Network Working Group   M. Allman, Editor
Request for Comments: 2760   NASA Glenn Research Center/BBN Technologies
Category: Informational   S. Dawkins
                                                              Nortel
                                                           D. Glover
                                                           J. Griner
                                                             D. Tran
                                         NASA Glenn Research Center
                                                        T. Henderson
                                     University of California at Berkeley
                                                        J. Heidemann
                                                            J. Touch
                                   University of Southern California/ISI
                                                           H. Kruse
                                                        S. Ostermann
                                                     Ohio University
                                                           K. Scott
                                            The MITRE Corporation
                                                           J. Semke
                                     Pittsburgh Supercomputing Center
                                                     February 2000

               Ongoing TCP Research Related to Satellites

Status of this Memo

Copyright Notice

Abstract

   This document outlines possible TCP enhancements that may allow TCP
   to better utilize the available bandwidth provided by networks
   containing satellite links. The algorithms and mechanisms outlined
   have not been judged to be mature enough to be recommended by the
   IETF. The goal of this document is to educate researchers as to the
   current work and progress being done in TCP research related to
   satellite networks.

Table of Contents

1 Introduction

   This document outlines mechanisms that may help the Transmission
   Control Protocol (TCP) [Pos81] better utilize the bandwidth provided
   by long-delay satellite environments. These mechanisms may also help
   in other environments or for other protocols. The proposals outlined
   in this document are currently being studied throughout the research
   community. Therefore, these mechanisms are not mature enough to be
   recommended for wide-spread use by the IETF. However, some of these
   mechanisms may be safely used today. It is hoped that this document
   will stimulate further study into the described mechanisms. If, at

some point, the mechanisms discussed in this memo prove to be safe
and appropriate to be recommended for general use, the appropriate
IETF documents will be written.

It should be noted that non-TCP mechanisms that help performance over
satellite links do exist (e.g., application-level changes, queueing
disciplines, etc.). However, outlining these non-TCP mitigations is
beyond the scope of this document and therefore is left as future
work. Additionally, there are a number of mitigations to TCP's
performance problems that involve very active intervention by
gateways along the end-to-end path from the sender to the receiver.
Documenting the pros and cons of such solutions is also left as
future work.

2   Satellite Architectures

   Specific characteristics of satellite links and the impact these
   characteristics have on TCP are presented in RFC 2488 [AGS99]. This
   section discusses several possible topologies where satellite links
   may be integrated into the global Internet. The mitigation outlined
   in section 3 will include a discussion of which environment the
   mechanism is expected to benefit.

2.1 Asymmetric Satellite Networks

   Some satellite networks exhibit a bandwidth asymmetry, a larger data
   rate in one direction than the reverse direction, because of limits
   on the transmission power and the antenna size at one end of the
   link. Meanwhile, some other satellite systems are unidirectional and
   use a non-satellite return path (such as a dialup modem link). The
   nature of most TCP traffic is asymmetric with data flowing in one
   direction and acknowledgments in opposite direction. However, the
   term asymmetric in this document refers to different physical
   capacities in the forward and return links. Asymmetry has been shown
   to be a problem for TCP [BPK97,BPK98].

2.2 Satellite Link as Last Hop

   Satellite links that provide service directly to end users, as
   opposed to satellite links located in the middle of a network, may
   allow for specialized design of protocols used over the last hop.
   Some satellite providers use the satellite link as a shared high
   speed downlink to users with a lower speed, non-shared terrestrial
   link that is used as a return link for requests and acknowledgments.
   Many times this creates an asymmetric network, as discussed above.

2.3 Hybrid Satellite Networks

   In the more general case, satellite links may be located at any point
   in the network topology. In this case, the satellite link acts as
   just another link between two gateways. In this environment, a given
   connection may be sent over terrestrial links (including terrestrial
   wireless), as well as satellite links. On the other hand, a
   connection could also travel over only the terrestrial network or
   only over the satellite portion of the network.

2.4 Point-to-Point Satellite Networks

   In point-to-point satellite networks, the only hop in the network is
   over the satellite link. This pure satellite environment exhibits
   only the problems associated with the satellite links, as outlined in
   [AGS99]. Since this is a private network, some mitigations that are
   not appropriate for shared networks can be considered.

2.5 Multiple Satellite Hops

   In some situations, network traffic may traverse multiple satellite
   hops between the source and the destination. Such an environment
   aggravates the satellite characteristics described in [AGS99].

3   Mitigations

   The following sections will discuss various techniques for mitigating
   the problems TCP faces in the satellite environment. Each of the
   following sections will be organized as follows: First, each
   mitigation will be briefly outlined. Next, research work involving
   the mechanism in question will be briefly discussed. Next the
   implementation issues of the mechanism will be presented (including
   whether or not the particular mechanism presents any dangers to
   shared networks). Then a discussion of the mechanism's potential
   with regard to the topologies outlined above is given. Finally, the
   relationships and possible interactions with other TCP mechanisms are
   outlined. The reader is expected to be familiar with the TCP
   terminology used in [AGS99].

3.1 TCP For Transactions

3.1.1 Mitigation Description

   TCP uses a three-way handshake to setup a connection between two
   hosts [Pos81]. This connection setup requires 1-1.5 round-trip times
   (RTTs), depending upon whether the data sender started the connection
   actively or passively. This startup time can be eliminated by using
   TCP extensions for transactions (T/TCP) [Bra94]. After the first

connection between a pair of hosts is established, T/TCP is able to
bypass the three-way handshake, allowing the data sender to begin
transmitting data in the first segment sent (along with the SYN).
This is especially helpful for short request/response traffic, as it
saves a potentially long setup phase when no useful data is being
transmitted.

3.1.2 Research

T/TCP is outlined and analyzed in [Bra92,Bra94].

3.1.3 Implementation Issues

T/TCP requires changes in the TCP stacks of both the data sender and
the data receiver. While T/TCP is safe to implement in shared
networks from a congestion control perspective, several security
implications of sending data in the first data segment have been
identified [ddKI99].

3.1.4 Topology Considerations

It is expected that T/TCP will be equally beneficial in all
environments outlined in section 2.

3.1.5 Possible Interaction and Relationships with Other Research

T/TCP allows data transfer to start more rapidly, much like using a
larger initial congestion window (see section 3.2.1), delayed ACKs
after slow start (section 3.2.3) or byte counting (section 3.2.2).

3.2 Slow Start

The slow start algorithm is used to gradually increase the size of
TCP's congestion window (cwnd) [Jac88,Ste97,APS99]. The algorithm is
an important safe-guard against transmitting an inappropriate amount
of data into the network when the connection starts up. However,
slow start can also waste available network capacity, especially in
long-delay networks [All97a,Hay97]. Slow start is particularly
inefficient for transfers that are short compared to the
delay*bandwidth product of the network (e.g., WWW transfers).

Delayed ACKs are another source of wasted capacity during the slow
start phase. RFC 1122 [Bra89] suggests data receivers refrain from
ACKing every incoming data segment. However, every second full-sized
segment should be ACKed. If a second full-sized segment does not
arrive within a given timeout, an ACK must be generated (this timeout
cannot exceed 500 ms). Since the data sender increases the size of
cwnd based on the number of arriving ACKs, reducing the number of

ACKs slows the cwnd growth rate. In addition, when TCP starts
sending, it sends 1 segment. When using delayed ACKs a second
segment must arrive before an ACK is sent. Therefore, the receiver
is always forced to wait for the delayed ACK timer to expire before
ACKing the first segment, which also increases the transfer time.

Several proposals have suggested ways to make slow start less time
consuming. These proposals are briefly outlined below and references
to the research work given.

3.2.1 Larger Initial Window

3.2.1.1 Mitigation Description

One method that will reduce the amount of time required by slow start
(and therefore, the amount of wasted capacity) is to increase the
initial value of cwnd. An experimental TCP extension outlined in
[AFP98] allows the initial size of cwnd to be increased from 1
segment to that given in equation (1).

$$\min (4*MSS, \max (2*MSS, 4380 \text{ bytes})) \qquad (1)$$

By increasing the initial value of cwnd, more packets are sent during
the first RTT of data transmission, which will trigger more ACKs,
allowing the congestion window to open more rapidly. In addition, by
sending at least 2 segments initially, the first segment does not
need to wait for the delayed ACK timer to expire as is the case when
the initial size of cwnd is 1 segment (as discussed above).
Therefore, the value of cwnd given in equation 1 saves up to 3 RTTs
and a delayed ACK timeout when compared to an initial cwnd of 1
segment.

Also, we note that RFC 2581 [APS99], a standards-track document,
allows a TCP to use an initial cwnd of up to 2 segments. This change
is highly recommended for satellite networks.

3.2.1.2 Research

Several researchers have studied the use of a larger initial window
in various environments. [Nic97] and [KAGT98] show a reduction in
WWW page transfer time over hybrid fiber coax (HFC) and satellite
links respectively. Furthermore, it has been shown that using an
initial cwnd of 4 segments does not negatively impact overall
performance over dialup modem links with a small number of buffers
[SP98]. [AHO98] shows an improvement in transfer time for 16 KB
files across the Internet and dialup modem links when using a larger
initial value for cwnd. However, a slight increase in dropped

segments was also shown. Finally, [PN98] shows improved transfer
time for WWW traffic in simulations with competing traffic, in
addition to a small increase in the drop rate.

3.2.1.3 Implementation Issues

The use of a larger initial cwnd value requires changes to the
sender's TCP stack. Using an initial congestion window of 2 segments
is allowed by RFC 2581 [APS99]. Using an initial congestion window
of 3 or 4 segments is not expected to present any danger of
congestion collapse [AFP98], however may degrade performance in some
networks.

3.2.1.4 Topology Considerations

It is expected that the use of a large initial window would be
equally beneficial to all network architectures outlined in section
2.

3.2.1.5 Possible Interaction and Relationships with Other Research

Using a fixed larger initial congestion window decreases the impact
of a long RTT on transfer time (especially for short transfers) at
the cost of bursting data into a network with unknown conditions. A
mechanism that mitigates bursts may make the use of a larger initial
congestion window more appropriate (e.g., limiting the size of line-
rate bursts [FF96] or pacing the segments in a burst [VH97a]).

Also, using delayed ACKs only after slow start (as outlined in
section 3.2.3) offers an alternative way to immediately ACK the first
segment of a transfer and open the congestion window more rapidly.
Finally, using some form of TCP state sharing among a number of
connections (as discussed in 3.8) may provide an alternative to using
a fixed larger initial window.

