Forward Acknowledgment: Refining TCP Congestion Control

Matthew Mathis and Jamshid Mahdavi*
Pittsburgh Supercomputing Center
<mathis@psc.edu> <mahdavi@psc.edu>

Abstract

We have developed a Forward Acknowledgment (FACK) congestion control algorithm which addresses many of the performance problems recently observed in the Internet. The FACK algorithm is based on first principles of congestion control and is designed to be used with the proposed TCP SACK option. By decoupling congestion control from other algorithms such as data recovery, it attains more precise control over the data flow in the network. We introduce two additional algorithms to improve the behavior in specific situations. Through simulations we compare FACK to both Reno and Reno with SACK. Finally, we consider the potential performance and impact of FACK in the Internet.

1 Introduction

The evolution of the Internet has pushed TCP to new limits over a wide variety of IP infrastructures. Anecdotal evidence suggests that TCP experiences lower than expected performance in a number of situations in the Internet [tcp95]. The common perception is that these weaknesses are a consequence of the failure to deploy a standard Selective Acknowledgment (SACK) [JB88] in any of today's TCP implementations. However, SACK is generally viewed as a method to address data recovery; it has not been widely investigated to address congestion control issues.

Floyd pointed out that multiple segment losses can cause Reno TCP to lose its Self-clock, resulting in a retransmission timeout [Flo95, Flo92]. These timeouts can cause a substantial performance degradation. During the timeout interval, no data is sent. In addition, the timeout is followed by a period of Slow-start. This sequence of events underutilizes the network over several round-trip times, which results in a significant performance reduction on long-delay links. At the heart of this problem is the inability of Reno TCP to accurately control congestion while recovering from dropped segments.

*This work is supported in part by National Science Foundation Grant No. NCR-9415552.

We have developed a new algorithm to improve TCP congestion control during recovery. This algorithm, called Forward Acknowledgment or FACK, works in conjunction with the proposed TCP SACK option [MMFR96]. The existence of the SACK option alone greatly improves the robustness of TCP following congestion. SACK will help TCP to survive multiple segment losses within a single window without incurring a retransmission timeout. SACK can also glean additional information about congestion state, leading to improved TCP behavior during recovery. The FACK algorithm uses this information to add more precise control to the injection of data into the network during recovery. Because FACK decouples the congestion control algorithms (which determine when and how much data to send) from the data recovery algorithms (which determine what data to send),[1] we believe that it is the simplest and most direct way to use SACK to improve congestion control.

Other researchers are currently studying congestion control issues in TCP. The research community is very interested in the potential of TCP Vegas [BOP94, DLY95]. Through the use of delay measurements, TCP Vegas attempts to eliminate the periodic self-induced segment losses caused in Reno TCP. The Vegas Congestion Avoidance Mechanism (CAM) algorithm modifies the "linear increase" phase of congestion avoidance. In another recent study, Hoe investigates congestion control issues during Slow-start [Hoe95, Hoe96]. Because our work is focused primarily on improving congestion control during recovery (the "exponential decrease" phase of congestion avoidance), it is compatible with these efforts. It is our expectation that each of these efforts can eventually be incorporated into TCP in order to incrementally improve performance.

In section 2 of this paper, we describe the principles of congestion control on which FACK is built. Section 3 presents a detailed description of the FACK algorithm. Section 4 examines the basic behavior of the FACK algorithm and several optional algorithms. Sections 5 and 6 explore the performance of the various algorithms presented in the paper. In section 7 we discuss future research directions for this work. Finally, we summarize our findings.

2 Congestion Control

2.1 Ideal Principles

In 1988, Van Jacobson published the paper that has become the standard for TCP congestion control algorithms

[1] This idea has been proposed before [CLZ87], but it has not been implemented for TCP.

[Jac88, Bra89]. We do not modify any of the algorithms described in that paper. Rather, FACK extends these congestion control algorithms to TCP's recovery interval. The key concepts of "conservation of packets," "Self-clock," "Congestion Avoidance" and "Slow-start" are reviewed below.

"Conservation of packets" requires that a new segment not be injected into the network until an old segment has left. This principle leads to an inherent stability by ensuring that the number of segments in the network remains constant. Other schemes, especially rate-based transmission, can cause the number of segments in the network to grow without bound during periods of congestion, because during congestion the transmission time for segments increases. TCP implements conservation of packets by relying on "Self-clocking": segment transmissions are generally triggered by returning acknowledgements. TCP's Self-clock contributes substantially to protecting the network from congestion.

"Congestion Avoidance" is the equilibrium state algorithm for TCP. TCP maintains a congestion window, *cwnd*, which represents the maximum amount of outstanding data on the connection. When the TCP sender detects congestion in the network — identified by the loss of one or more segments — the congestion window is halved. Under other conditions, the congestion window is increased linearly by one maximum segment size (MSS) per round trip on the network. The stability of this linear increase and multiplicative decrease algorithm has been demonstrated in many investigations since its publication in 1988 [ZSC91, FJ91, Mog92, FJ92, FJ93].

"Slow-start" is the algorithm which TCP uses to reach the equilibrium state when *cwnd* is less than a threshold, *ssthresh*. *Ssthresh* attempts to dynamically estimate the correct window size for the connection. At connection establishment and after retransmission timeouts, TCP sets *cwnd* to 1 MSS and increases *cwnd* by 1 MSS for each received ACK.[2] This exponential increase continues until *cwnd* reaches the Slow-start threshold, *ssthresh*. Once *ssthresh* is reached, TCP passes into the Congestion Avoidance regime. *Ssthresh* is set to half of the current value of *cwnd* when the sender detects congestion or undergoes a retransmission timeout.

2.2 Reno TCP Behavior

Reno TCP is currently the de facto standard implementation of TCP [Ste94]. Reno implements Slow-start and Congestion Avoidance in the manner described above. It includes the Fast Retransmit algorithm from Tahoe TCP and adds one new algorithm: Fast Recovery.

Both Fast Retransmit and Fast Recovery [Ste96] rely on counting "duplicate ACKs" – TCP acknowledgments sent by the data receiver in response to each additional received segment following some missing data.

Fast Retransmit and Fast Recovery [Jac90, Ste94] are algorithms intended to preserve Self-clock during recovery from a lost segment. Fast Retransmit uses duplicate ACKs to detect the loss of a segment. When three duplicate ACKs are detected, TCP assumes that a segment has been lost and retransmits it. The number three was chosen to minimize the likelihood of out-of-order segments triggering spurious retransmissions.

The Fast Recovery algorithm attempts to estimate how much data remains outstanding in the network by counting duplicate ACKs. It artificially inflates *cwnd* on each duplicate ACK that is received, causing new data to be transmitted as *cwnd* becomes large enough. Fast Recovery allows one (halved) window of new data to be transmitted following a Fast Retransmit.

Under single segment losses, Fast Retransmit and Fast Recovery preserve TCP's Self-clock and enable it to keep the network full while recovering from one lost segment. If there are multiple lost segments, Reno is unlikely to fully recover, resulting in a timeout and subsequent Slow-start [Flo95].

