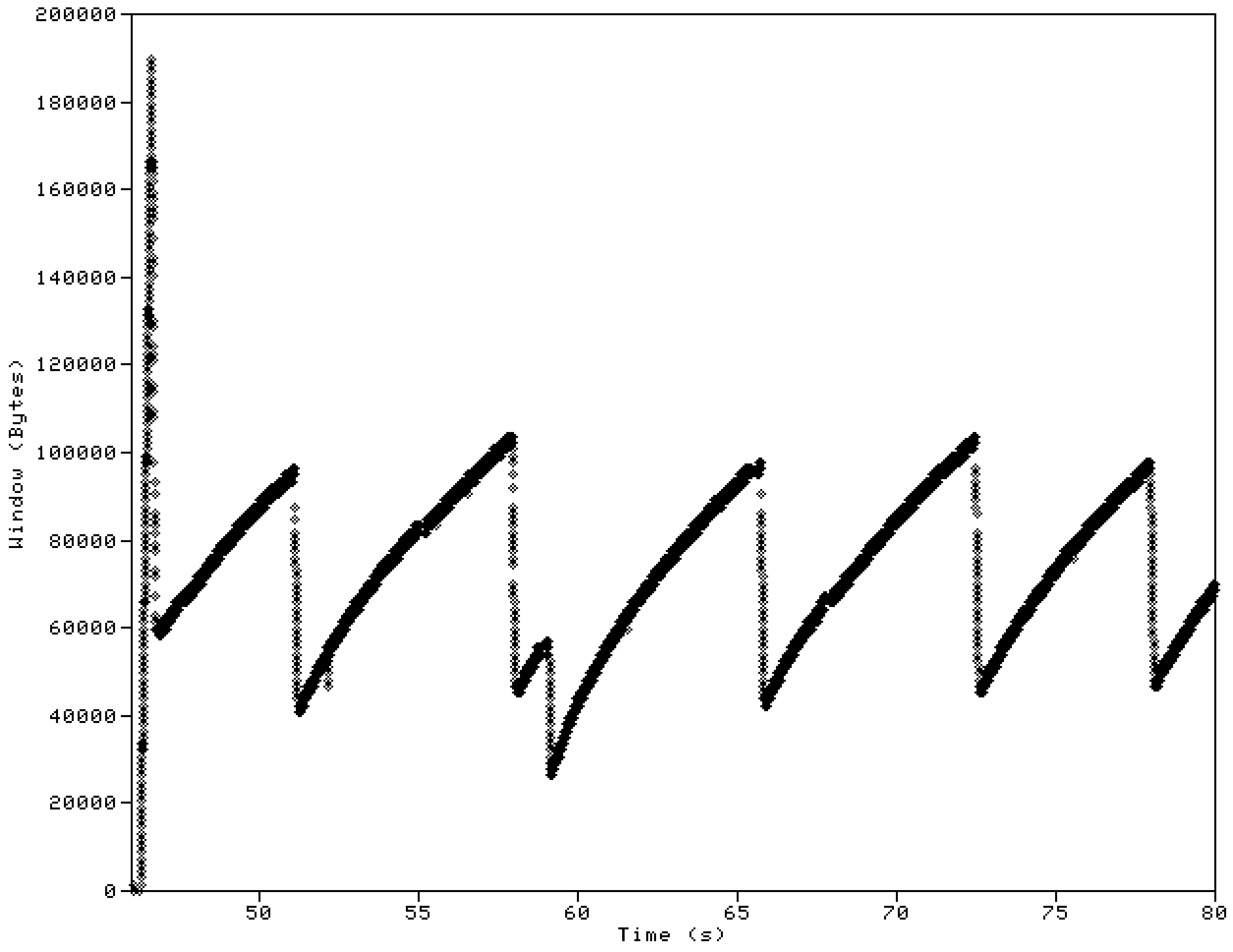
The newest draft will be updated from time to time. Other, older drafts will be preserved and indexed. This draft (which may be the most recent) is dated Fri Dec 19 17:11:16 EST 1997.

Introduction.