3.2.2 Byte Counting

3.2.2.1 Mitigation Description

As discussed above, the wide-spread use of delayed ACKs increases the
time needed by a TCP sender to increase the size of the congestion
window during slow start. This is especially harmful to flows
traversing long-delay GEO satellite links. One mechanism that has
been suggested to mitigate the problems caused by delayed ACKs is the
use of "byte counting", rather than standard ACK counting
[All97a,All98]. Using standard ACK counting, the congestion window
is increased by 1 segment for each ACK received during slow start.
However, using byte counting the congestion window increase is based

on the number of previously unacknowledged bytes covered by each
incoming ACK, rather than on the number of ACKs received. This makes
the increase relative to the amount of data transmitted, rather than
being dependent on the ACK interval used by the receiver.

Two forms of byte counting are studied in [All98]. The first is
unlimited byte counting (UBC). This mechanism simply uses the number
of previously unacknowledged bytes to increase the congestion window
each time an ACK arrives. The second form is limited byte counting
(LBC). LBC limits the amount of cwnd increase to 2 segments. This
limit throttles the size of the burst of data sent in response to a
"stretch ACK" [Pax97]. Stretch ACKs are acknowledgments that cover
more than 2 segments of previously unacknowledged data. Stretch ACKs
can occur by design [Joh95] (although this is not standard), due to
implementation bugs [All97b,PADHV99] or due to ACK loss. [All98]
shows that LBC prevents large line-rate bursts when compared to UBC,
and therefore offers fewer dropped segments and better performance.
In addition, UBC causes large bursts during slow start based loss
recovery due to the large cumulative ACKs that can arrive during loss
recovery. The behavior of UBC during loss recovery can cause large
decreases in performance and [All98] strongly recommends UBC not be
deployed without further study into mitigating the large bursts.

Note: The standards track RFC 2581 [APS99] allows a TCP to use byte
counting to increase cwnd during congestion avoidance, however not
during slow start.

3.2.2.2 Research

Using byte counting, as opposed to standard ACK counting, has been
shown to reduce the amount of time needed to increase the value of
cwnd to an appropriate size in satellite networks [All97a]. In
addition, [All98] presents a simulation comparison of byte counting
and the standard cwnd increase algorithm in uncongested networks and
networks with competing traffic. This study found that the limited
form of byte counting outlined above can improve performance, while
also increasing the drop rate slightly.

[BPK97,BPK98] also investigated unlimited byte counting in
conjunction with various ACK filtering algorithms (discussed in
section 3.10) in asymmetric networks.

3.2.2.3 Implementation Issues

   Changing from ACK counting to byte counting requires changes to the
   data sender's TCP stack. Byte counting violates the algorithm for
   increasing the congestion window outlined in RFC 2581 [APS99] (by
   making congestion window growth more aggressive during slow start)
   and therefore should not be used in shared networks.

3.2.2.4 Topology Considerations

   It has been suggested by some (and roundly criticized by others) that
   byte counting will allow TCP to provide uniform cwnd increase,
   regardless of the ACKing behavior of the receiver. In addition, byte
   counting also mitigates the retarded window growth provided by
   receivers that generate stretch ACKs because of the capacity of the
   return link, as discussed in [BPK97,BPK98]. Therefore, this change
   is expected to be especially beneficial to asymmetric networks.

3.2.2.5 Possible Interaction and Relationships with Other Research

   Unlimited byte counting should not be used without a method to
   mitigate the potentially large line-rate bursts the algorithm can
   cause. Also, LBC may send bursts that are too large for the given
   network conditions. In this case, LBC may also benefit from some
   algorithm that would lessen the impact of line-rate bursts of
   segments. Also note that using delayed ACKs only after slow start
   (as outlined in section 3.2.3) negates the limited byte counting
   algorithm because each ACK covers only one segment during slow start.
   Therefore, both ACK counting and byte counting yield the same
   increase in the congestion window at this point (in the first RTT).

3.2.3 Delayed ACKs After Slow Start

3.2.3.1 Mitigation Description

   As discussed above, TCP senders use the number of incoming ACKs to
   increase the congestion window during slow start. And, since delayed
   ACKs reduce the number of ACKs returned by the receiver by roughly
   half, the rate of growth of the congestion window is reduced. One
   proposed solution to this problem is to use delayed ACKs only after
   the slow start (DAASS) phase. This provides more ACKs while TCP is
   aggressively increasing the congestion window and less ACKs while TCP
   is in steady state, which conserves network resources.

3.2.3.2 Research

   [All98] shows that in simulation, using delayed ACKs after slow start
   (DAASS) improves transfer time when compared to a receiver that
   always generates delayed ACKs. However, DAASS also slightly
   increases the loss rate due to the increased rate of cwnd growth.

3.2.3.3 Implementation Issues

   The major problem with DAASS is in the implementation. The receiver
   has to somehow know when the sender is using the slow start
   algorithm. The receiver could implement a heuristic that attempts to
   watch the change in the amount of data being received and change the
   ACKing behavior accordingly. Or, the sender could send a message (a
   flipped bit in the TCP header, perhaps) indicating that it was using
   slow start. The implementation of DAASS is, therefore, an open
   issue.

   Using DAASS does not violate the TCP congestion control specification
   [APS99]. However, the standards (RFC 2581 [APS99]) currently
   recommend using delayed acknowledgments and DAASS goes (partially)
   against this recommendation.

3.2.3.4 Topology Considerations

   DAASS should work equally well in all scenarios presented in section
   2. However, in asymmetric networks it may aggravate ACK congestion
   in the return link, due to the increased number of ACKs (see sections
   3.9 and 3.10 for a more detailed discussion of ACK congestion).

3.2.3.5 Possible Interaction and Relationships with Other Research

   DAASS has several possible interactions with other proposals made in
   the research community. DAASS can aggravate congestion on the path
   between the data receiver and the data sender due to the increased
   number of returning acknowledgments. This can have an especially
   adverse effect on asymmetric networks that are prone to experiencing
   ACK congestion. As outlined in sections 3.9 and 3.10, several
   mitigations have been proposed to reduce the number of ACKs that are
   passed over a low-bandwidth return link. Using DAASS will increase
   the number of ACKs sent by the receiver. The interaction between
   DAASS and the methods for reducing the number of ACKs is an open
   research question. Also, as noted in section 3.2.1.5 above, DAASS
   provides some of the same benefits as using a larger initial
   congestion window and therefore it may not be desirable to use both
   mechanisms together. However, this remains an open question.
   Finally, DAASS and limited byte counting are both used to increase

   the rate at which the congestion window is opened. The DAASS
   algorithm substantially reduces the impact limited byte counting has
   on the rate of congestion window increase.

3.2.4 Terminating Slow Start

3.2.4.1 Mitigation Description

   The initial slow start phase is used by TCP to determine an
   appropriate congestion window size for the given network conditions
   [Jac88]. Slow start is terminated when TCP detects congestion, or
   when the size of cwnd reaches the size of the receiver's advertised
   window. Slow start is also terminated if cwnd grows beyond a certain
   size. The threshold at which TCP ends slow start and begins using
   the congestion avoidance algorithm is called "ssthresh" [Jac88]. In
   most implementations, the initial value for ssthresh is the
   receiver's advertised window. During slow start, TCP roughly doubles
   the size of cwnd every RTT and therefore can overwhelm the network
   with at most twice as many segments as the network can handle. By
   setting ssthresh to a value less than the receiver's advertised
   window initially, the sender may avoid overwhelming the network with
   twice the appropriate number of segments. Hoe [Hoe96] proposes using
   the packet-pair algorithm [Kes91] and the measured RTT to determine a
   more appropriate value for ssthresh. The algorithm observes the
   spacing between the first few returning ACKs to determine the
   bandwidth of the bottleneck link. Together with the measured RTT,
   the delay*bandwidth product is determined and ssthresh is set to this
   value. When TCP's cwnd reaches this reduced ssthresh, slow start is
   terminated and transmission continues using congestion avoidance,
   which is a more conservative algorithm for increasing the size of the
   congestion window.

3.2.4.2 Research

   It has been shown that estimating ssthresh can improve performance
   and decrease packet loss in simulations [Hoe96]. However, obtaining
   an accurate estimate of the available bandwidth in a dynamic network
   is very challenging, especially attempting to do so on the sending
   side of the TCP connection [AP99]. Therefore, before this mechanism
   is widely deployed, bandwidth estimation must be studied in a more
   detail.

3.2.4.3 Implementation Issues

   As outlined in [Hoe96], estimating ssthresh requires changes to the
   data sender's TCP stack. As suggested in [AP99], bandwidth estimates
   may be more accurate when taken by the TCP receiver, and therefore
   both sender and receiver changes would be required. Estimating

ssthresh is safe to implement in production networks from a
congestion control perspective, as it can only make TCP more
conservative than outlined in RFC 2581 [APS99] (assuming the TCP
implementation is using an initial ssthresh of infinity as allowed by
[APS99]).

3.2.4.4 Topology Considerations

It is expected that this mechanism will work equally well in all
symmetric topologies outlined in section 2. However, asymmetric
links pose a special problem, as the rate of the returning ACKs may
not be the bottleneck bandwidth in the forward direction. This can
lead to the sender setting ssthresh too low. Premature termination
of slow start can hurt performance, as congestion avoidance opens
cwnd more conservatively. Receiver-based bandwidth estimators do not
suffer from this problem.

3.2.4.5 Possible Interaction and Relationships with Other Research

Terminating slow start at the right time is useful to avoid multiple
dropped segments. However, using a selective acknowledgment-based
loss recovery scheme (as outlined in section 3.3.2) can drastically
improve TCP's ability to quickly recover from multiple lost segments
Therefore, it may not be as important to terminate slow start before
a large loss event occurs. [AP99] shows that using delayed
acknowledgments [Bra89] reduces the effectiveness of sender-side
bandwidth estimation. Therefore, using delayed ACKs only during slow
start (as outlined in section 3.2.3) may make bandwidth estimation
more feasible.

3.3 Loss Recovery

3.3.1 Non-SACK Based Mechanisms

3.3.1.1 Mitigation Description

Several similar algorithms have been developed and studied that
improve TCP's ability to recover from multiple lost segments in a
window of data without relying on the (often long) retransmission
timeout. These sender-side algorithms, known as NewReno TCP, do not
depend on the availability of selective acknowledgments (SACKs)
[MMFR96].