2.3 SACK TCP Behavior

The new TCP SACK option [MMFR96] is progressing through the IETF standards track. It is a slight modification to the original SACK option described in RFC1072 [JB88]. When the receiver holds non-contiguous data, it sends duplicate ACKs bearing SACK options to inform the sender which segments have been correctly received. Each block of contiguous data is expressed in the SACK option using the sequence number of the first octet of data in the block, and the sequence number of the octet just beyond the end of the block. In the new SACK option the first block is required to include the most recently received segment. Additional SACK blocks repeat previously sent SACK blocks, to increase robustness in the presence of lost ACKs.

To illustrate FACK, we compare its behavior to a SACK implementation using Reno congestion control. Since there is not yet a standard implementation of SACK, we make the following assumptions about a SACK implementation using Reno congestion control:

- Fast Retransmit and Fast Recovery are modified to not resend already SACKed segments (as one would expect).
- Fast Recovery continues to estimate the amount of outstanding data by counting returning ACKs. This assumption is made in order to retain the congestion properties of Reno TCP, and is the main distinction between a SACK implementation using Reno congestion control and a SACK implementation using FACK congestion control.
- The algorithm for detecting the end of recovery uses the presence of SACK blocks to prevent partial advances of *snd.una*[3] from causing TCP to leave the recovery state prematurely.[4]

In the remainder of this paper, "Reno+SACK" will refer to an implementation as outlined above. A SACK implementation which uses the FACK congestion control algorithm will be referred to simply as "FACK".

[2] Jacobson describes the algorithm as we do, however, he goes on to note that the time it takes to open to a given window is "$R\log_2 W$ where R is the round-trip-time and W is the window size in packets." When the receiver's Delayed ACK sends one ACK per two segments, this estimate should actually be $R\log_{1.5} W$. It is generally agreed that, during Slow-start, it is correct to increase the window size by one MSS per ACK, even when the ACK acknowledges more than one MSS of data.

[3] The TCP sender state variable *snd.una* holds the sequence number of the first byte of unacknowledged data, *snd.nxt* holds the sequence number of the first byte of unsent data. These variables are defined in the TCP standard [Pos81].

[4] This fixes a problem in Reno which has been pointed out by Hoe [Hoe95] and Floyd [Flo95]. In some cases, Reno may incorrectly reinvoke Fast Retransmit and Fast Recovery. Floyd and Hoe have observed that strengthening Reno's test for the end of recovery improves its behavior in a number of situations [FF96, Hoe95].

2.4 FACK Design Goals

Under single segment losses, Reno implements the ideal congestion control principles set forth above. However in the case of multiple losses, Reno fails to meet the ideal principles because it lacks a sufficiently accurate estimate of the data outstanding in the network, at precisely the time when it is needed most.[5]

The requisite network state information can be obtained with accurate knowledge about the forward-most data held by the receiver. By forward-most, we mean the correctly-received data with the highest sequence number. This is the origin of the name "forward acknowledgment." The goal of the FACK algorithm is to perform precise congestion control during recovery by keeping an accurate estimate of the amount of data outstanding in the network. In doing so, FACK attempts to preserve TCP's Self-clock and reduce the overall burstiness of TCP.

Note that all TCP implementations discussed in this paper have nearly identical behavior under single segment losses. This reduces the need for rigorous testing under "ordinary" conditions because all implementations have the same expected performance.

3 The FACK Algorithm

The FACK algorithm uses the additional information provided by the SACK option to keep an explicit measure of the total number of bytes of data outstanding in the network. In contrast, Reno and Reno+SACK both attempt to estimate this by assuming that each duplicate ACK received represents one segment which has left the network. The FACK algorithm is able to do this in a straightforward way by introducing two new state variables, $snd.fack$ and $retran\_data$. Also, the sender must retain information on data blocks held by the receiver, which is required in order to use SACK information to correctly retransmit data. In addition to what is needed to control data retransmission, information on retransmitted segments must be kept in order to accurately determine when they have left the network.

At the core of the FACK congestion control algorithm is a new TCP state variable in the data sender. This new variable, $snd.fack$, is updated to reflect the forward-most data held by the receiver. In non-recovery states, the $snd.fack$ variable is updated from the acknowledgment number in the TCP header and is the same as $snd.una$. During recovery (while the receiver holds non-contiguous data) the sender continues to update $snd.una$ from the acknowledgment number in the TCP header, but utilizes information contained in TCP SACK options[6] to update $snd.fack$. When a SACK block is received which acknowledges data with a higher sequence number than the current value of $snd.fack$, $snd.fack$ is updated to reflect the highest sequence number known to have been received plus one.

Sender algorithms that address reliable transport continue to use the existing state variable $snd.una$. Sender algorithms that address congestion management are altered to use $snd.fack$, which provides a more accurate view for the state of the network.

We define $awnd$ to be the data sender's estimate of the actual quantity of data outstanding in the network. Assuming that all unacknowledged segments have left the network:[7]

$$awnd = snd.nxt - snd.fack \tag{1}$$

During recovery, data which is retransmitted must also be included in the computation of $awnd$. The sender computes a new variable, $retran\_data$, reflecting the quantity of outstanding retransmitted data in the network. Each time a segment is retransmitted, $retran\_data$ is increased by the segment's size; when a retransmitted segment is determined to have left the network, $retran\_data$ is decreased by the segment's size. Therefore TCP's estimate of the amount of data outstanding in the network during recovery is given by:

$$awnd = snd.nxt - snd.fack + retran\_data \tag{2}$$

Using this measure of outstanding data, the FACK congestion control algorithm can regulate the amount of data outstanding in the network to be within one MSS of the current value of $cwnd$:[8]

```
while (awnd < cwnd)
    sendsomething();
```

The FACK congestion control algorithm does not place special requirements on *sendsomething*(); the algorithm implied by the SACK Internet-Draft is sufficient. Generally *sendsomething*() should choose to send the oldest data first.[9]

FACK derives its robustness from the simplicity of updating its state variables: if *sendsomething*() retransmits old data, it will increase $retran\_data$; if it sends new data, it advances $snd.nxt$. Correspondingly, ACKs which report new data at the receiver either decrease $retran\_data$ or advance $snd.fack$. Furthermore, if the sender receives an ACK which advances $snd.fack$ beyond the value of $snd.nxt$ at the time a segment was retransmitted (and that retransmitted segment is otherwise unaccounted for), the sender knows that the segment which was retransmitted has been lost.

3.1 Triggering Recovery

Reno invokes Fast Recovery by counting duplicate acknowledgments:

[5] The observation that Reno inaccurately assesses the network state arose as a part of ongoing research aimed at developing tools for benchmarking the production Internet [ipp96]. Our efforts focused on a tool called "TReno" for Traceroute-Reno [Mat95, Mat96], which is an evolution of an earlier tool "Windowed Ping" [Mat94]. TReno attempts to measure the available network headroom by emulating Reno TCP over a traceroute-like UDP stream. Although based on Reno congestion control, TReno was observed to exhibit significantly different behavior largely due to its precise picture of the congestion state of the network. Our investigation of the differences between TReno and Reno behaviors led us to discover FACK's underlying principles.

[6] In principle, the FACK algorithm could also be implemented by utilizing the information provided by the receiver through other mechanisms, such as TCP Timestamp option, to determine the rightmost segment received [Kar95]. This would allow the benefits of improved congestion control during recovery to be immediately realized in existing TCP implementations. However, because of the complementary nature of FACK and SACK, and the expected imminent deployment of SACK, in our research we are assuming that FACK is implemented in conjunction with SACK.