The FACK algorithm makes it possible to treat congestion control during recovery in the same fashion as during other parts of the TCP state space. Our work on FACK includes a core FACK algorithm, along with several other ancillary algorithms, all designed to work together. Recent developments have replaced "Overdamping" and "Rampdown" (described in the SIGCOMM paper) by a combined "Rate-Halving" algorithm, which preserves the best properties of each. Rate-Halving attempts to find the correct window size following packet loss (even under adverse conditions), while at the same time maintaining TCP's Self-clock. We further strengthen Rate-Halving with "bounding parameters," in order to assure that Rate-Halving results in an acceptable window, even following many types of pathological network behavior. In addition, we strengthen the retransmission strategy by decoupling it completely from congestion control considerations during recovery. An algorithm we call "Thresholded Retransmission" moves the *tcprexmtthresh* logic to the SACK scoreboard and applies it to every hole (not just the first). We also added "Lost Retransmission Detection" to determine when retransmitted segments have been lost in the network.

All of these algorithms are still under development, but we encourage others to experiment with our research FACK TCP implementation.

Figure 1 is a plot of the actual window size vs. time of a real TCP connection (See the endnotes for a detailed discussion of the test setup and graphs). Its Slow-Start lacks the large discontinuities and curious artifacts typically seen following Reno initial Slow-Start. Furthermore all of the recovery intervals, including the one following the massive losses at the end of Slow-Start, arrive at appropriate congestion windows. (Janie Hoe [Ho96] has investigated Reno Slow-Start behavior and suggested some improvements. In addition, she has suggested the use of Rate-Halving during recovery [Ho95].)


Window (Bytes)
Time (s)
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
200000
50
55
60
65
70
75
80