These algorithms generally work by updating the fast recovery
algorithm to use information provided by "partial ACKs" to trigger
retransmissions. A partial ACK covers some new data, but not all
data outstanding when a particular loss event starts. For instance,
consider the case when segment N is retransmitted using the fast

retransmit algorithm and segment M is the last segment sent when
segment N is resent. If segment N is the only segment lost, the ACK
elicited by the retransmission of segment N would be for segment M.
If, however, segment N+1 was also lost, the ACK elicited by the
retransmission of segment N will be N+1. This can be taken as an
indication that segment N+1 was lost and used to trigger a
retransmission.

3.3.1.2 Research

Hoe [Hoe95,Hoe96] introduced the idea of using partial ACKs to
trigger retransmissions and showed that doing so could improve
performance. [FF96] shows that in some cases using partial ACKs to
trigger retransmissions reduces the time required to recover from
multiple lost segments. However, [FF96] also shows that in some
cases (many lost segments) relying on the RTO timer can improve
performance over simply using partial ACKs to trigger all
retransmissions. [HK99] shows that using partial ACKs to trigger
retransmissions, in conjunction with SACK, improves performance when
compared to TCP using fast retransmit/fast recovery in a satellite
environment. Finally, [FH99] describes several slightly different
variants of NewReno.

3.3.1.3 Implementation Issues

Implementing these fast recovery enhancements requires changes to the
sender-side TCP stack. These changes can safely be implemented in
production networks and are allowed by RFC 2581 [APS99].

3.3.1.4 Topology Considerations

It is expected that these changes will work well in all environments
outlined in section 2.

3.3.1.5 Possible Interaction and Relationships with Other Research

See section 3.3.2.2.5.

3.3.2 SACK Based Mechanisms

3.3.2.1 Fast Recovery with SACK

3.3.2.1.1 Mitigation Description

Fall and Floyd [FF96] describe a conservative extension to the fast
recovery algorithm that takes into account information provided by
selective acknowledgments (SACKs) [MMFR96] sent by the receiver. The
algorithm starts after fast retransmit triggers the resending of a

segment. As with fast retransmit, the algorithm cuts cwnd in half
when a loss is detected. The algorithm keeps a variable called
"pipe", which is an estimate of the number of outstanding segments in
the network. The pipe variable is decremented by 1 segment for each
duplicate ACK that arrives with new SACK information. The pipe
variable is incremented by 1 for each new or retransmitted segment
sent. A segment may be sent when the value of pipe is less than cwnd
(this segment is either a retransmission per the SACK information or
a new segment if the SACK information indicates that no more
retransmits are needed).

This algorithm generally allows TCP to recover from multiple segment
losses in a window of data within one RTT of loss detection. Like
the forward acknowledgment (FACK) algorithm described below, the SACK
information allows the pipe algorithm to decouple the choice of when
to send a segment from the choice of what segment to send.

[APS99] allows the use of this algorithm, as it is consistent with
the spirit of the fast recovery algorithm.

3.3.2.1.2 Research

[FF96] shows that the above described SACK algorithm performs better
than several non-SACK based recovery algorithms when 1--4 segments
are lost from a window of data. [AHKO97] shows that the algorithm
improves performance over satellite links. Hayes [Hay97] shows the
in certain circumstances, the SACK algorithm can hurt performance by
generating a large line-rate burst of data at the end of loss
recovery, which causes further loss.

3.3.2.1.3 Implementation Issues

This algorithm is implemented in the sender's TCP stack. However, it
relies on SACK information generated by the receiver. This algorithm
is safe for shared networks and is allowed by RFC 2581 [APS99].

3.3.2.1.4 Topology Considerations

It is expected that the pipe algorithm will work equally well in all
scenarios presented in section 2.

3.3.2.1.5 Possible Interaction and Relationships with Other Research

See section 3.3.2.2.5.

3.3.2.2 Forward Acknowledgments

3.3.2.2.1 Mitigation Description

   The Forward Acknowledgment (FACK) algorithm [MM96a,MM96b] was
   developed to improve TCP congestion control during loss recovery.
   FACK uses TCP SACK options to glean additional information about the
   congestion state, adding more precise control to the injection of
   data into the network during recovery. FACK decouples the congestion
   control algorithms from the data recovery algorithms to provide a
   simple and direct way to use SACK to improve congestion control. Due
   to the separation of these two algorithms, new data may be sent
   during recovery to sustain TCP's self-clock when there is no further
   data to retransmit.

   The most recent version of FACK is Rate-Halving [MM96b], in which one
   packet is sent for every two ACKs received during recovery.
   Transmitting a segment for every-other ACK has the result of reducing
   the congestion window in one round trip to half of the number of
   packets that were successfully handled by the network (so when cwnd
   is too large by more than a factor of two it still gets reduced to
   half of what the network can sustain). Another important aspect of
   FACK with Rate-Halving is that it sustains the ACK self-clock during
   recovery because transmitting a packet for every-other ACK does not
   require half a cwnd of data to drain from the network before
   transmitting, as required by the fast recovery algorithm
   [Ste97,APS99].

   In addition, the FACK with Rate-Halving implementation provides
   Thresholded Retransmission to each lost segment. "Tcprexmtthresh" is
   the number of duplicate ACKs required by TCP to trigger a fast
   retransmit and enter recovery. FACK applies thresholded
   retransmission to all segments by waiting until tcprexmtthresh SACK
   blocks indicate that a given segment is missing before resending the
   segment. This allows reasonable behavior on links that reorder
   segments. As described above, FACK sends a segment for every second
   ACK received during recovery. New segments are transmitted except
   when tcprexmtthresh SACK blocks have been observed for a dropped
   segment, at which point the dropped segment is retransmitted.

   [APS99] allows the use of this algorithm, as it is consistent with
   the spirit of the fast recovery algorithm.

3.3.2.2.2 Research

   The original FACK algorithm is outlined in [MM96a]. The algorithm
   was later enhanced to include Rate-Halving [MM96b]. The real-world
   performance of FACK with Rate-Halving was shown to be much closer to

the theoretical maximum for TCP than either TCP Reno or the SACK-
based extensions to fast recovery outlined in section 3.3.2.1
[MSMO97].

3.3.2.2.3 Implementation Issues

In order to use FACK, the sender's TCP stack must be modified. In
addition, the receiver must be able to generate SACK options to
obtain the full benefit of using FACK. The FACK algorithm is safe
for shared networks and is allowed by RFC 2581 [APS99].

3.3.2.2.4 Topology Considerations

FACK is expected to improve performance in all environments outlined
in section 2. Since it is better able to sustain its self-clock than
TCP Reno, it may be considerably more attractive over long delay
paths.

3.3.2.2.5 Possible Interaction and Relationships with Other Research

Both SACK based loss recovery algorithms described above (the fast
recovery enhancement and the FACK algorithm) are similar in that they
attempt to effectively repair multiple lost segments from a window of
data. Which of the SACK-based loss recovery algorithms to use is
still an open research question. In addition, these algorithms are
similar to the non-SACK NewReno algorithm described in section 3.3.1,
in that they attempt to recover from multiple lost segments without
reverting to using the retransmission timer. As has been shown, the
above SACK based algorithms are more robust than the NewReno
algorithm. However, the SACK algorithm requires a cooperating TCP
receiver, which the NewReno algorithm does not. A reasonable TCP
implementation might include both a SACK-based and a NewReno-based
loss recovery algorithm such that the sender can use the most
appropriate loss recovery algorithm based on whether or not the
receiver supports SACKs. Finally, both SACK-based and non-SACK-based
versions of fast recovery have been shown to transmit a large burst
of data upon leaving loss recovery, in some cases [Hay97].
Therefore, the algorithms may benefit from some burst suppression
algorithm.

3.3.3 Explicit Congestion Notification

3.3.3.1 Mitigation Description

Explicit congestion notification (ECN) allows routers to inform TCP
senders about imminent congestion without dropping segments. Two
major forms of ECN have been studied. A router employing backward
ECN (BECN), transmits messages directly to the data originator

informing it of congestion. IP routers can accomplish this with an
ICMP Source Quench message. The arrival of a BECN signal may or may
not mean that a TCP data segment has been dropped, but it is a clear
indication that the TCP sender should reduce its sending rate (i.e.,
the value of cwnd). The second major form of congestion notification
is forward ECN (FECN). FECN routers mark data segments with a
special tag when congestion is imminent, but forward the data
segment. The data receiver then echos the congestion information
back to the sender in the ACK packet. A description of a FECN
mechanism for TCP/IP is given in [RF99].

As described in [RF99], senders transmit segments with an "ECN-
Capable Transport" bit set in the IP header of each packet. If a
router employing an active queueing strategy, such as Random Early
Detection (RED) [FJ93,BCC+98], would otherwise drop this segment, an
"Congestion Experienced" bit in the IP header is set instead. Upon
reception, the information is echoed back to TCP senders using a bit
in the TCP header. The TCP sender adjusts the congestion window just
as it would if a segment was dropped.

The implementation of ECN as specified in [RF99] requires the
deployment of active queue management mechanisms in the affected
routers. This allows the routers to signal congestion by sending TCP
a small number of "congestion signals" (segment drops or ECN
messages), rather than discarding a large number of segments, as can
happen when TCP overwhelms a drop-tail router queue.

Since satellite networks generally have higher bit-error rates than
terrestrial networks, determining whether a segment was lost due to
congestion or corruption may allow TCP to achieve better performance
in high BER environments than currently possible (due to TCP's
assumption that all loss is due to congestion). While not a solution
to this problem, adding an ECN mechanism to TCP may be a part of a
mechanism that will help achieve this goal. See section 3.3.4 for a
more detailed discussion of differentiating between corruption and
congestion based losses.

3.3.3.2 Research

[Flo94] shows that ECN is effective in reducing the segment loss rate
which yields better performance especially for short and interactive
TCP connections. Furthermore, [Flo94] also shows that ECN avoids
some unnecessary, and costly TCP retransmission timeouts. Finally,
[Flo94] also considers some of the advantages and disadvantages of
various forms of explicit congestion notification.