[7] This is true when the network is not reordering segments and there have been no retransmissions.

[8] In the case when $cwnd$ has been halved immediately following a lost segment, $awnd$ will be significant larger than $cwnd$. This issue is addressed in section 4.5.

[9] If *sendsomething*() chooses to send new data, it is also constrained by the receiver's window ($snd.wnd$) and must make an additional check to ensure that the new data does not lie beyond the limit imposed by $snd.wnd$. If *sendsomething*() chooses to retransmit old data, it is not constrained by the receiver's window.

```
if (dupacks == 3) {
    ...
}
```

This algorithm causes an unnecessary delay if several segments are lost prior to receiving three duplicate acknowledgments. In the FACK version, the *cwnd* adjustment and retransmission are also triggered when the receiver reports that the reassembly queue is longer than 3 segments:

```
if ((snd.fack - snd.una) > (3 * MSS) ||
    (dupacks == 3)) {
    ...
}
```

If exactly one segment is lost, the two algorithms trigger recovery on exactly the same duplicate acknowledgment.

3.2 Ending Recovery

The recovery period ends when *snd.una* advances to or beyond *snd.nxt* at the time the first loss was detected. During the recovery period, *cwnd* is held constant; when recovery ends TCP returns to Congestion Avoidance and performs linear increase on *cwnd*. In the implementation tested in this paper, a timeout is forced if it is detected that a retransmitted segment has been lost (again). This condition is included to prevent FACK from being too aggressive in the presence of persistent congestion.

4 FACK behavior

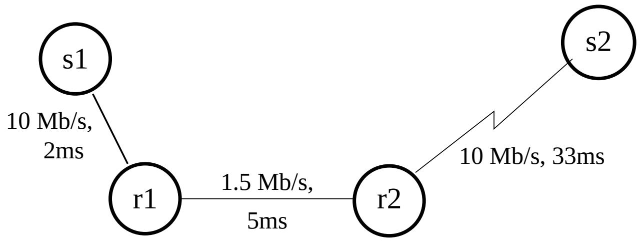
In this section we explore the behavior of the FACK algorithm in a simulator environment. We introduce another algorithm, Overdamping, which estimates the correct window more conservatively following losses as a result of Slow-start. Finally, we introduce a Rampdown algorithm to smooth data transmission during the recovery period.

4.1 Simulation Environment

We tested these new algorithms by implementing them under the LBNL simulator "ns" [MF], where we added the necessary new congestion control algorithms.[10] The simulator includes models of Tahoe, Reno, and Reno+SACK. We added a FACK sender to the simulator, but were able to use the existing SACK TCP receiver without modification. Our first set of tests uses a simple network containing four nodes (figure 1).

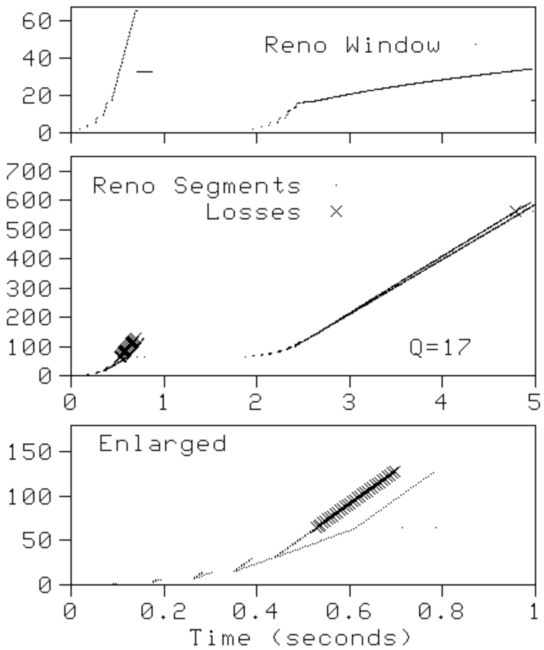
Two of these nodes represent routers connected by a 5 ms T1 link; one is a host in close proximity to these routers, and the other is a host 33 ms away. The bandwidth∗delay product for this network is 16.3 kBytes, including store and forward delays. In all of our tests we utilize an MSS of 1 kB. Thus, properly provisioned routers in this network should have queues at least 17 packets long.

We varied queue lengths in order to examine both adequately provisioned and underprovisioned cases.[11] In all of our investigations we utilize drop-tail routers. The details of the FACK algorithm and implementation do not require any



The round trip time between S1 and S2 is 80 ms, plus another 7 ms of store and forward delay, yielding a total pipe size of 16.3 kBytes.

Figure 1: The test topology



The network is provisioned with queues of length 17 packets. 30 segments are unnecessarily retransmitted.

Figure 2: Reno behavior during Slow-start.

changes to operate in networks with more intelligent queuing disciplines. However, the relative benefit of the FACK algorithm in these networks will be slightly lower because episodes of congestion in such networks are expected to be less extreme.

In this paper, most of our examples plot segment numbers vs. time in seconds.[12] Each segment is shown twice, once when it enters the bottleneck queue and once when it leaves. Dropped segments are indicated by an "×". Retransmissions always stand out because both the enqueue and dequeue events are visibly out of order. In some cases, plots of window size and router queue occupancy are shown as well.

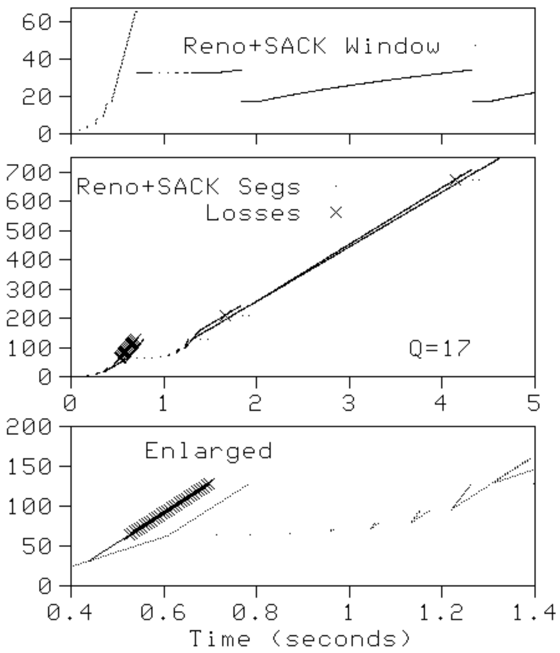
4.2 Behavior During Slow-start

During Slow-start, TCP opens its window exponentially, forcing the network into congestion and often dropping many segments. Figure 2 shows the behavior of Reno during

[10] An implementation of FACK will be available in a future release of ns.