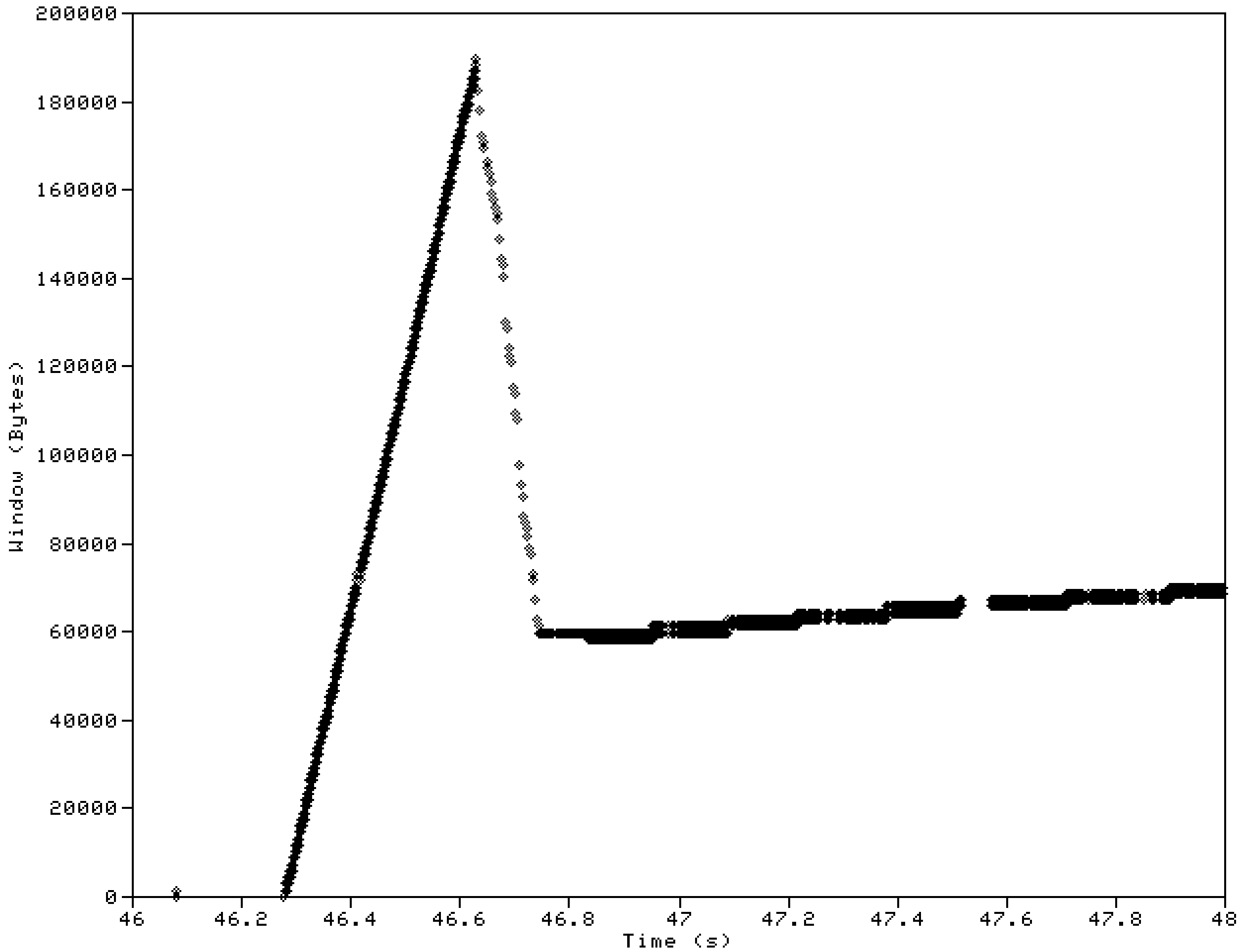

Window (Bytes)
Time (s)
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
200000
46
46.2
46.4
46.6
46.8
47
47.2
47.4
47.6
47.8
48

Figure 1: Congestion Window vs. Time.

Figure 2 is a detailed plot of sequence number vs time for the 5th recovery episode in Figure 1. There are 6 retransmissions, clearly visible one round trip time to the right of the trace. Since this path has more than sufficient buffering, the data rates (slope) before and after recovery are the same. During recovery, there is exactly one round trip at half rate, which reduces the congestion window by half.