3.3.3.3 Implementation Issues

   Deployment of ECN requires changes to the TCP implementation on both
   sender and receiver. Additionally, deployment of ECN requires
   deployment of some active queue management infrastructure in routers.
   RED is assumed in most ECN discussions, because RED is already
   identifying segments to drop, even before its buffer space is
   exhausted. ECN simply allows the delivery of "marked" segments while
   still notifying the end nodes that congestion is occurring along the
   path. ECN is safe (from a congestion control perspective) for shared
   networks, as it maintains the same TCP congestion control principles
   as are used when congestion is detected via segment drops.

3.3.3.4 Topology Considerations

   It is expected that none of the environments outlined in section 2
   will present a bias towards or against ECN traffic.

3.3.3.5 Possible Interaction and Relationships with Other Research

   Note that some form of active queueing is necessary to use ECN (e.g.,
   RED queueing).

3.3.4 Detecting Corruption Loss

   Differentiating between congestion (loss of segments due to router
   buffer overflow or imminent buffer overflow) and corruption (loss of
   segments due to damaged bits) is a difficult problem for TCP. This
   differentiation is particularly important because the action that TCP
   should take in the two cases is entirely different. In the case of
   corruption, TCP should merely retransmit the damaged segment as soon
   as its loss is detected; there is no need for TCP to adjust its
   congestion window. On the other hand, as has been widely discussed
   above, when the TCP sender detects congestion, it should immediately
   reduce its congestion window to avoid making the congestion worse.

   TCP's defined behavior, as motivated by [Jac88,Jac90] and defined in
   [Bra89,Ste97,APS99], is to assume that all loss is due to congestion
   and to trigger the congestion control algorithms, as defined in
   [Ste97,APS99]. The loss may be detected using the fast retransmit
   algorithm, or in the worst case is detected by the expiration of
   TCP's retransmission timer.

   TCP's assumption that loss is due to congestion rather than
   corruption is a conservative mechanism that prevents congestion
   collapse [Jac88,FF98]. Over satellite networks, however, as in many
   wireless environments, loss due to corruption is more common than on
   terrestrial networks. One common partial solution to this problem is

to add Forward Error Correction (FEC) to the data that's sent over
the satellite/wireless link. A more complete discussion of the
benefits of FEC can be found in [AGS99]. However, given that FEC
does not always work or cannot be universally applied, other
mechanisms have been studied to attempt to make TCP able to
differentiate between congestion-based and corruption-based loss.

TCP segments that have been corrupted are most often dropped by
intervening routers when link-level checksum mechanisms detect that
an incoming frame has errors. Occasionally, a TCP segment containing
an error may survive without detection until it arrives at the TCP
receiving host, at which point it will almost always either fail the
IP header checksum or the TCP checksum and be discarded as in the
link-level error case. Unfortunately, in either of these cases, it's
not generally safe for the node detecting the corruption to return
information about the corrupt packet to the TCP sender because the
sending address itself might have been corrupted.

3.3.4.1 Mitigation Description

Because the probability of link errors on a satellite link is
relatively greater than on a hardwired link, it is particularly
important that the TCP sender retransmit these lost segments without
reducing its congestion window. Because corrupt segments do not
indicate congestion, there is no need for the TCP sender to enter a
congestion avoidance phase, which may waste available bandwidth.
Simulations performed in [SF98] show a performance improvement when
TCP can properly differentiate between between corruption and
congestion of wireless links.

Perhaps the greatest research challenge in detecting corruption is
getting TCP (a transport-layer protocol) to receive appropriate
information from either the network layer (IP) or the link layer.
Much of the work done to date has involved link-layer mechanisms that
retransmit damaged segments. The challenge seems to be to get these
mechanisms to make repairs in such a way that TCP understands what
happened and can respond appropriately.

3.3.4.2 Research

Research into corruption detection to date has focused primarily on
making the link level detect errors and then perform link-level
retransmissions. This work is summarized in [BKVP97,BPSK96]. One of
the problems with this promising technique is that it causes an
effective reordering of the segments from the TCP receiver's point of
view. As a simple example, if segments A B C D are sent across a
noisy link and segment B is corrupted, segments C and D may have
already crossed the link before B can be retransmitted at the link

level, causing them to arrive at the TCP receiver in the order A C D
B. This segment reordering would cause the TCP receiver to generate
duplicate ACKs upon the arrival of segments C and D. If the
reordering was bad enough, the sender would trigger the fast
retransmit algorithm in the TCP sender, in response to the duplicate
ACKs. Research presented in [MV98] proposes the idea of suppressing
or delaying the duplicate ACKs in the reverse direction to counteract
this behavior. Alternatively, proposals that make TCP more robust in
the face of re-ordered segment arrivals [Flo99] may reduce the side
effects of the re-ordering caused by link-layer retransmissions.

A more high-level approach, outlined in the [DMT96], uses a new
"corruption experienced" ICMP error message generated by routers that
detect corruption. These messages are sent in the forward direction,
toward the packet's destination, rather than in the reverse direction
as is done with ICMP Source Quench messages. Sending the error
messages in the forward direction allows this feedback to work over
asymmetric paths. As noted above, generating an error message in
response to a damaged packet is problematic because the source and
destination addresses may not be valid. The mechanism outlined in
[DMT96] gets around this problem by having the routers maintain a
small cache of recent packet destinations; when the router
experiences an error rate above some threshold, it sends an ICMP
corruption-experienced message to all of the destinations in its
cache. Each TCP receiver then must return this information to its
respective TCP sender (through a TCP option). Upon receiving an ACK
with this "corruption-experienced" option, the TCP sender assumes
that packet loss is due to corruption rather than congestion for two
round trip times (RTT) or until it receives additional link state
information (such as "link down", source quench, or additional
"corruption experienced" messages). Note that in shared networks,
ignoring segment loss for 2 RTTs may aggravate congestion by making
TCP unresponsive.

3.3.4.3 Implementation Issues

All of the techniques discussed above require changes to at least the
TCP sending and receiving stacks, as well as intermediate routers.
Due to the concerns over possibly ignoring congestion signals (i.e.,
segment drops), the above algorithm is not recommended for use in
shared networks.

3.3.4.4 Topology Considerations

It is expected that corruption detection, in general would be
beneficial in all environments outlined in section 2. It would be
particularly beneficial in the satellite/wireless environment over
which these errors may be more prevalent.

3.3.4.5 Possible Interaction and Relationships with Other Research

   SACK-based loss recovery algorithms (as described in 3.3.2) may
   reduce the impact of corrupted segments on mostly clean links because
   recovery will be able to happen more rapidly (and without relying on
   the retransmission timer). Note that while SACK-based loss recovery
   helps, throughput will still suffer in the face of non-congestion
   related packet loss.

3.4 Congestion Avoidance

3.4.1 Mitigation Description

   During congestion avoidance, in the absence of loss, the TCP sender
   adds approximately one segment to its congestion window during each
   RTT [Jac88,Ste97,APS99]. Several researchers have observed that this
   policy leads to unfair sharing of bandwidth when multiple connections
   with different RTTs traverse the same bottleneck link, with the long
   RTT connections obtaining only a small fraction of their fair share
   of the bandwidth.

   One effective solution to this problem is to deploy fair queueing and
   TCP-friendly buffer management in network routers [Sut98]. However,
   in the absence of help from the network, other researchers have
   investigated changes to the congestion avoidance policy at the TCP
   sender, as described in [Flo91,HK98].

3.4.2 Research

   The "Constant-Rate" increase policy has been studied in [Flo91,HK98].
   It attempts to equalize the rate at which TCP senders increase their
   sending rate during congestion avoidance. Both [Flo91] and [HK98]
   illustrate cases in which the "Constant-Rate" policy largely corrects
   the bias against long RTT connections, although [HK98] presents some
   evidence that such a policy may be difficult to incrementally deploy
   in an operational network. The proper selection of a constant (for
   the constant rate of increase) is an open issue.

   The "Increase-by-K" policy can be selectively used by long RTT
   connections in a heterogeneous environment. This policy simply
   changes the slope of the linear increase, with connections over a
   given RTT threshold adding "K" segments to the congestion window
   every RTT, instead of one. [HK98] presents evidence that this
   policy, when used with small values of "K", may be successful in
   reducing the unfairness while keeping the link utilization high, when
   a small number of connections share a bottleneck link. The selection
   of the constant "K," the RTT threshold to invoke this policy, and
   performance under a large number of flows are all open issues.

3.4.3 Implementation Issues

   Implementation of either the "Constant-Rate" or "Increase-by-K"
   policies requires a change to the congestion avoidance mechanism at
   the TCP sender. In the case of "Constant-Rate," such a change must
   be implemented globally. Additionally, the TCP sender must have a
   reasonably accurate estimate of the RTT of the connection. The
   algorithms outlined above violate the congestion avoidance algorithm
   as outlined in RFC 2581 [APS99] and therefore should not be
   implemented in shared networks at this time.

3.4.4 Topology Considerations

   These solutions are applicable to all satellite networks that are
   integrated with a terrestrial network, in which satellite connections
   may be competing with terrestrial connections for the same bottleneck
   link.

3.4.5 Possible Interaction and Relationships with Other Research

   As shown in [PADHV99], increasing the congestion window by multiple
   segments per RTT can cause TCP to drop multiple segments and force a
   retransmission timeout in some versions of TCP. Therefore, the above
   changes to the congestion avoidance algorithm may need to be
   accompanied by a SACK-based loss recovery algorithm that can quickly
   repair multiple dropped segments.

3.5 Multiple Data Connections

3.5.1 Mitigation Description

   One method that has been used to overcome TCP's inefficiencies in the
   satellite environment is to use multiple TCP flows to transfer a
   given file. The use of N TCP connections makes the sender N times
   more aggressive and therefore can improve throughput in some
   situations. Using N multiple TCP connections can impact the transfer
   and the network in a number of ways, which are listed below.

   1. The transfer is able to start transmission using an effective
      congestion window of N segments, rather than a single segment as
      one TCP flow uses. This allows the transfer to more quickly
      increase the effective cwnd size to an appropriate size for the
      given network. However, in some circumstances an initial window
      of N segments is inappropriate for the network conditions. In
      this case, a transfer utilizing more than one connection may
      aggravate congestion.

2. During the congestion avoidance phase, the transfer increases the
   effective cwnd by N segments per RTT, rather than the one segment
   per RTT increase that a single TCP connection provides. Again,
   this can aid the transfer by more rapidly increasing the effective
   cwnd to an appropriate point. However, this rate of increase can
   also be too aggressive for the network conditions. In this case,
   the use of multiple data connections can aggravate congestion in
   the network.