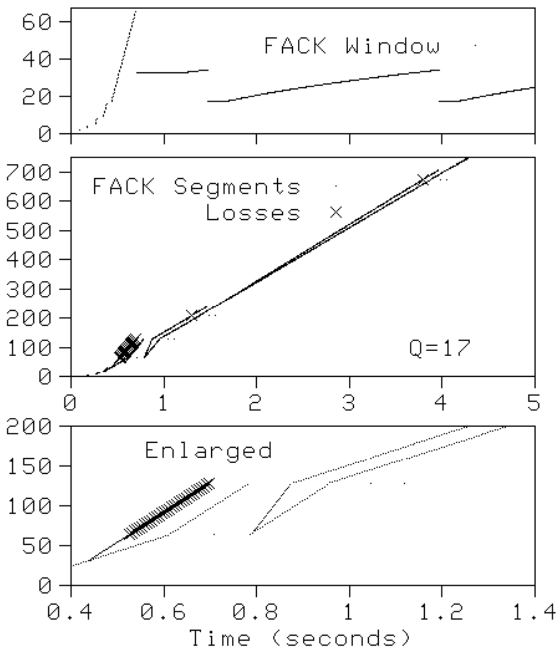
[11] Note that many historical papers investigating TCP dynamics use underbuffered networks in their simulations. We believe that any protocol development work must adequately address both properly provisioned and underbuffered networks, and protocols must be shown to be stable (if not optimal) in both environments.

[12] See http://www.psc.edu/networking/papers/ for enlarged figures.



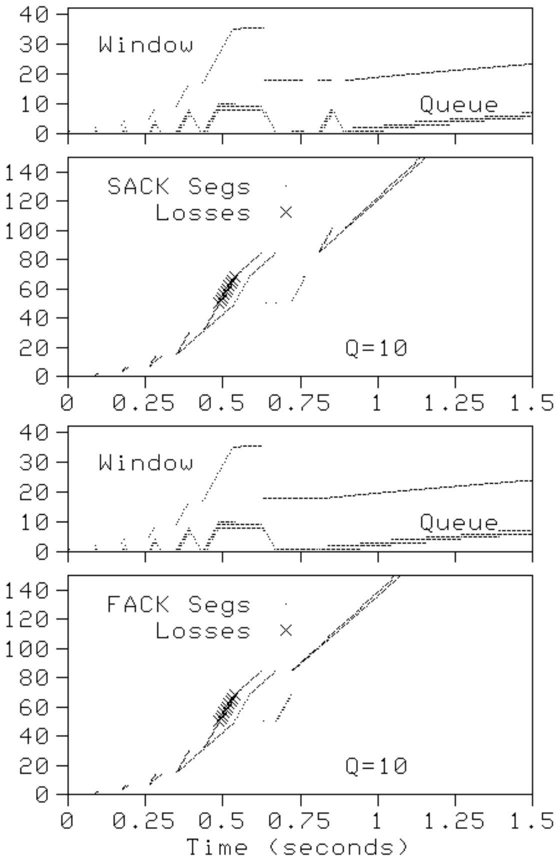
The network is provisioned with queues of length 17 packets. No data is unnecessarily retransmitted.

Figure 3: Reno+SACK behavior during Slow-start.



The network is provisioned with queues of length 17 packets. No data is unnecessarily retransmitted.

Figure 4: FACK behavior during Slow-start.



The network is provisioned with queues of length 10 packets, and *ssthresh* is preset to 35 segments. No data is unnecessarily retransmitted.

Figure 5: SACK and FACK loss recovery details.

Slow-start. Reno is unable to handle the multiple segment losses; it times out and then proceeds with a Slow-start after the timeout interval.

Figure 3 shows the behavior of Reno+SACK under the same circumstances. Reno+SACK does not incur the timeout. However, due to the large number of lost segments, Reno+SACK underestimates the window during recovery, and requires several round trip times to complete recovery.

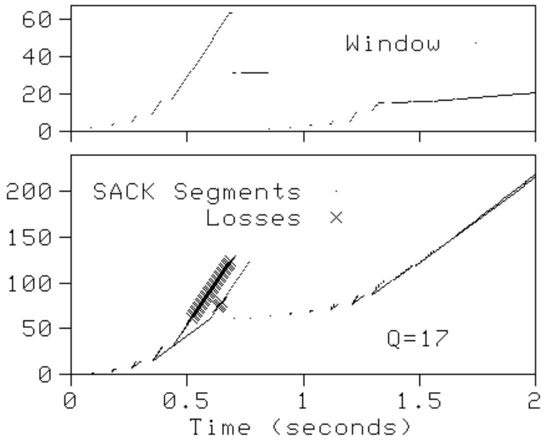
Figure 4 shows the behavior of FACK in this situation. FACK divides its window size by two, waits half of an RTT for data to exit the network, and then proceeds to retransmit lost segments.

In these examples, both Reno+SACK and FACK make no unnecessary retransmissions. Reno, on the other hand, unnecessarily retransmits 30 segments.

4.3 FACK vs. Reno+SACK

Figure 5 compares the detailed behaviors of FACK and Reno+SACK in a slightly different case. Here, the variable *ssthresh* is preset to 35 and the bottleneck queue has only 10 packet buffers. In this case, the behaviors of FACK and Reno+SACK are very similar. The primary difference is visible in the queue length at the bottleneck link. At the end of recovery (about .8 sec), Reno+SACK makes a burst transmission which causes a spike in the queue length.[13] Since the window size after the end of recovery is identical for both algorithms, FACK and Reno+SACK will have roughly the same overall performance for environments where TCP never loses more than half a window of data.

If more than half a window of data is lost, the window estimate of Reno+SACK will not be sufficiently accurate. Figure 6 shows such a case. Here, in addition to the segments lost during Slow-start, four additional segments were dropped in transit on the bottleneck link. In this case TCP runs out of ACKs before invoking Fast Recovery. In the worst case, this would result in a retransmit timeout followed by a Slow-start. One of the requirements of a SACK implementation is that if the TCP sender takes a retransmit timeout, it must clear all information about SACK blocks held by the receiver. Thus, the sender would timeout and then Slow-start with the possibility of retransmitting data which has already been received. The SACK implementation in the simulator includes an additional test specifically for the case where more than half a window of data is lost, and proceeds directly into Slow-start. This avoids the retransmit timeout, but still incurs the penalties of Slow-start and duplicated data. The final result, in this case, is that 6 round trip times are lost to the Slow-start, and 25 segments are unnecessarily retransmitted. Note that it would be possible to further optimize Reno+SACK for this case by keeping the information stored in the SACK blocks. The resulting TCP would only take the penalty of the Slow-start for this case.



The network is provisioned with queues of length 17 packets, and four non-congestion related losses have been injected. 25 segments are unnecessarily retransmitted.

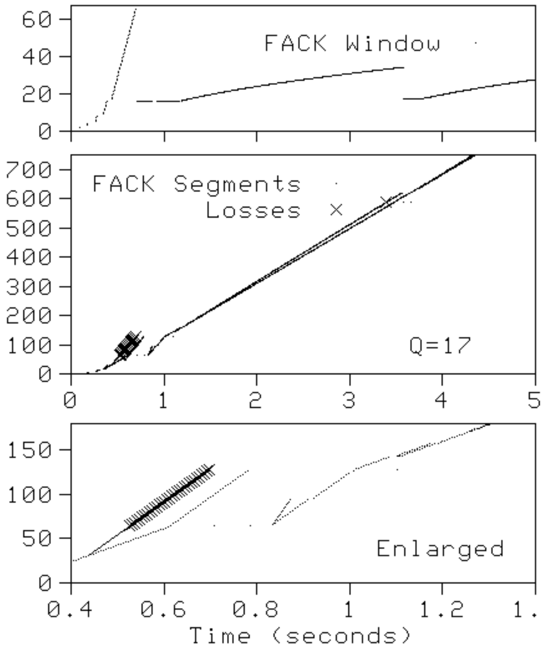
Figure 6: SACK recovery detail under greater than 1/2 window of loss.