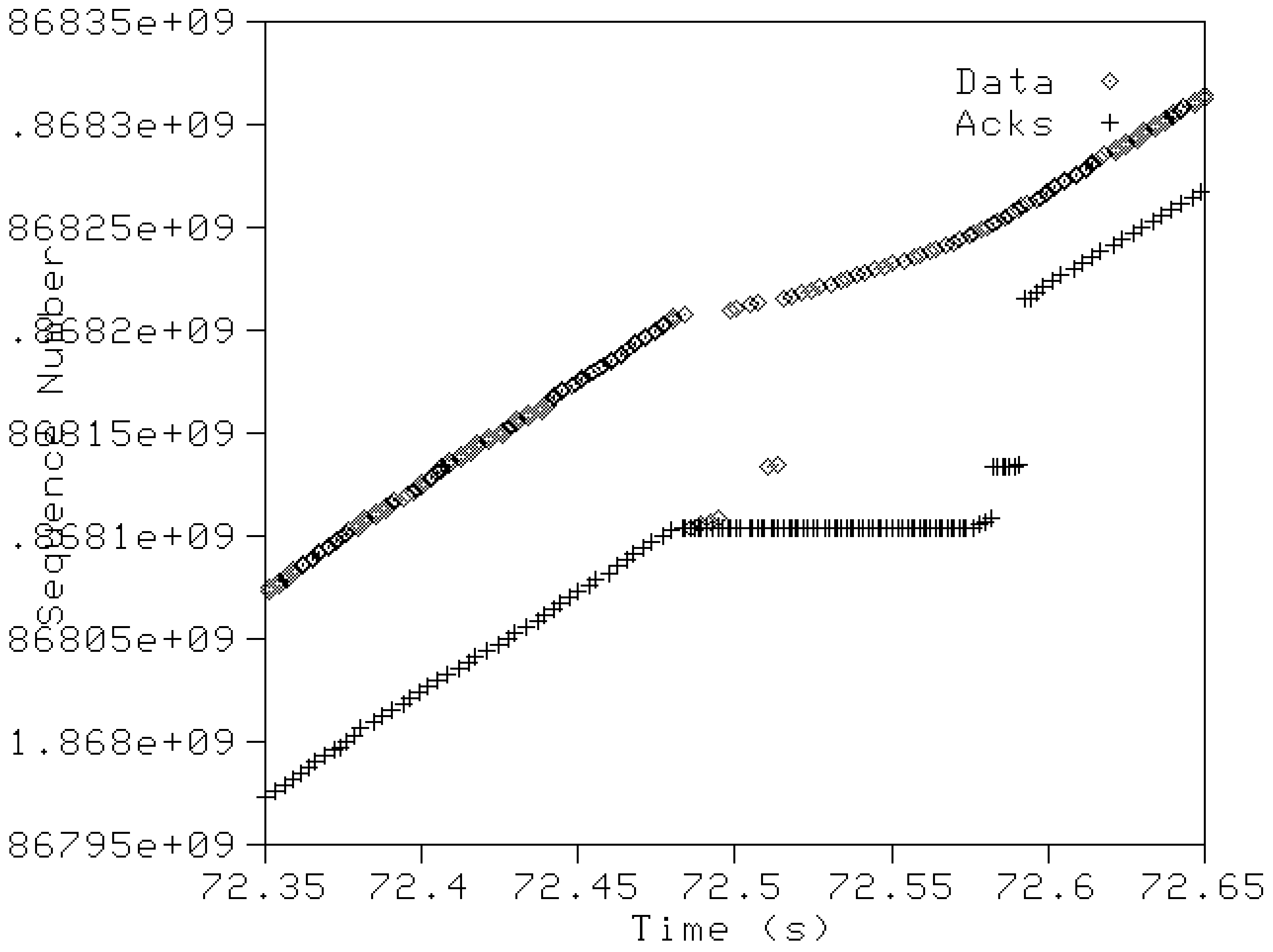


Figure 2: Data Sequence Number vs. Time.

SACK Processing, the Scoreboard and Data Recovery.

SACK options are parsed and recorded in a "scoreboard" in the manner suggested by RFC 2018 [MMFR96]. We have added two algorithms to the scoreboard.

The first algorithm, Thresholded Retransmission, is implemented by associating a counter with each hole (missing data) in the scoreboard. This counter is incremented every time a SACK is parsed that reports any data to the right of the hole. Each hole becomes eligible for repair only when its associated counter reaches *tcprexmtthresh*, (which we set to 3).

Second, we detect when retransmitted segments have left the network with a Lost Retransmission Detection Algorithm. We implement a transmission strategy that nearly always transmits segments in ascending order. This is natural, because holes are discovered in ascending order (and new data is always sent in ascending order). The common exception is when old data is retransmitted after some new data. Each hole is tagged with the sequence number of the most recently sent new data (i.e. at the time of retransmission, the hole is tagged with the value of *snd_max*). If a hole is still unfilled and *snd_fack* (the highest ACKed or SACKed data) advances beyond the saved *snd_max* then later new data has arrived, and the retransmission must have been lost. Our current implementation, in order to be conservative, forces a timeout upon loss of a retransmitted segment.

Organization of the Code

Before we discuss the congestion control mechanisms we need to describe how the functions have been reorganized in the code.

Tcp_input() parses SACK options and updates the scoreboard. All congestion window (*snd_cwnd*) adjustments including Slow-Start, Congestion Avoidance and Rate-Halving are together in the same section of the code. *Tcp_input()* does not overload *snd_cwnd* during the recovery phase, monkey with *snd_nxt* or force a Fast Retransmission through contrived calls to *tcp_output()*. Rate-Halving (which replaces Fast Recovery) starts immediately when a SACK is detected.

Tcp_output() queries the scoreboard to determine what data should be sent. During Rate-Halving, the scoreboard initially chooses not to retransmit data. *Tcp_output()* therefore transmits new data (albeit at the reduced rate), and only retransmits segments when the scoreboard determines that *tcprexmtthresh* has been reached for those segments.