3. Using multiple connections can provide a very large overall
   congestion window. This can be an advantage for TCP
   implementations that do not support the TCP window scaling
   extension [JBB92]. However, the aggregate cwnd size across all N
   connections is equivalent to using a TCP implementation that
   supports large windows.

4. The overall cwnd decrease in the face of dropped segments is
   reduced when using N parallel connections. A single TCP
   connection reduces the effective size of cwnd to half when a
   single segment loss is detected. When utilizing N connections
   each using a window of W bytes, a single drop reduces the window
   to:

       $(N * W) - (W / 2)$

Clearly this is a less dramatic reduction in the effective cwnd size
than when using a single TCP connection. And, the amount by which
the cwnd is decreased is further reduced by increasing N.

The use of multiple data connections can increase the ability of
non-SACK TCP implementations to quickly recover from multiple dropped
segments without resorting to a timeout, assuming the dropped
segments cross connections.

The use of multiple parallel connections makes TCP overly aggressive
for many environments and can contribute to congestive collapse in
shared networks [FF99]. The advantages provided by using multiple
TCP connections are now largely provided by TCP extensions (larger
windows, SACKs, etc.). Therefore, the use of a single TCP connection
is more "network friendly" than using multiple parallel connections.
However, using multiple parallel TCP connections may provide
performance improvement in private networks.

3.5.2 Research

   Research on the use of multiple parallel TCP connections shows
   improved performance [IL92,Hah94,AOK95,AKO96]. In addition, research
   has shown that multiple TCP connections can outperform a single
   modern TCP connection (with large windows and SACK) [AHKO97].
   However, these studies did not consider the impact of using multiple
   TCP connections on competing traffic. [FF99] argues that using
   multiple simultaneous connections to transfer a given file may lead
   to congestive collapse in shared networks.

3.5.3 Implementation Issues

   To utilize multiple parallel TCP connections a client application and
   the corresponding server must be customized. As outlined in [FF99]
   using multiple parallel TCP connections is not safe (from a
   congestion control perspective) in shared networks and should not be
   used.

3.5.4 Topological Considerations

   As stated above, [FF99] outlines that the use of multiple parallel
   connections in a shared network, such as the Internet, may lead to
   congestive collapse. However, the use of multiple connections may be
   safe and beneficial in private networks. The specific topology being
   used will dictate the number of parallel connections required. Some
   work has been done to determine the appropriate number of connections
   on the fly [AKO96], but such a mechanism is far from complete.

3.5.5 Possible Interaction and Relationships with Other Research

   Using multiple concurrent TCP connections enables use of a large
   congestion window, much like the TCP window scaling option [JBB92].
   In addition, a larger initial congestion window is achieved, similar
   to using [AFP98] or TCB sharing (see section 3.8).

3.6 Pacing TCP Segments

3.6.1 Mitigation Description

   Slow-start takes several round trips to fully open the TCP congestion
   window over routes with high bandwidth-delay products. For short TCP
   connections (such as WWW traffic with HTTP/1.0), the slow-start
   overhead can preclude effective use of the high-bandwidth satellite
   links. When senders implement slow-start restart after a TCP
   connection goes idle (suggested by Jacobson and Karels [JK92]),

performance is reduced in long-lived (but bursty) connections (such
as HTTP/1.1, which uses persistent TCP connections to transfer
multiple WWW page elements) [Hei97a].

Rate-based pacing (RBP) is a technique, used in the absence of
incoming ACKs, where the data sender temporarily paces TCP segments
at a given rate to restart the ACK clock. Upon receipt of the first
ACK, pacing is discontinued and normal TCP ACK clocking resumes. The
pacing rate may either be known from recent traffic estimates (when
restarting an idle connection or from recent prior connections), or
may be known through external means (perhaps in a point-to-point or
point-to-multipoint satellite network where available bandwidth can
be assumed to be large).

In addition, pacing data during the first RTT of a transfer may allow
TCP to make effective use of high bandwidth-delay links even for
short transfers. However, in order to pace segments during the first
RTT a TCP will have to be using a non-standard initial congestion
window and a new mechanism to pace outgoing segments rather than send
them back-to-back. Determining an appropriate size for the initial
cwnd is an open research question. Pacing can also be used to reduce
bursts in general (due to buggy TCPs or byte counting, see section
3.2.2 for a discussion on byte counting).

3.6.2 Research

Simulation studies of rate-paced pacing for WWW-like traffic have
shown reductions in router congestion and drop rates [VH97a]. In
this environment, RBP substantially improves performance compared to
slow-start-after-idle for intermittent senders, and it slightly
improves performance over burst-full-cwnd-after-idle (because of
drops) [VH98]. More recently, pacing has been suggested to eliminate
burstiness in networks with ACK filtering [BPK97].

3.6.3 Implementation Issues

RBP requires only sender-side changes to TCP. Prototype
implementations of RBP are available [VH97b]. RBP requires an
additional sender timer for pacing. The overhead of timer-driven
data transfer is often considered too high for practical use.
Preliminary experiments suggest that in RBP this overhead is minimal
because RBP only requires this timer for one RTT of transmission
[VH98]. RBP is expected to make TCP more conservative in sending
bursts of data after an idle period in hosts that do not revert to
slow start after an idle period. On the other hand, RBP makes TCP
more aggressive if the sender uses the slow start algorithm to start
the ACK clock after a long idle period.

3.6.4 Topology Considerations

   RBP could be used to restart idle TCP connections for all topologies
   in Section 2. Use at the beginning of new connections would be
   restricted to topologies where available bandwidth can be estimated
   out-of-band.

3.6.5 Possible Interaction and Relationships with Other Research

   Pacing segments may benefit from sharing state amongst various flows
   between two hosts, due to the time required to determine the needed
   information. Additionally, pacing segments, rather than sending
   back-to-back segments, may make estimating the available bandwidth
   (as outlined in section 3.2.4) more difficult.

3.7 TCP Header Compression

   The TCP and IP header information needed to reliably deliver packets
   to a remote site across the Internet can add significant overhead,
   especially for interactive applications. Telnet packets, for
   example, typically carry only a few bytes of data per packet, and
   standard IPv4/TCP headers add at least 40 bytes to this; IPv6/TCP
   headers add at least 60 bytes. Much of this information remains
   relatively constant over the course of a session and so can be
   replaced by a short session identifier.

3.7.1 Mitigation Description

   Many fields in the TCP and IP headers either remain constant during
   the course of a session, change very infrequently, or can be inferred
   from other sources. For example, the source and destination
   addresses, as well as the IP version, protocol, and port fields
   generally do not change during a session. Packet length can be
   deduced from the length field of the underlying link layer protocol
   provided that the link layer packet is not padded. Packet sequence
   numbers in a forward data stream generally change with every packet,
   but increase in a predictable manner.

   The TCP/IP header compression methods described in
   [DNP99,DENP97,Jac90] reduce the overhead of TCP sessions by replacing
   the data in the TCP and IP headers that remains constant, changes
   slowly, or changes in a predictable manner with a short "connection
   number". Using this method, the sender first sends a full TCP/IP
   header, including in it a connection number that the sender will use
   to reference the connection. The receiver stores the full header and
   uses it as a template, filling in some fields from the limited

information contained in later, compressed headers. This compression
can reduce the size of an IPv4/TCP headers from 40 to as few as 3 to
5 bytes (3 bytes for some common cases, 5 bytes in general).

Compression and decompression generally happen below the IP layer, at
the end-points of a given physical link (such as at two routers
connected by a serial line). The hosts on either side of the
physical link must maintain some state about the TCP connections that
are using the link.

The decompresser must pass complete, uncompressed packets to the IP
layer. Thus header compression is transparent to routing, for
example, since an incoming packet with compressed headers is expanded
before being passed to the IP layer.

A variety of methods can be used by the compressor/decompressor to
negotiate the use of header compression. For example, the PPP serial
line protocol allows for an option exchange, during which time the
compressor/decompressor agree on whether or not to use header
compression. For older SLIP implementations, [Jac90] describes a
mechanism that uses the first bit in the IP packet as a flag.

The reduction in overhead is especially useful when the link is
bandwidth-limited such as terrestrial wireless and mobile satellite
links, where the overhead associated with transmitting the header
bits is nontrivial. Header compression has the added advantage that
for the case of uniformly distributed bit errors, compressing TCP/IP
headers can provide a better quality of service by decreasing the
packet error probability. The shorter, compressed packets are less
likely to be corrupted, and the reduction in errors increases the
connection's throughput.

Extra space is saved by encoding changes in fields that change
relatively slowly by sending only their difference from their values
in the previous packet instead of their absolute values. In order to
decode headers compressed this way, the receiver keeps a copy of each
full, reconstructed TCP header after it is decoded, and applies the
delta values from the next decoded compressed header to the
reconstructed full header template.

A disadvantage to using this delta encoding scheme where values are
encoded as deltas from their values in the previous packet is that if
a single compressed packet is lost, subsequent packets with
compressed headers can become garbled if they contain fields which
depend on the lost packet. Consider a forward data stream of packets
with compressed headers and increasing sequence numbers. If packet N
is lost, the full header of packet N+1 will be reconstructed at the
receiver using packet N-1's full header as a template. Thus the

   sequence number, which should have been calculated from packet N's
   header, will be wrong, the checksum will fail, and the packet will be
   discarded. When the sending TCP times out and retransmits a packet
   with a full header is forwarded to re-synchronize the decompresser.

   It is important to note that the compressor does not maintain any
   timers, nor does the decompresser know when an error occurred (only
   the receiving TCP knows this, when the TCP checksum fails). A single
   bit error will cause the decompresser to lose sync, and subsequent
   packets with compressed headers will be dropped by the receiving TCP,
   since they will all fail the TCP checksum. When this happens, no
   duplicate acknowledgments will be generated, and the decompresser can
   only re-synchronize when it receives a packet with an uncompressed
   header. This means that when header compression is being used, both
   fast retransmit and selective acknowledgments will not be able
   correct packets lost on a compressed link. The "twice" algorithm,
   described below, may be a partial solution to this problem.