4.4 Slow-start Overshoots and the Overdamping Algorithm

In both the Reno and FACK examples, the congestion window is almost immediately cut in half a second time. The reason for this behavior is that when dividing *cwnd* by two, TCP should utilize the value of *cwnd* when the first lost segment was sent. At this point, the session fills the available buffer space exactly, whereas when the loss is detected one RTT later, *cwnd* has doubled.[14] We can improve this behavior by implementing the following additional window adjustment:

```
if (cwnd <= ssthresh + .5*mss)
    cwnd /= 2;
```

If TCP has recently[15] been in Slow-start, it reduces *cwnd* by an extra factor of two prior to reducing the window and setting *ssthresh*. This takes into account the fact that, at the time the segment was sent, *cwnd* was smaller than it was at the time the loss was detected, and therefore is more conservative about setting *cwnd* and *ssthresh*. With this additional algorithm in place, the results of our test simulation are shown in figure 7. Note that the first segment loss following Slow-start does not occur until time 3.4 sec, compared with figure 4 where it occurs at time 1.7 sec.



The network is provisioned with queues of length 17 packets. No data is unnecessarily retransmitted.

Figure 7: Behavior of FACK with Overdamping.

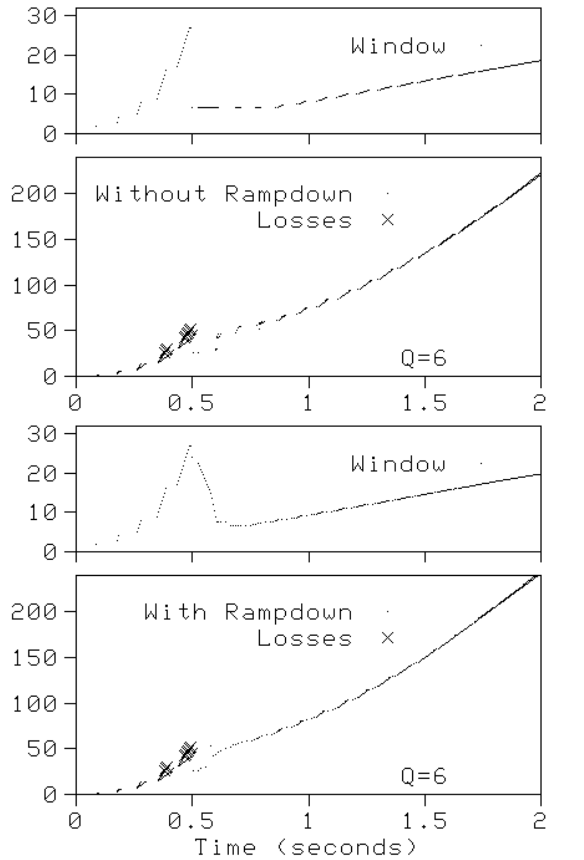
4.5 Data Smoothing

During a congestion epoch, when one or more segments are lost, TCP performs an exponential backoff by cutting *cwnd*

[13] The size of the burst will be equal to the number of dropped segments plus the number of dropped ACKs minus one.

[14] In this section, we have not utilized Delayed ACKs, which would cause *cwnd* to increase by a factor of 1.5. The effects of Overdamping in this case are shown in section 5.

[15] We define "recently" as "within one half of a round-trip" of being in Slow-start. The choice of one half is somewhat subjective, but preserves continuity at the boundary conditions.



The network is provisioned with queues of length 6 packets. No data is unnecessarily retransmitted.

Figure 8: FACK behavior with (bottom) and without (top) Rampdown. Overdamping is utilized in both cases.

in half. In current TCP implementations, the sender stops transmitting data until enough data has left the network to reduce *awnd* below the new value of *cwnd*. The sender then resumes transmission of data. This typically results in a full window of data being transmitted in one half of a round trip time, resulting in uneven transmission of data for this and subsequent round trips. Solutions to this problem have been suggested [Hoe95, Jac95], but have not yet been deployed.[16] The recommended solution for this problem is to smooth the transmission of data over one RTT by slowly reducing *cwnd*, rather than instantly halving it. We implemented this solution as follows:

At the time congestion is detected:

$$wintrim = (snd.nxt - snd.fack) * (1 - winmult) \quad (3)$$

Each time *snd.fack* advances by $\Delta fack$:

$$wintrim = wintrim - \Delta fack * (1 - winmult) \quad (4)$$

Here, *wintrim* is added to *cwnd* during the "Rampdown" phase of congestion control. At the time recovery begins, *cwnd*+*wintrim* is slightly less than *awnd*. After one round trip of recovery, *wintrim* is reduced to zero. While *wintrim* is non-zero, it acts to smooth the data evenly over one round trip, so that exactly *cwnd* bytes of data are outstanding at the end of this round trip. The variable *winmult* is the scale factor controlling how quickly *wintrim* is pulled to zero. Normally *winmult* is set to 0.5; if Overdamping is invoked, *winmult* is set to 0.25 instead.

In figure 8 we set the queue length in the routers to 6 packets, causing the network to be underutilized following Slow-start. In each RTT following Slow-start, FACK with Overdamping (top of figure 8) clusters its transmissions together. On the other hand, FACK with Overdamping and Rampdown (bottom of figure 8) evenly distributes the data across a full round trip time, minimizing the effects of bursts on the network.

5 Comparison of Algorithm Performance During Slow-start

In order to compare the performance of the various algorithms presented in section 4, we ran simulations of six algorithms over an exhaustive range of queue-lengths in the bottleneck router. The six algorithms are Reno, Reno+SACK, FACK, FACK with Overdamping, FACK with Rampdown, and FACK with both Overdamping and Rampdown. In order to compare the performance of the various algorithms in a meaningful way, we computed the "lost opportunity" for each run — the amount of additional data which could have been sent if the connection had run entirely in Congestion Avoidance. Events which cause idle time on the link during Slow-start, such as retransmit timeouts or deep reductions in *cwnd*, result in higher "lost opportunity".

The results of this comparison are shown in figure 9. The upper graph shows the "lost opportunity" for each algorithm with a receiver which acknowledges every segment (as used in all of the examples in Section 4). The lower graph uses a receiver with Delayed ACK.[17]

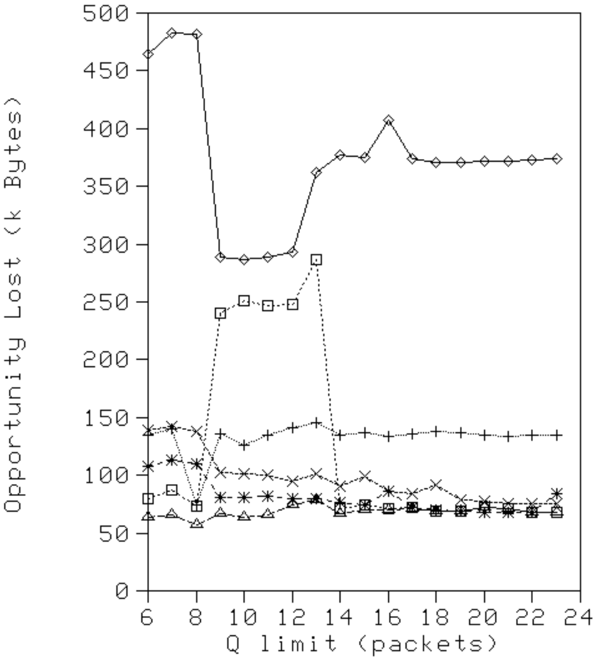
In both graphs, the effects of retransmit timeouts in Reno are clearly visible at all queue sizes. Without Delayed ACK, Reno loses between 300 kB and 500 kB of potential data transfer capability during slowstart. With Delayed ACK, this value increases to between 650 kB and 900 kB. All of the options presented for SACK congestion control perform significantly better than Reno in the cases presented here.