The signals between the algorithms are limited to only 3 flag bits: one indicating that recovery is in progress, one indicating that Rate-Halving is in effect, and the last indicating that there has been a retransmission. The Rate-Halving and Recovery flags are separated to allow Tahoe-style recovery under extreme circumstances. These signals can easily be reconstructed from a packet trace, and their simplicity creates an opportunity for future conformance testing.

Rate-Halving Congestion Control.

Rate-Halving Congestion Control adjusts the window by sending one segment per two acknowledgments for exactly one round trip. This sets the new window to exactly one half of the data which was actually held in the network during the congested round trip. At the beginning of the congested round trip we have injected cwnd segments into the network. Given that there have been some losses, we expect to receive (*snd_cwnd-loss*) acknowledgments. Under Rate-Halving we send half as many segments, so the net effect on the congestion window will be:

(Eq. 1)

$$snd\_cwnd = (snd\_cwnd - loss)/2$$

This (together with Slow-Start and Congestion Avoidance) has the behavior as shown in Figure 1. Rate-Halving finds the correct window following the Slow-Start because the Slow-Start experienced nearly 30% losses making the net window adjustment roughly a factor of 3.

Note that Equation 1 is more conservative than Van Jacobson's 1988 paper [Ja88]: rather than setting the window to 1/2 of the outstanding data, we set it to 1/2 of the data that was actually held in the network.

We detect when exactly one round trip has elapsed by comparing the current value of *snd_fack* to the value of *snd_nxt* saved when the first SACK block arrived. Also, at the end of recovery *ssthresh* is set to *snd_cwnd*.

Bounding Parameters

Rate-Halving with bounding parameters adds additional controls to guarantee that the final window is appropriate, in spite of unexpected network behaviors. The first change is the addition of a window Hold State which follows Rate-Halving. The Hold State prevents the window from increasing until data recovery is complete, since data recovery always takes longer than the one round trip used by Rate-Halving. The earliest that the data recovery can be completed is one round trip plus roughly *tcprexmtthresh/2* acknowledgments after a single hole. If there are additional holes, the Hold State lasts until the data receiver indicates that there are no outstanding holes (e.g. sends an ACK with no SACK blocks.)

During the Hold State each ACK causes precisely one segment to be transmitted. If there are additional losses, they reduce the number of outstanding ACKs, causing the window to drop by the amount of data lost (and further postponing the end of the Hold State). TCP will remain in the Hold State until there has been one full round trip with zero losses. If congestion persists for more than 1 round trip (e.g. due to rerouting onto a skinny link) the window will be (more than) halved during the first round trip and will continue to drop during the Hold State.

This guarantees that the congestion window at the end of recovery actually fits into the network. Define *RHloss* and *HSloss* to be the losses during the Rate-Halving and Hold States respectively. The final window adjustment is then:

(Eq. 2)

$$cwnd = (cwnd - RHloss)/2 - HSloss$$

The bounding parameters also constrain other aspects of the recovery. These are not important under normal congestion but offset some pathologies observed in real networks. Part of our unfinished work is tinkering with the details of the bounding parameters and verifying them under as many environments as possible. Currently we implement 3 bounding parameters beyond the basic Hold State:

- The final window must never be larger than *snd_cwnd/2* (if there was a retransmission). This explicit check should be redundant.
- If the window falls below *snd_cwnd/4* (or *2*MSS*) during Rate-Halving we enter the Hold State early.
- We set *ssthresh* to *snd_cwnd/4* (but not less than *2*MSS*) so that if cwnd does fall too far, TCP will reopen the window quickly after the end of recovery.

We are considering other bounding parameters as well. Again, under normal conditions none of these has any effect on TCP behavior. All are second order corrections applicable only under limited conditions.

The Behavior of Rate-Halving with Bounding Parameters.