   [DNP99] and [DENP97] describe TCP/IPv4 and TCP/IPv6 compression
   algorithms including compressing the various IPv6 extension headers
   as well as methods for compressing non-TCP streams. [DENP97] also
   augments TCP header compression by introducing the "twice" algorithm.
   If a particular packet fails to decompress properly, the twice
   algorithm modifies its assumptions about the inferred fields in the
   compressed header, assuming that a packet identical to the current
   one was dropped between the last correctly decoded packet and the
   current one. Twice then tries to decompress the received packet
   under the new assumptions and, if the checksum passes, the packet is
   passed to IP and the decompresser state has been re-synchronized.
   This procedure can be extended to three or more decoding attempts.
   Additional robustness can be achieved by caching full copies of
   packets which don't decompress properly in the hopes that later
   arrivals will fix the problem. Finally, the performance improvement
   if the decompresser can explicitly request a full header is
   discussed. Simulation results show that twice, in conjunction with
   the full header request mechanism, can improve throughput over
   uncompressed streams.

3.7.2 Research

   [Jac90] outlines a simple header compression scheme for TCP/IP.

   In [DENP97] the authors present the results of simulations showing
   that header compression is advantageous for both low and medium
   bandwidth links. Simulations show that the twice algorithm, combined
   with an explicit header request mechanism, improved throughput by
   10-15% over uncompressed sessions across a wide range of bit error
   rates.

   Much of this improvement may have been due to the twice algorithm
   quickly re-synchronizing the decompresser when a packet is lost.
   This is because the twice algorithm, applied one or two times when
   the decompresser becomes unsynchronized, will re-sync the
   decompresser in between 83% and 99% of the cases examined. This
   means that packets received correctly after twice has resynchronized
   the decompresser will cause duplicate acknowledgments. This re-
   enables the use of both fast retransmit and SACK in conjunction with
   header compression.

3.7.3 Implementation Issues

   Implementing TCP/IP header compression requires changes at both the
   sending (compressor) and receiving (decompresser) ends of each link
   that uses compression. The twice algorithm requires very little
   extra machinery over and above header compression, while the explicit
   header request mechanism of [DENP97] requires more extensive
   modifications to the sending and receiving ends of each link that
   employs header compression. Header compression does not violate
   TCP's congestion control mechanisms and therefore can be safely
   implemented in shared networks.

3.7.4 Topology Considerations

   TCP/IP header compression is applicable to all of the environments
   discussed in section 2, but will provide relatively more improvement
   in situations where packet sizes are small (i.e., overhead is large)
   and there is medium to low bandwidth and/or higher BER. When TCP's
   congestion window size is large, implementing the explicit header
   request mechanism, the twice algorithm, and caching packets which
   fail to decompress properly becomes more critical.

3.7.5 Possible Interaction and Relationships with Other Research

   As discussed above, losing synchronization between a sender and
   receiver can cause many packet drops. The frequency of losing
   synchronization and the effectiveness of the twice algorithm may
   point to using a SACK-based loss recovery algorithm to reduce the
   impact of multiple lost segments. However, even very robust SACK-
   based algorithms may not work well if too many segments are lost.

3.8 Sharing TCP State Among Similar Connections

3.8.1 Mitigation Description

   Persistent TCP state information can be used to overcome limitations
   in the configuration of the initial state, and to automatically tune
   TCP to environments using satellite links and to coordinate multiple
   TCP connections sharing a satellite link.

   TCP includes a variety of parameters, many of which are set to
   initial values which can severely affect the performance of TCP
   connections traversing satellite links, even though most TCP
   parameters are adjusted later after the connection is established.
   These parameters include initial size of cwnd and initial MSS size.
   Various suggestions have been made to change these initial
   conditions, to more effectively support satellite links. However, it
   is difficult to select any single set of parameters which is
   effective for all environments.

   An alternative to attempting to select these parameters a-priori is
   sharing state across TCP connections and using this state when
   initializing a new connection. For example, if all connections to a
   subnet result in extended congestion windows of 1 megabyte, it is
   probably more efficient to start new connections with this value,
   than to rediscover it by requiring the cwnd to increase using slow
   start over a period of dozens of round-trip times.

3.8.2 Research

   Sharing state among connections brings up a number of questions such
   as what information to share, with whom to share, how to share it,
   and how to age shared information. First, what information is to be
   shared must be determined. Some information may be appropriate to
   share among TCP connections, while some information sharing may be
   inappropriate or not useful. Next, we need to determine with whom to
   share information. Sharing may be appropriate for TCP connections
   sharing a common path to a given host. Information may be shared
   among connections within a host, or even among connections between
   different hosts, such as hosts on the same LAN. However, sharing
   information between connections not traversing the same network may
   not be appropriate. Given the state to share and the parties that
   share it, a mechanism for the sharing is required. Simple state,
   like MSS and RTT, is easy to share, but congestion window information
   can be shared a variety of ways. The sharing mechanism determines
   priorities among the sharing connections, and a variety of fairness
   criteria need to be considered. Also, the mechanisms by which
   information is aged require further study. See RFC 2140 for a
   discussion of the security issues in both sharing state within a
   single host and sharing state among hosts on a subnet. Finally, the
   security concerns associated with sharing a piece of information need

   to be carefully considered before introducing such a mechanism. Many
   of these open research questions must be answered before state
   sharing can be widely deployed.

   The opportunity for such sharing, both among a sequence of
   connections, as well as among concurrent connections, is described in
   more detail in [Tou97]. The state management itself is largely an
   implementation issue, however what information should be shared and
   the specific ways in which the information should be shared is an
   open question.

   Sharing parts of the TCB state was originally documented in T/TCP
   [Bra92], and is used there to aggregate RTT values across connection
   instances, to provide meaningful average RTTs, even though most
   connections are expected to persist for only one RTT. T/TCP also
   shares a connection identifier, a sequence number separate from the
   window number and address/port pairs by which TCP connections are
   typically distinguished. As a result of this shared state, T/TCP
   allows a receiver to pass data in the SYN segment to the receiving
   application, prior to the completion of the three-way handshake,
   without compromising the integrity of the connection. In effect, this
   shared state caches a partial handshake from the previous connection,
   which is a variant of the more general issue of TCB sharing.

   Sharing state among connections (including transfers using non-TCP
   protocols) is further investigated in [BRS99].

3.8.3 Implementation Issues

   Sharing TCP state across connections requires changes to the sender's
   TCP stack, and possibly the receiver's TCP stack (as in the case of
   T/TCP, for example). Sharing TCP state may make a particular TCP
   connection more aggressive. However, the aggregate traffic should be
   more conservative than a group of independent TCP connections.
   Therefore, sharing TCP state should be safe for use in shared
   networks. Note that state sharing does not present any new security
   problems within multiuser hosts. In such a situation, users can
   steal network resources from one another with or without state
   sharing.

3.8.4 Topology Considerations

   It is expected that sharing state across TCP connections may be
   useful in all network environments presented in section 2.

3.8.5 Possible Interaction and Relationships with Other Research

   The state sharing outlined above is very similar to the Congestion
   Manager proposal [BRS99] that attempts to share congestion control
   information among both TCP and UDP flows between a pair of hosts.

3.9 ACK Congestion Control

   In highly asymmetric networks, a low-speed return link can restrict
   the performance of the data flow on a high-speed forward link by
   limiting the flow of acknowledgments returned to the data sender.
   For example, if the data sender uses 1500 byte segments, and the
   receiver generates 40 byte acknowledgments (IPv4, TCP without
   options), the reverse link will congest with ACKs for asymmetries of
   more than 75:1 if delayed ACKs are used, and 37:1 if every segment is
   acknowledged. For a 1.5 Mb/second data link, ACK congestion will
   occur for reverse link speeds below 20 kilobits/sec. These levels of
   asymmetry will readily occur if the reverse link is shared among
   multiple satellite receivers, as is common in many VSAT satellite
   networks. If a terrestrial modem link is used as a reverse link, ACK
   congestion is also likely, especially as the speed of the forward
   link is increased. Current congestion control mechanisms are aimed
   at controlling the flow of data segments, but do not affect the flow
   of ACKs.

   In [KVR98] the authors point out that the flow of acknowledgments can
   be restricted on the low-speed link not only by the bandwidth of the
   link, but also by the queue length of the router. The router may
   limit its queue length by counting packets, not bytes, and therefore
   begin discarding ACKs even if there is enough bandwidth to forward
   them.

3.9.1 Mitigation Description

   ACK Congestion Control extends the concept of flow control for data
   segments to acknowledgment segments. In the method described in
   [BPK97], any intermediate router can mark an acknowledgment with an
   Explicit Congestion Notification (ECN) bit once the queue occupancy
   in the router exceeds a given threshold. The data sender (which
   receives the acknowledgment) must "echo" the ECN bit back to the data
   receiver (see section 3.3.3 for a more detailed discussion of ECN).
   The proposed algorithm for marking ACK segments with an ECN bit is
   Random Early Detection (RED) [FJ93]. In response to the receipt of
   ECN marked data segments, the receiver will dynamically reduce the
   rate of acknowledgments using a multiplicative backoff. Once
   segments without ECN are received, the data receiver speeds up
   acknowledgments using a linear increase, up to a rate of either 1 (no

delayed ACKs) or 2 (normal delayed ACKs) data segments per ACK. The
authors suggest that an ACK be generated at least once per window,
and ideally a few times per window.

As in the RED congestion control mechanism for data flow, the
bottleneck gateway can randomly discard acknowledgments, rather than
marking them with an ECN bit, once the queue fills beyond a given
threshold.

3.9.2 Research

[BPK97] analyze the effect of ACK Congestion Control (ACC) on the
performance of an asymmetric network. They note that the use of ACC,
and indeed the use of any scheme which reduces the frequency of
acknowledgments, has potential unwanted side effects. Since each ACK
will acknowledge more than the usual one or two data segments, the
likelihood of segment bursts from the data sender is increased. In
addition, congestion window growth may be impeded if the receiver
grows the window by counting received ACKs, as mandated by
[Ste97,APS99]. The authors therefore combine ACC with a series of
modifications to the data sender, referred to as TCP Sender
Adaptation (SA). SA combines a limit on the number of segments sent
in a burst, regardless of window size. In addition, byte counting
(as opposed to ACK counting) is employed for window growth. Note
that byte counting has been studied elsewhere and can introduce
side-effects, as well [All98].

The results presented in [BPK97] indicate that using ACC and SA will
reduce the bursts produced by ACK losses in unmodified (Reno) TCP.
In cases where these bursts would lead to data loss at an
intermediate router, the ACC and SA modification significantly
improve the throughput for a single data transfer. The results
further suggest that the use of ACC and SA significantly improve
fairness between two simultaneous transfers.