Without Delayed ACK, the FACK algorithm alone shows poor performance for a subset of the queue sizes examined. In these cases, FACK is too aggressive following Slow-start, and takes additional packet loss resulting in a retransmission timeout. Reno+SACK also shows lower performance across all queue sizes than the remaining three variations of FACK. This is the result of additional round trips caused by ACK starvation immediately following Slow-start (see figure 3). The two versions of FACK which include the Overdamping algorithm show poorer performance at low queue lengths. The best and most consistent performer is the FACK algorithm with Rampdown alone.

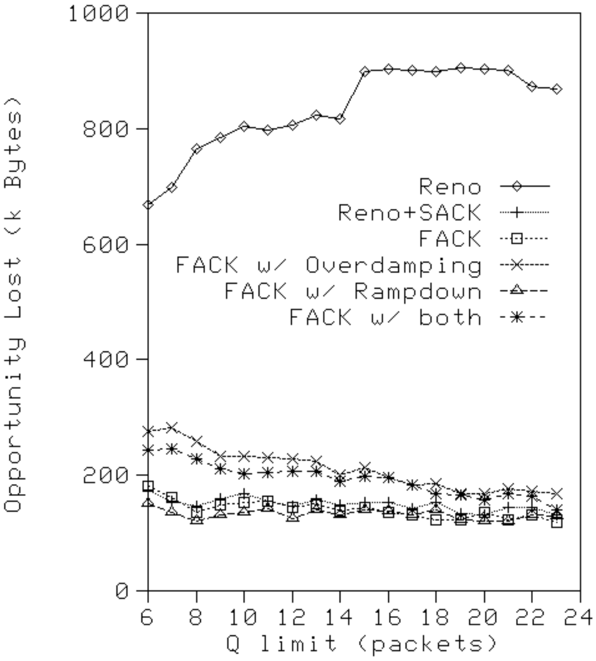
With Delayed ACK, the FACK and Reno+SACK cases no longer exhibit the behaviors mentioned above, because Slow-start does not push the network as far into congestion. The effects of Overdamping are even more pronounced, and even at the largest queue sizes we tested, Overdamping is too conservative compared with the other algorithms.

[16] We are aware of one research group working with a TCP implementation which includes a solution to this problem similar to ours [Bal96].

[17] A Delayed ACK receiver sends ACKs less frequently, and at minimum, sends one ACK for every two MSS of data received. Delayed ACK is used by almost all TCP implementations in the Internet.



The receiver is not using Delayed ACK.



The receiver is using Delayed ACK.

Figure 9: Comparison of the behavior of various congestion algorithms during Slow-start.

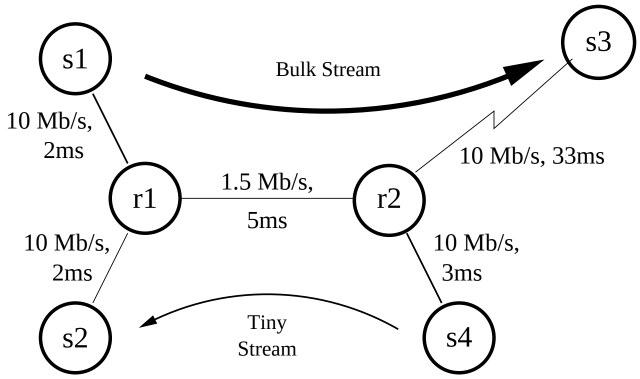
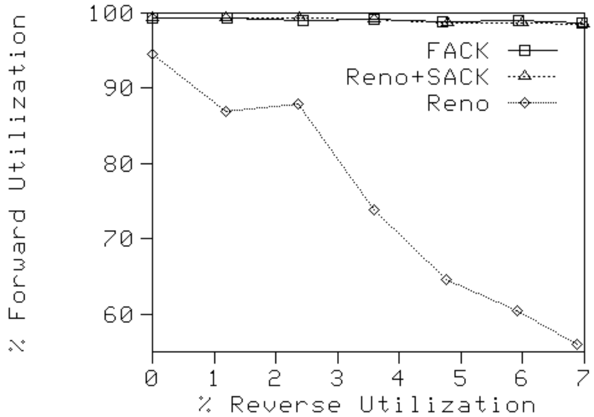


Figure 10: The jitter test topology



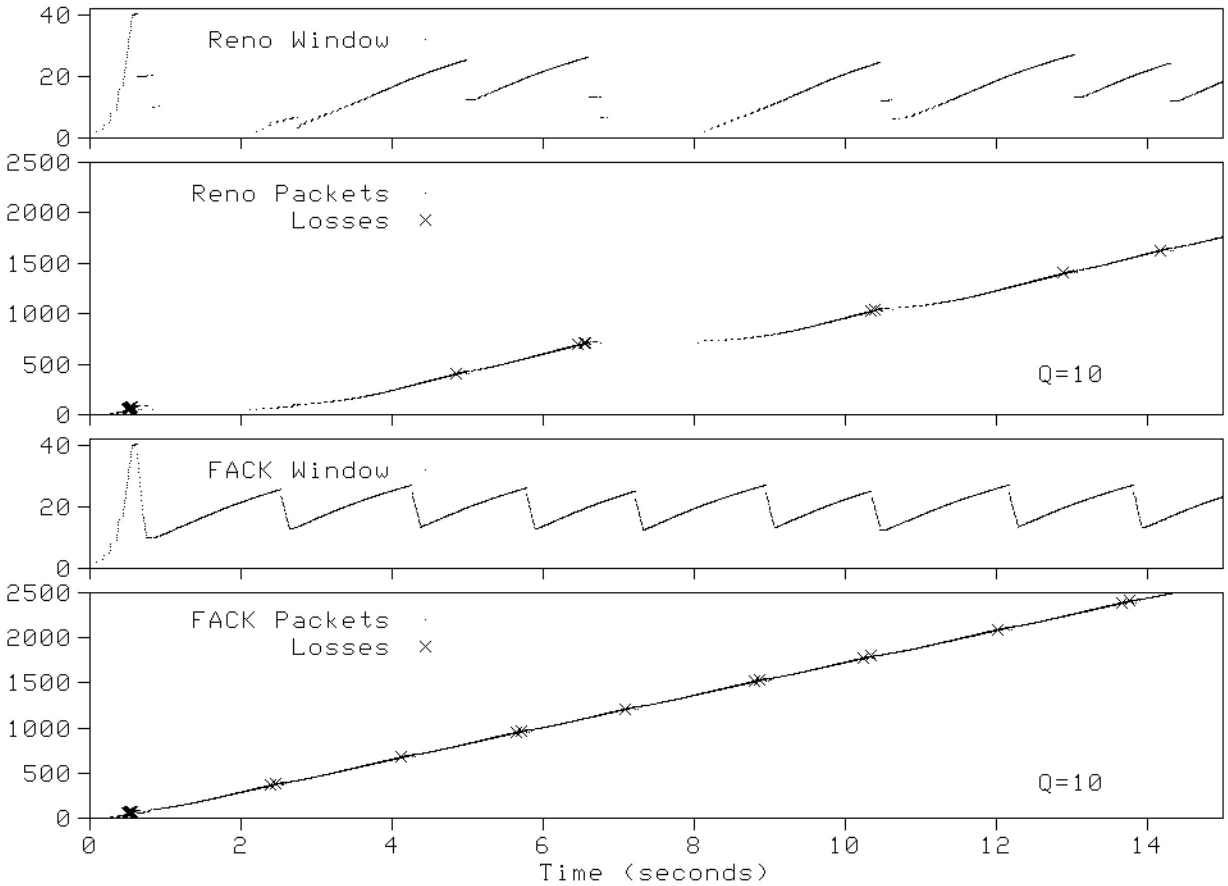
TCP forward path utilization as a function of the reverse path utilization. Note that 7% load on the reverse path causes nearly 45% idle capacity on the forward path. This example uses a 20 packet queue length, which is more than sufficient buffering for the network.