Note: At this time, some details of this discussion are theoretical. Portions of this section are based on an analysis of the algorithms, and have not been confirmed with direct observations

- Retransmissions triggered by SACK options all start out in the same way: the first ACK with a SACK triggers the transmission of one segment of new data. The second causes no transmissions, thereby reducing the window (assuming the window is large enough to stay out of the Hold State). The third ACK with a SACK will trigger the retransmission of the first missing segment. (The divide by two in Equations 1 and 2 is implemented with a toggle bit, which is initialized such that Equation 2 is rounded up to a multiple of the MSS.)
- Slightly out of order segments will invoke Rate-Halving, which will always reduce *snd_cwnd* by at least some fraction of a MSS. If the segment arrives just prior to reaching *tcprexmtthresh*, Rate-Halving will have pulled *snd_cwnd* down by at least 1 full MSS. If the segment arrives even later and a retransmission has already occurred, we make the full *snd_cwnd*/2 window adjustment mandatory, because otherwise there exist conditions where TCP will perpetually waste bandwidth with spurious retransmissions.
- We are working on a new version of the TReno tool (see IP Provider Metrics), which implements precisely the same congestion behavior as Rate-Halving TCP. This tool has been instrumental in testing and understanding congestion dynamics. (The new version of TReno is not ready yet.)
- If cwnd is two segments, one drop does not cause a time out. The segment following the dropped segment triggers a SACK block which causes the data sender to adjust cwnd down to one. Since this is too small, the sender immediately enters the Hold State, and sends more new data. This causes another SACK and still more new data, until the sender acquires *tcprexmtthresh* duplicate ACKs, at which point it retransmits the missing segment. We have validated that TReno performs as expected when *tcprexmtthresh* is much larger than the window size. This causes the window adjustment and data recovery to be performed during disjoint round trips. *We will test TCP extensively at small window sizes and with* tcprexmtthresh *much larger than the window.*
- Rate-Halving is sufficiently conservative to safely address persistent congestion because it quickly converges to

a window which is small enough to fit within the network, no matter how large an adjustment is needed.

- We have observed serious pathological behaviors when an overly aggressive experimental congestion control algorithm failed to continue to reduce the window during ongoing congestion. The SACK data recovery was sufficiently robust to repair the ongoing losses and sustain TCP's self clock indefinitely.
- Rate-Halving does not suffer from this problem. Consider the following situation: a routing event causes a long fast connection to switch to a slow link, such that the congestion window needs to change by a large factor. (For example when re-routing from a T3 onto a T1, the window may need to be reduced by a factor of 24.) Equation 2 will get the window nearly correct within 1 round trip (due to massive losses), perhaps with some additional adjustment on the next round trip. It may then take many round trips for the data recovery. This will keep extending the Hold State until there are no outstanding holes (and there has been one loss-less round trip). At this point *snd_cwnd* is almost certainly below *ssthresh*, so when we leave the Hold State we go directly into Slow-Start and soon experience additional losses. Since we will now be within a factor of 2 of the correct window at the end of Slow-Start, Equation 2 should put us at exactly the correct window for the new link.

  (Note that the Lost Retransmission Detection Algorithm is not a precondition of this pathological behavior because the relative phases of the losses can precess on each successive round trip. Under some conditions, persistent congestion does not cause any lost retransmissions at all, therefore requiring timeouts on lost retransmissions is insufficient by itself to guarantee that this sort of pathological behavior is prevented.)

Closing Remarks

We characterize our TCP as aggressively and precisely implementing a collection of congestion control algorithms which are more tightly controlled and less aggressive than existing TCP implementations. This precision is due to tracking the forward-most (right-most) received data *(snd_fack)* giving the congestion control algorithms a better view of the network state, independent of other parts of TCP.

We believe that the dynamics of the congestion control algorithms (Slow-Start, Congestion Avoidance, Rate-Halving, bounding parameters) can be investigated with an abstract model [MSMO97], independent of other TCP issues. Furthermore, we believe that our TCP can be validated against this model.

End Notes

Test environment and results.

All of the plots in this note are from the same TCP connection. The data sender is a Pentium processor running NetBSD 1.1 with FACK. The data receiver (SACK generator) is a DEC Alpha running Digital Unix 3.2C and SACK. The data receiver is implementing Delayed ACK while not in recovery, sending one ACK per 2 data segments.