ACC is further reported to prevent the increase in round trip time
(RTT) that occurs when an unmodified TCP fills the reverse router
queue with acknowledgments.

In networks where the forward direction is expected to suffer losses
in one of the gateways, due to queue limitations, the authors report
at best a very slight improvement in performance for ACC and SA,
compared to unmodified Reno TCP.

3.9.3 Implementation Issues

   Both ACC and SA require modification of the sending and receiving
   hosts, as well as the bottleneck gateway. The current research
   suggests that implementing ACC without the SA modifications results
   in a data sender which generates potentially disruptive segment
   bursts. It should be noted that ACC does require host modifications
   if it is implemented in the way proposed in [BPK97]. The authors
   note that ACC can be implemented by discarding ACKs (which requires
   only a gateway modification, but no changes in the hosts), as opposed
   to marking them with ECN. Such an implementation may, however,
   produce bursty data senders if it is not combined with a burst
   mitigation technique. ACC requires changes to the standard ACKing
   behavior of a receiving TCP and therefore is not recommended for use
   in shared networks.

3.9.4 Topology Considerations

   Neither ACC nor SA require the storage of state in the gateway.
   These schemes should therefore be applicable for all topologies,
   provided that the hosts using the satellite or hybrid network can be
   modified. However, these changes are expected to be especially
   beneficial to networks containing asymmetric satellite links.

3.9.5 Possible Interaction and Relationships with Other Research

   Note that ECN is a pre-condition for using ACK congestion control.
   Additionally, the ACK Filtering algorithm discussed in the next
   section attempts to solve the same problem as ACC. Choosing between
   the two algorithms (or another mechanism) is currently an open
   research question.

3.10 ACK Filtering

   ACK Filtering (AF) is designed to address the same ACK congestion
   effects described in 3.9. Contrary to ACC, however, AF is designed
   to operate without host modifications.

3.10.1 Mitigation Description

   AF takes advantage of the cumulative acknowledgment structure of TCP.
   The bottleneck router in the reverse direction (the low speed link)
   must be modified to implement AF. Upon receipt of a segment which
   represents a TCP acknowledgment, the router scans the queue for
   redundant ACKs for the same connection, i.e. ACKs which acknowledge
   portions of the window which are included in the most recent ACK.
   All of these "earlier" ACKs are removed from the queue and discarded.

   The router does not store state information, but does need to
   implement the additional processing required to find and remove
   segments from the queue upon receipt of an ACK.

3.10.2 Research

   [BPK97] analyzes the effects of AF. As is the case in ACC, the use
   of ACK filtering alone would produce significant sender bursts, since
   the ACKs will be acknowledging more previously-unacknowledged data.
   The SA modifications described in 3.9.2 could be used to prevent
   those bursts, at the cost of requiring host modifications. To
   prevent the need for modifications in the TCP stack, AF is more
   likely to be paired with the ACK Reconstruction (AR) technique, which
   can be implemented at the router where segments exit the slow reverse
   link.

   AR inspects ACKs exiting the link, and if it detects large "gaps" in
   the ACK sequence, it generates additional ACKs to reconstruct an
   acknowledgment flow which more closely resembles what the data sender
   would have seen had ACK Filtering not been introduced. AR requires
   two parameters; one parameter is the desired ACK frequency, while the
   second controls the spacing, in time, between the release of
   consecutive reconstructed ACKs.

   In [BPK97], the authors show the combination of AF and AR to increase
   throughput, in the networks studied, over both unmodified TCP and the
   ACC/SA modifications. Their results also strongly suggest that the
   use of AF alone, in networks where congestion losses are expected,
   decreases performance (even below the level of unmodified TCP Reno)
   due to sender bursting.

   AF delays acknowledgments from arriving at the receiver by dropping
   earlier ACKs in favor of later ACKs. This process can cause a slight
   hiccup in the transmission of new data by the TCP sender.

3.10.3 Implementation Issues

   Both ACK Filtering and ACK Reconstruction require only router
   modification. However, the implementation of AR requires some
   storage of state information in the exit router. While AF does not
   require storage of state information, its use without AR (or SA)
   could produce undesired side effects. Furthermore, more research is
   required regarding appropriate ranges for the parameters needed in
   AR.

3.10.4 Topology Considerations

   AF and AR appear applicable to all topologies, assuming that the
   storage of state information in AR does not prove to be prohibitive
   for routers which handle large numbers of flows. The fact that TCP
   stack modifications are not required for AF/AR makes this approach
   attractive for hybrid networks and networks with diverse types of
   hosts. These modifications, however, are expected to be most
   beneficial in asymmetric network paths.

   On the other hand, the implementation of AF/AR requires the routers
   to examine the TCP header, which prohibits their use in secure
   networks where IPSEC is deployed. In such networks, AF/AR can be
   effective only inside the security perimeter of a private, or virtual
   private network, or in private networks where the satellite link is
   protected only by link-layer encryption (as opposed to IPSEC). ACK
   Filtering is safe to use in shared networks (from a congestion
   control point-of-view), as the number of ACKs can only be reduced,
   which makes TCP less aggressive. However, note that while TCP is
   less aggressive, the delays that AF induces (outlined above) can lead
   to larger bursts than would otherwise occur.

3.10.5 Possible Interaction and Relationships with Other Research

   ACK Filtering attempts to solve the same problem as ACK Congestion
   Control (as outlined in section 3.9). Which of the two algorithms is
   more appropriate is currently an open research question.

4   Conclusions

   This document outlines TCP items that may be able to mitigate the
   performance problems associated with using TCP in networks containing
   satellite links. These mitigations are not IETF standards track
   mechanisms and require more study before being recommended by the
   IETF. The research community is encouraged to examine the above
   mitigations in an effort to determine which are safe for use in
   shared networks such as the Internet.

5   Security Considerations

   Several of the above sections noted specific security concerns which
   a given mitigation aggravates.

   Additionally, any form of wireless communication link is more
   susceptible to eavesdropping security attacks than standard wire-
   based links due to the relative ease with which an attacker can watch
   the network and the difficultly in finding attackers monitoring the
   network.

6   Acknowledgments

   Our thanks to Aaron Falk and Sally Floyd, who provided very helpful
   comments on drafts of this document.

7   References

   [AFP98]   Allman, M., Floyd, S. and C. Partridge, "Increasing TCP's
              Initial Window", RFC 2414, September 1998.

   [AGS99]   Allman, M., Glover, D. and L. Sanchez, "Enhancing TCP Over
              Satellite Channels using Standard Mechanisms", BCP 28, RFC
              2488, January 1999.

   [AHKO97]   Mark Allman, Chris Hayes, Hans Kruse, Shawn Ostermann. TCP
              Performance Over Satellite Links. In Proceedings of the
              5th International Conference on Telecommunication Systems,
              March 1997.

   [AHO98]   Mark Allman, Chris Hayes, Shawn Ostermann. An Evaluation
              of TCP with Larger Initial Windows. Computer Communication
              Review, 28(3), July 1998.

   [AKO96]   Mark Allman, Hans Kruse, Shawn Ostermann. An Application-
              Level Solution to TCP's Satellite Inefficiencies. In
              Proceedings of the First International Workshop on
              Satellite-based Information Services (WOSBIS), November
              1996.

   [All97a]   Mark Allman. Improving TCP Performance Over Satellite
              Channels. Master's thesis, Ohio University, June 1997.

   [All97b]   Mark Allman. Fixing Two BSD TCP Bugs. Technical Report
              CR-204151, NASA Lewis Research Center, October 1997.

   [All98]   Mark Allman. On the Generation and Use of TCP
              Acknowledgments. ACM Computer Communication Review, 28(5),
              October 1998.

   [AOK95]   Mark Allman, Shawn Ostermann, Hans Kruse. Data Transfer
              Efficiency Over Satellite Circuits Using a Multi-Socket
              Extension to the File Transfer Protocol (FTP). In
              Proceedings of the ACTS Results Conference, NASA Lewis
              Research Center, September 1995.

   [AP99]   Mark Allman, Vern Paxson. On Estimating End-to-End Network
              Path Properties. ACM SIGCOMM, September 1999.

   [APS99]   Allman, M., Paxson, V. and W. Richard Stevens, "TCP
              Congestion Control", RFC 2581, April 1999.

   [BCC+98]   Braden, B., Clark, D., Crowcroft, J., Davie, B., Deering,
              S., Estrin, D., Floyd, S., Jacobson, V., Minshall, G.,
              Partridge, C., Peterson, L., Ramakrishnan, K., Shenker, S.,
              Wroclawski, J. and L. Zhang, "Recommendations on Queue
              Management and Congestion Avoidance in the Internet", RFC
              2309, April 1998.

   [BKVP97]   B. Bakshi and P. Krishna and N. Vaidya and D. Pradham,
              "Improving Performance of TCP over Wireless Networks", 17th
              International Conference on Distributed Computing Systems
              (ICDCS), May 1997.

   [BPK97]   Hari Balakrishnan, Venkata N. Padmanabhan, and Randy H.
              Katz. The Effects of Asymmetry on TCP Performance. In
              Proceedings of the ACM/IEEE Mobicom, Budapest, Hungary,
              ACM. September, 1997.

   [BPK98]   Hari Balakrishnan, Venkata Padmanabhan, Randy H. Katz. The
              Effects of Asymmetry on TCP Performance. ACM Mobile
              Networks and Applications (MONET), 1998 (to appear).

   [BPSK96]   H. Balakrishnan and V. Padmanabhan and S. Sechan and R.
              Katz, "A Comparison of Mechanisms for Improving TCP
              Performance over Wireless Links", ACM SIGCOMM, August 1996.

   [Bra89]   Braden, R., "Requirements for Internet Hosts --
              Communication Layers", STD 3, RFC 1122, October 1989.

   [Bra92]   Braden, R., "Transaction TCP -- Concepts", RFC 1379,
              September 1992.

   [Bra94]   Braden, R., "T/TCP -- TCP Extensions for Transactions:
              Functional Specification", RFC 1644, July 1994.

   [BRS99]   Hari Balakrishnan, Hariharan Rahul, and Srinivasan Seshan.
              An Integrated Congestion Management Architecture for
              Internet Hosts. ACM SIGCOMM, September 1999.

   [ddKI99]   M. deVivo, G.O. deVivo, R. Koeneke, G. Isern. Internet
              Vulnerabilities Related to TCP/IP and T/TCP. Computer
              Communication Review, 29(1), January 1999.