Figure 11: Comparison of FACK, Reno, and Reno+SACK

6 Performance Comparisons

We have investigated the behavior and performance of the various congestion control algorithms under several scenarios. One scenario, in which TCP is subjected to delay jitter and bursty losses, demonstrates some interesting differences between Reno, Reno+SACK, and FACK.

In the simulator, we have been able to investigate TCP's behavior in this situation with a single, very low bandwidth data stream in the reverse direction (figure 10). The reverse data stream is one connection with small, randomly distributed bursts of data at an average rate of two bursts per second. The bursts are of small constant size for each run, ranging from 1 to 6 kB. This traffic could be, for example, characteristic of a small NetNews stream or sporadic e-mail. In this environment, we ran each of the algorithms — Reno, Reno+Sack and FACK — and compared their performance. Figure 11 shows the forward path performance versus the reverse path load for each algorithm. Note that with only 7% load on the reverse path, Reno leaves almost 50% idle capacity on the forward path. This reflects the combined effects of ACK compression [ZSC91], drop-tail routers and the high penalty of retransmit timeouts. Note that this example uses a 20 packet queue length, which is more than sufficient buffering for this network.



In this trace we slightly reduced the buffering from figure 11, to accent interesting detail. All of the behaviors shown in this figure are present in one or more of the simulations used to generate figure 11.

Figure 12: Reno and Fack with jitter

6.1 Reno vs. FACK

Figure 12 shows detailed behavior of Reno and FACK in a situation only slightly different than in figure 11. The tiny reverse traffic causes ACK compression and competes for router buffer space, which, in turn, causes clusters of packet loss in the bulk stream.

In response to these clusters of loss, Reno behavior appears chaotic, showing multiple window adjustments in a single congestion episode and timeouts due to loss of its Self-clock.

The bottom of figure 12 shows FACK (with Overdamping and Rampdown) in exactly the same situation. Even though many congestion epochs experience clusters of loss, FACK correctly performs exactly one multiplicative decrease of *cwnd* per congestion epoch, preserves the TCP Self-clock, and avoids all timeouts.[18] In this regime FACK appears to be a stable, well-behaved control system, consistent with the principles of ideal congestion control.

6.2 Impact to the Internet

In the Internet, anecdotal evidence suggests that episodes of multiple packet loss in one round trip are common. Paxson observes the following behavior in roughly 13% of the traces he collected at major Internet exchange points:

> ...a fast retransmit followed by a retransmit timeout, with the additional condition that the packet retransmitted after the retransmit timeout had not been previously retransmitted... [FF96]

It is most likely that this behavior is the result of minor congestion episodes which cause multiple packet loss in one round trip. Note that because only Reno TCP implementations exhibit this particular behavior, the prevalence of multiple packet loss within one round trip may be significantly more common than suggested by this data.

On our networks at PSC (a national supercomputing center with high bandwidth connectivity to the global Internet), the behavior shown in figure 12 appears regularly for bulk data transfers over moderately loaded wide area links.[19] The deployment of any version of SACK should nearly double the throughput of bulk transfers using TCP for these cases. In addition, we believe SACK TCP will be less biased against ATM than Reno TCP. For more typical Internet transfers, the benefits of SACK will likely be more moderate, but still result in overall improvements to both latency and goodput.

[18] Reno+SACK performs as well as FACK in this situation.

[19] Over a fixed path, Reno's performance can be improved by defeating TCP's *cwnd* calculation by setting the maximum window size to just slightly smaller than needed to fill the network.

7 Future Work

We are currently working on an implementation of SACK TCP which will include FACK.[20] Once implemented, FACK should be evaluated in both a testbed environment and in the Internet, to verify the performance of the algorithms and to look for any adverse side effects. These investigations should also explore the data recovery aspects of SACK.

There are several unresolved issues surrounding the algorithms presented in this paper. We are investigating a single, simple algorithm to replace the Overdamping and Rampdown, as well as several methods for addressing persistent congestion (when halving is not a sufficient window reduction). We have been moderately successful at deriving closed-form mathematical models for FACK TCP performance in some topologies and believe that this technique deserves further exploration.

The new state variable *snd.fack* might also be used to strengthen Round Trip Time Measurements (RTTM) and Protection Against Wrapped Sequence (PAWS) algorithms [JBB92] during recovery.

The FACK algorithm was first implemented in TReno, an Internet performance metric [Mat96]. Tools to measure Internet performance should track the evolution of TCP [Mat].

The production Internet still lacks adequate attention to issues of congestion and congestion detection. Many routers are incapable of providing full bandwidth×delay buffering and do not signal the onset of congestion through mechanisms such as Random Early Detection (RED) [FJ93]. Although the FACK algorithm is designed to help in times of congestion, it is not a substitute for these signals at the Internet layer. The transport and internet layers must work together to improve the behavior of the Internet under high load.

Other current research into TCP congestion is largely independent of FACK. The Congestion Avoidance Mechanism (CAM) of TCP Vegas [BOP94, DLY95] attempts to avoid unnecessary inflation of the congestion window through delay sensing techniques. Hoe has done extensive work in analyzing the effects of congestion during Slow-start [Hoe95, Hoe96], where there can be significant performance problems. The implementation of SACK and/or FACK may reduce the gravity of these problems, but will not eliminate them. Both of these efforts address different aspects of the TCP congestion control problem. Hoe also discusses a form of Rampdown, which was the inspiration for this part of our work. It should be possible to incorporate all of these concepts in a single TCP implementation, allowing for study of their combined benefits.

Finally, applications which do not use TCP are becoming more prevalent in the Internet, and many of these applications pay little or no attention to congestion control issues. The more predictable behavior and better understanding of TCP congestion control may be a step toward a standardized transport layer congestion behavior for use by all Internet applications.