The forward data path is FDDI, through a PSC internal router, and across a lightly loaded office ethernet to the data receiver. The data receiver has a second interface on the same FDDI ring, such that the return path used by the ACKs is FDDI only. Thus the only bottleneck is from the router onto the ethernet. This bottleneck is also carrying local office traffic, and has about 100kB of buffering. (It is in principle the controlling bottleneck for wide area traffic to PSC office workstations.) The round trip time is very small -- less than 1 ms for small packets, thus the queueing delay at the bottleneck almost always dominates over all other delays. The MSS is 1460 Bytes.

Figure 1 is generated by sniffing packets on the FDDI ring and capturing the sequence numbers, ACK and SACK fields. These are used to reconstruct Equation 2 from the SIGCOMM paper *(awnd=snd_nxt-snd_fack+retran.data.) Awnd* is plotted for each segment transmitted.

In this environment Slow-Start appears to be linear, because the window growth is absorbed by the queue at the bottleneck. During Slow-Start, data arriving at the receiver is metered to roughly 10 Mb/s by the bottleneck. The receiver ACKs alternate segments, which in turn clocks data out of the transmitter at roughly 15 Mb/s, so the queue at the bottleneck fills at the rate of 5 Mb/s. Since the min RTT is so small, the exponential part of Slow-Start is no more than about 3 segments in duration.

Since we plot *awnd* on each transmitted segment, *awnd* dithers between two values whenever Congestion Avoidance and Delayed ACK are in effect. When Delayed ACK is not in effect, the window plot is smooth. The short interval at 46.8 seconds (in the enlarged part of Figure 1) is the only visible time when the window plot is smooth. This happens to correspond exactly to the Hold State, which lasted until the receiver reported that there were no outstanding holes (and the receiver resumed using delayed ACKs.) The only effect of the bounding parameters on this trace is that first opening of the window following recovery is delayed by the duration of the Hold State.

In Figure 2, there are two clusters of lost segments. The first is 4 consecutive segments that formed the initial hole that triggered recovery. Slightly later, 2 more segments were lost. Since there is more than sufficient buffering at the bottleneck, reducing the window by a factor of 2 reduces the RTT by a factor of 2 but does not change the data rate, as indicated by the slope of the ACK numbers.

The sequence plot for new data is piecewise linear, as you would expect. The apparent curvature is an optical illusion and is not really present in the data.

Further Work

FACK behaves poorly under circumstances in which *awnd* falls below *cwnd* and a loss is experienced. The algorithm for determining the correct value of *cwnd* during recovery breaks down when the connection rate is not *cwnd*-controlled. As a result, *cwnd* gets pulled down to zero, forcing a timeout. This can be illustrated by two obvious scenarios, and probably applies to many more.

Scenario 1: Since FACK continues to send new data during recovery, it sometimes fills the window advertised by the receiver, at which point it must stop sending data. Because the Self-clock is disrupted (and one ACK at this point is likely to acknowledge nearly an entire window of data), the actual data held by the network (*awnd*) can approach zero, causing *cwnd* to be set to zero.

Scenario 2: If an application has no new data to send during recovery, *awnd* becomes zero, which in turn causes *cwnd* to be set to zero.

Bibliography

[Ho95]
J. Hoe, "Startup Dynamics of TCP's Congestion Control and Avoidance Schemes," Master's Thesis, Massachusetts Institute of Technology, June 1995.

[Ho96]
J.Hoe, "Improving the Start-up Behavior of a Congestion Control Scheme for TCP," Proceedings of SIGCOMM'96, August, 1996, Stanford, CA.

[Ja88]
V. Jacobson, "Congestion Avoidance and Control," Proceedings of SIGCOMM'88, August, 1988, Palo Alto, CA.

[MM96]
M. Mathis, J. Mahdavi, "Forward Acknowledgment: Refining TCP Congestion Control," Proceedings of SIGCOMM'96, August, 1996, Stanford, CA.

- Abstract for this paper.
- Presentation for this paper.

[MMFR96]
M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgment Options," Internet Request for Comments 2018 (rfc2018.txt)

[MSMO97]
M. Mathis, J. Semke, J. Mahdavi, T. Ott, "The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm,"Computer Communication Review, volume 27, number3, July 1997.

- Abstract for this paper.