[DENP97]   Mikael Degermark, Mathias Engan, Bjorn Nordgren, Stephen
           Pink. Low-Loss TCP/IP Header Compression for Wireless
           Networks. ACM/Baltzer Journal on Wireless Networks, vol.3,
           no.5, p. 375-87.

[DMT96]   R. C. Durst and G. J. Miller and E. J. Travis, "TCP
           Extensions for Space Communications", Mobicom 96, ACM, USA,
           1996.

[DNP99]   Degermark, M., Nordgren, B. and S. Pink, "IP Header
           Compression", RFC 2507, February 1999.

[FF96]   Kevin Fall, Sally Floyd. Simulation-based Comparisons of
           Tahoe, Reno, and SACK TCP. Computer Communication Review,
           V. 26 N. 3, July 1996, pp. 5-21.

[FF99]   Sally Floyd, Kevin Fall. Promoting the Use of End-to-End
           Congestion Control in the Internet, IEEE/ACM Transactions
           on Networking, August 1999.

[FH99]   Floyd, S. and T. Henderson, "The NewReno Modification to
           TCP's Fast Recovery Algorithm", RFC 2582, April 1999.

[FJ93]   Sally Floyd and Van Jacobson. Random Early Detection
           Gateways for Congestion Avoidance, IEEE/ACM Transactions on
           Networking, V. 1 N. 4, August 1993.

[Flo91]   Sally Floyd. Connections with Multiple Congested Gateways
           in Packet-Switched Networks, Part 1: One-way Traffic. ACM
           Computer Communications Review, V. 21, N. 5, October 1991.

[Flo94]   Sally Floyd. TCP and Explicit Congestion Notification, ACM
           Computer Communication Review, V. 24 N. 5, October 1994.

[Flo99]   Sally Floyd. "Re: TCP and out-of-order delivery", email to
           end2end-interest mailing list, February, 1999.

[Hah94]   Jonathan Hahn. MFTP: Recent Enhancements and Performance
           Measurements. Technical Report RND-94-006, NASA Ames
           Research Center, June 1994.

[Hay97]   Chris Hayes. Analyzing the Performance of New TCP
           Extensions Over Satellite Links. Master's Thesis, Ohio
           University, August 1997.

[HK98]   Tom Henderson, Randy Katz. On Improving the Fairness of
           TCP Congestion Avoidance. Proceedings of IEEE Globecom '98
           Conference, 1998.

[HK99]   Tim Henderson, Randy Katz. Transport Protocols for
            Internet-Compatible Satellite Networks, IEEE Journal on
            Selected Areas of Communications, February, 1999.

[Hoe95]   J. Hoe, Startup Dynamics of TCP's Congestion Control and
            Avoidance Schemes. Master's Thesis, MIT, 1995.

[Hoe96]   Janey Hoe. Improving the Startup Behavior of a Congestion
            Control Scheme for TCP. In ACM SIGCOMM, August 1996.

[IL92]   David Iannucci and John Lakashman. MFTP: Virtual TCP
            Window Scaling Using Multiple Connections. Technical
            Report RND-92-002, NASA Ames Research Center, January 1992.

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

[Jac90]   Jacobson, V., "Compressing TCP/IP Headers", RFC 1144,
            February 1990.

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

[JK92]   Van Jacobson and Mike Karels. Congestion Avoidance and
            Control. Originally appearing in the proceedings of
            SIGCOMM '88 by Jacobson only, this revised version includes
            an additional appendix. The revised version is available
            at ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z. 1992.

[Joh95]   Stacy Johnson. Increasing TCP Throughput by Using an
            Extended Acknowledgment Interval. Master's Thesis, Ohio
            University, June 1995.

[KAGT98]   Hans Kruse, Mark Allman, Jim Griner, Diepchi Tran. HTTP
            Page Transfer Rates Over Geo-Stationary Satellite Links.
            March 1998. Proceedings of the Sixth International
            Conference on Telecommunication Systems.

[Kes91]   Srinivasan Keshav. A Control Theoretic Approach to Flow
            Control. In ACM SIGCOMM, September 1991.

[KM97]   S. Keshav, S. Morgan. SMART Retransmission: Performance
            with Overload and Random Losses. Proceeding of Infocom.
            1997.

   [KVR98]   Lampros Kalampoukas, Anujan Varma, and K. K.Ramakrishnan.
              Improving TCP Throughput Over Two-Way Asymmetric Links:
              Analysis and Solutions. Measurement and Modeling of
              Computer Systems, 1998, Pages 78-89.

   [MM96a]   M. Mathis, J. Mahdavi, "Forward Acknowledgment: Refining
              TCP Congestion Control," Proceedings of SIGCOMM'96, August,
              1996, Stanford, CA. Available from
              http://www.psc.edu/networking/papers/papers.html

   [MM96b]   M. Mathis, J. Mahdavi, "TCP Rate-Halving with Bounding
              Parameters" Available from
              http://www.psc.edu/networking/papers/FACKnotes/current.

   [MMFR96]   Mathis, M., Mahdavi, J., Floyd, S. and A. Romanow, "TCP
              Selective Acknowledgment Options", RFC 2018, October 1996.

   [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. Available from
              http://www.psc.edu/networking/papers/papers.html

   [MV98]   Miten N. Mehta and Nitin H. Vaidya. Delayed Duplicate-
              Acknowledgments: A Proposal to Improve Performance of TCP
              on Wireless Links. Technical Report 98-006, Department of
              Computer Science, Texas A&M University, February 1998.

   [Nic97]   Kathleen Nichols. Improving Network Simulation with
              Feedback. Com21, Inc. Technical Report. Available from
              http://www.com21.com/pages/papers/068.pdf.

   [PADHV99] Paxson, V., Allman, M., Dawson, S., Heavens, I. and B.
              Volz, "Known TCP Implementation Problems", RFC 2525, March
              1999.

   [Pax97]   Vern Paxson. Automated Packet Trace Analysis of TCP
              Implementations. In Proceedings of ACM SIGCOMM, September
              1997.

   [PN98]   Poduri, K. and K. Nichols, "Simulation Studies of Increased
              Initial TCP Window Size", RFC 2415, September 1998.

   [Pos81]   Postel, J., "Transmission Control Protocol", STD 7, RFC
              793, September 1981.

   [RF99]   Ramakrishnan, K. and S. Floyd, "A Proposal to add Explicit
               Congestion Notification (ECN) to IP", RFC 2481, January
               1999.

   [SF98]   Nihal K. G. Samaraweera and Godred Fairhurst,
               "Reinforcement of TCP error Recovery for Wireless
               Communication", Computer Communication Review, volume 28,
               number 2, April 1998.

   [SP98]   Shepard, T. and C. Partridge, "When TCP Starts Up With Four
               Packets Into Only Three Buffers", RFC 2416, September 1998.

   [Ste97]   Stevens, W., "TCP Slow Start, Congestion Avoidance, Fast
               Retransmit, and Fast Recovery Algorithms", RFC 2001,
               January 1997.

   [Sut98]   B. Suter, T. Lakshman, D. Stiliadis, and A. Choudhury.
               Design Considerations for Supporting TCP with Per-flow
               Queueing. Proceedings of IEEE Infocom '98 Conference,
               1998.

   [Tou97]   Touch, J., "TCP Control Block Interdependence", RFC 2140,
               April 1997.

   [VH97a]   Vikram Visweswaraiah and John Heidemann. Improving Restart
               of Idle TCP Connections. Technical Report 97-661,
               University of Southern California, 1997.

   [VH97b]   Vikram Visweswaraiah and John Heidemann. Rate-based pacing
               Source Code Distribution, Web page:
               http://www.isi.edu/lsam/publications/rate_based_pacing/README.html
               November, 1997.

   [VH98]   Vikram Visweswaraiah and John Heidemann. Improving Restart
               of Idle TCP Connections (revised). Submitted for
               publication.

8   Authors' Addresses

Mark Allman
NASA Glenn Research Center/BBN Technologies
Lewis Field
21000 Brookpark Rd. MS 54-2
Cleveland, OH 44135

EMail: mallman@grc.nasa.gov
http://roland.grc.nasa.gov/~mallman

Spencer Dawkins
Nortel
P.O.Box 833805
Richardson, TX 75083-3805

EMail: Spencer.Dawkins.sdawkins@nt.com

Dan Glover
NASA Glenn Research Center
Lewis Field
21000 Brookpark Rd. MS 3-6
Cleveland, OH 44135

EMail: Daniel.R.Glover@grc.nasa.gov
http://roland.grc.nasa.gov/~dglover

Jim Griner
NASA Glenn Research Center
Lewis Field
21000 Brookpark Rd. MS 54-2
Cleveland, OH 44135

EMail: jgriner@grc.nasa.gov
http://roland.grc.nasa.gov/~jgriner

Diepchi Tran
NASA Glenn Research Center
Lewis Field
21000 Brookpark Rd. MS 54-2
Cleveland, OH 44135

EMail: dtran@grc.nasa.gov

Tom Henderson
University of California at Berkeley
Phone: +1 (510) 642-8919

EMail: tomh@cs.berkeley.edu
URL: http://www.cs.berkeley.edu/~tomh/

John Heidemann
University of Southern California/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292-6695

EMail: johnh@isi.edu

Joe Touch
University of Southern California/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292-6601
USA

Phone: +1 310-448-9151
Fax: +1 310-823-6714
URL: http://www.isi.edu/touch
EMail: touch@isi.edu

Hans Kruse
J. Warren McClure School of Communication Systems Management
Ohio University
9 S. College Street
Athens, OH 45701

Phone: 740-593-4891
Fax: 740-593-4889
EMail: hkruse1@ohiou.edu
http://www.csm.ohiou.edu/kruse

Shawn Ostermann
School of Electrical Engineering and Computer Science
Ohio University
416 Morton Hall
Athens, OH 45701

Phone: (740) 593-1234
EMail: ostermann@cs.ohiou.edu

Keith Scott
The MITRE Corporation
M/S W650
1820 Dolley Madison Blvd.
McLean VA 22102-3481

EMail: kscott@mitre.org

Jeffrey Semke
Pittsburgh Supercomputing Center
4400 Fifth Ave.
Pittsburgh, PA 15213

EMail: semke@psc.edu
http://www.psc.edu/~semke

9 Full Copyright Statement

Acknowledgement