8 Conclusion

In this paper, we have presented the FACK algorithm for congestion control, the Overdamping algorithm to offset Slow-start overshoot, and the Rampdown algorithm for transmission smoothing. In our investigations, we have discovered that both FACK and Reno+SACK provide major performance improvements over existing Reno implementations, due primarily to the avoidance of retransmission timeouts. Eventually, Reno users will perceive SACK implementations as having a significant advantage; this will provide incentive for the rapid widespread deployment of SACK in the Internet.

The FACK algorithm has several benefits over Reno+SACK. Since FACK more accurately controls the outstanding data in the network, it is less bursty than Reno+SACK, and can recover from episodes of heavy loss better than Reno+SACK. Because FACK uniformly adheres to basic principles of congestion control, it may be possible to produce formal mathematical models of its behavior and to support further advances in congestion control theory. Furthermore, based on our experience in implementing FACK in the simulator, it is more straightforward to code and less prone to subtle bugs than Reno+SACK.

For the additional algorithms presented, Overdamping and Rampdown, we obtained mixed success. The Overdamping algorithm is too conservative in the general case. The Rampdown algorithm, however, appears to work quite well. Based on the results in this paper, future work should explore variations on the Rampdown algorithm which incorporate the ideas included in the Overdamping algorithm.

Finally, we had difficulties developing realistic simulations of the Internet's observed clustered packet loss. Current simulation technologies do not accurately model the Internet with its vast complexity and huge populations of users, hosts, connections and packets.[21] This limitation makes it difficult to predict the operational impact of deploying new protocols in the Internet. Limited simulations and traffic playback approaches are not likely to reveal phenomena resembling turbulent coupling between protocols. We hope to investigate new simulation paradigms in the future.

9 Acknowledgements

We would like to thank Sally Floyd and Steve McCanne for making the LBNL simulator publicly available, without which we would have been unable to complete this work. We are especially grateful to the five anonymous reviewers for their insightful comments on our initial draft of this work, as well as to Sally Floyd and Craig Partridge for their invaluable assistance in moving it to final form. We would like to thank Susan Blackman and Karen Fabrizius for repeated readings and markups on our grammar and spelling. Finally, we would like to acknowledge our management at PSC for encouraging our research activities on TCP performance.

[20] This implementation will be made publicly available when completed.

[21] In our experiments, we did not take advantage of the capabilities of tcplib [DJ91], which models some of these complexities.

References

[Bal96] Hari Balakrishnan, March 1996. Presentation to the IETF TCP-LW working group.

[BOP94] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP Vegas: New Techniques for COngestion Detection and Avoidance. *Proceedings of ACM SIGCOMM '94*, August 1994.

[Bra89] R. Braden. Requirements for Internet Hosts – Communication Layers, October 1989. Request for Comments 1122.

[CLZ87] D. D. Clark, M. L. Lambert, and L. Zhang. NETBLT: a high throughput transport protocol. *Computer Communications Review*, 17(5):353–359, 1987.

[DJ91] Peter B. Danzig and Sugih Jamin. tcplib: A library of TCP/IP traffic characteristics. Technical Report TR-SYS-91-01, USC Networking and Distributed Systems Laboratory, October 1991. Obtain via: ftp://catarina.usc.edu/pub/jamin/tcplib.

[DLY95] Peter B. Danzig, Zhen Liu, and Limim Yan. An Evaluation of TCP Vegas by Live Emulation. *ACM SIGMetrics '95*, 1995.

[FF96] Kevin Fall and Sally Floyd. Comparisons of Tahoe, Reno and Sack TCP, May 1996. Submitted to CCR, Obtain via ftp://ftp.ee.lbl.gov/papers/sacks_v2.ps.Z.

[FJ91] Sally Floyd and Van Jacobson. Traffic Phase Effects in Packet-Switched Gateways. *Computer Communications Review*, 21(2), April 1991.

[FJ92] Sally Floyd and Van Jacobson. On Traffic Phase Effects in Packet-Switched Gateways. *Internetworking: Research and Experience*, 3(3):115–156, September 1992.

[FJ93] Sally Floyd and Van Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, August 1993.

[Flo92] Sally Floyd, February 1992. Private Communication.

[Flo95] Sally Floyd. TCP and Successive Fast Retransmits, February 1995. Obtain via ftp://ftp.ee.lbl.gov/papers/fastretrans.ps.

[Hoe95] Janey C. Hoe. Startup Dynamics of TCP's Congestion Control and Avoidance Schemes. Master's thesis, Massachusetts Institute of Technology, June 1995.

[Hoe96] Janey C. Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. *Proceedings of ACM SIGCOMM '96*, August 1996.

[ipp96] Charter of the Benchmarking Working Group (BMWG) of the IETF, 1996. Obtain via: http://www.ietf.cnri.reston.va.us/html.charters/bmwg–charter.html.

[Jac88] Van Jacobson. Congestion Avoidance and Control. *Proceedings of ACM SIGCOMM '88*, August 1988.

[Jac90] Van L. Jacobson. Fast Retransmit. Message to the end2end-interest mailing list, April 1990.

[Jac95] Van Jacobson, July 1995. Private Communication.

[JB88] V. Jacobson and R. Braden. TCP extensions for long-delay paths, October 1988. Request for Comments 1072.

[JBB92] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance, May 1992. Request for Comments 1323.

[Kar95] Phil Karn, December 1995. Private Communication.

[Mat] Matthew Mathis. Internet Performance and IP Provider Metrics information page. http://www.psc.edu/~mathis/ippm/.

[Mat94] Matthew B. Mathis. Windowed Ping: An IP Layer Performance Diagnostic. In *Proceedings of INET'94/JENC5*, volume 2, Prague, Czech Republic, June 1994.

[Mat95] Matthew Mathis. Source code for the TReno package, 1995. Obtain via: ftp://ftp.psc.edu/pub/net_tools/treno.shar.

[Mat96] Matthew Mathis. Diagnosing Internet Congestion with a Transport Layer Performance Tool. In *Proceedings of INET'96*, Montreal, Quebec, June 1996.

[MF] S. McCanne and S. Floyd. ns–LBNL Network Simulator. Obtain via: http://www–nrg.ee.lbl.gov/ns/.

[MMFR96] Matthew Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. TCP Selective Acknowledgement Options, May 1996. Internet Draft ("work in progress") draft-ietf-tcplw-sack-02.txt, Expires: 29/7/96.

[Mog92] Jeff C. Mogul. Observing TCP Dynamics in Real Networks. *Proceedings of ACM SIGCOMM '92*, pages 305–317, October 1992.

[Pos81] J. Postel. Transmission Control Protocol, September 1981. Request for Comments 793.

[Ste94] W. Stevens. *TCP/IP Illustrated*, volume 1. Addison-Wesley, Reading MA, 1994.

[Ste96] W. Richard Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, March 1996. Currently an Internet Draft: draft-stevens-tcpca-spec-01.txt.

[tcp95] Minutes of the tcpfix meeting at the 34th IETF, in Dallas TX, December 1995. Obtain via: http://www.ietf.cnri.reston.va.us/proceedings/95dec/tsv/tcplw.html.

[ZSC91] Lixia Zhang, Scott Shenker, and David D. Clark. Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic. *Proceedings of ACM SIGCOMM '91*, pages 133–148, 1991.