       The NewReno Modification to TCP's Fast Recovery Algorithm

Abstract

   RFC 5681 documents the following four intertwined TCP congestion
   control algorithms: slow start, congestion avoidance, fast
   retransmit, and fast recovery. RFC 5681 explicitly allows certain
   modifications of these algorithms, including modifications that use
   the TCP Selective Acknowledgment (SACK) option (RFC 2883), and
   modifications that respond to "partial acknowledgments" (ACKs that
   cover new data, but not all the data outstanding when loss was
   detected) in the absence of SACK. This document describes a specific
   algorithm for responding to partial acknowledgments, referred to as
   "NewReno". This response to partial acknowledgments was first
   proposed by Janey Hoe. This document obsoletes RFC 3782.

Copyright Notice

1. Introduction

   For the typical implementation of the TCP fast recovery algorithm
   described in [RFC5681] (first implemented in the 1990 BSD Reno
   release, and referred to as the "Reno algorithm" in [FF96]), the TCP
   data sender only retransmits a packet after a retransmit timeout has
   occurred, or after three duplicate acknowledgments have arrived
   triggering the fast retransmit algorithm. A single retransmit
   timeout might result in the retransmission of several data packets,
   but each invocation of the fast retransmit algorithm in RFC 5681
   leads to the retransmission of only a single data packet.

   Two problems arise with Reno TCP when multiple packet losses occur in
   a single window. First, Reno will often take a timeout, as has been
   documented in [Hoe95]. Second, even if a retransmission timeout is
   avoided, multiple fast retransmits and window reductions can occur,
   as documented in [F94]. When multiple packet losses occur, if the
   SACK option [RFC2883] is available, the TCP sender has the
   information to make intelligent decisions about which packets to
   retransmit and which packets not to retransmit during fast recovery.

This document applies to TCP connections that are unable to use the
TCP Selective Acknowledgment (SACK) option, either because the option
is not locally supported or because the TCP peer did not indicate a
willingness to use SACK.

In the absence of SACK, there is little information available to the
TCP sender in making retransmission decisions during fast recovery.
From the three duplicate acknowledgments, the sender infers a packet
loss, and retransmits the indicated packet. After this, the data
sender could receive additional duplicate acknowledgments, as the
data receiver acknowledges additional data packets that were already
in flight when the sender entered fast retransmit.

In the case of multiple packets dropped from a single window of data,
the first new information available to the sender comes when the
sender receives an acknowledgment for the retransmitted packet (that
is, the packet retransmitted when fast retransmit was first entered).
If there is a single packet drop and no reordering, then the
acknowledgment for this packet will acknowledge all of the packets
transmitted before fast retransmit was entered. However, if there
are multiple packet drops, then the acknowledgment for the
retransmitted packet will acknowledge some but not all of the packets
transmitted before the fast retransmit. We call this acknowledgment
a partial acknowledgment.

Along with several other suggestions, [Hoe95] suggested that during
fast recovery the TCP data sender respond to a partial acknowledgment
by inferring that the next in-sequence packet has been lost and
retransmitting that packet. This document describes a modification
to the fast recovery algorithm in RFC 5681 that incorporates a
response to partial acknowledgments received during fast recovery.
We call this modified fast recovery algorithm NewReno, because it is
a slight but significant variation of the behavior that has been
historically referred to as Reno. This document does not discuss the
other suggestions in [Hoe95] and [Hoe96], such as a change to the
ssthresh parameter during slow start, or the proposal to send a new
packet for every two duplicate acknowledgments during fast recovery.
The version of NewReno in this document also draws on other
discussions of NewReno in the literature [LM97] [Hen98].

We do not claim that the NewReno version of fast recovery described
here is an optimal modification of fast recovery for responding to
partial acknowledgments, for TCP connections that are unable to use
SACK. Based on our experiences with the NewReno modification in the
network simulator known as ns-2 [NS] and with numerous
implementations of NewReno, we believe that this modification
improves the performance of the fast retransmit and fast recovery

algorithms in a wide variety of scenarios. Previous versions of this
RFC [RFC2582] [RFC3782] provide simulation-based evidence of the
possible performance gains.

2. Terminology and Definitions

This document assumes that the reader is familiar with the terms
SENDER MAXIMUM SEGMENT SIZE (SMSS), CONGESTION WINDOW (cwnd), and
FLIGHT SIZE (FlightSize) defined in [RFC5681].

This document defines an additional sender-side state variable called
"recover":

   recover:
       When in fast recovery, this variable records the send sequence
       number that must be acknowledged before the fast recovery
       procedure is declared to be over.

3. The Fast Retransmit and Fast Recovery Algorithms in NewReno

3.1. Protocol Overview

The basic idea of these extensions to the fast retransmit and fast
recovery algorithms described in Section 3.2 of [RFC5681] is as
follows. The TCP sender can infer, from the arrival of duplicate
acknowledgments, whether multiple losses in the same window of data
have most likely occurred, and avoid taking a retransmit timeout or
making multiple congestion window reductions due to such an event.

The NewReno modification applies to the fast recovery procedure that
begins when three duplicate ACKs are received and ends when either a
retransmission timeout occurs or an ACK arrives that acknowledges all
of the data up to and including the data that was outstanding when
the fast recovery procedure began.

3.2. Specification

   The procedures specified in Section 3.2 of [RFC5681] are followed,
   with the modifications listed below. Note that this specification
   avoids the use of the key words defined in RFC 2119 [RFC2119], since
   it mainly provides sender-side implementation guidance for
   performance improvement, and does not affect interoperability.

   1) Initialization of TCP protocol control block:
       When the TCP protocol control block is initialized, recover is
       set to the initial send sequence number.

   2) Three duplicate ACKs:
       When the third duplicate ACK is received, the TCP sender first
       checks the value of recover to see if the Cumulative
       Acknowledgment field covers more than recover. If so, the value
       of recover is incremented to the value of the highest sequence
       number transmitted by the TCP so far. The TCP then enters fast
       retransmit (step 2 of Section 3.2 of [RFC5681]). If not, the TCP
       does not enter fast retransmit and does not reset ssthresh.

   3) Response to newly acknowledged data:
       Step 6 of [RFC5681] specifies the response to the next ACK that
       acknowledges previously unacknowledged data. When an ACK arrives
       that acknowledges new data, this ACK could be the acknowledgment
       elicited by the initial retransmission from fast retransmit, or
       elicited by a later retransmission. There are two cases:

       Full acknowledgments:
       If this ACK acknowledges all of the data up to and including
       recover, then the ACK acknowledges all the intermediate segments
       sent between the original transmission of the lost segment and
       the receipt of the third duplicate ACK. Set cwnd to either (1)
       min (ssthresh, max(FlightSize, SMSS) + SMSS) or (2) ssthresh,
       where ssthresh is the value set when fast retransmit was entered,
       and where FlightSize in (1) is the amount of data presently
       outstanding. This is termed "deflating" the window. If the
       second option is selected, the implementation is encouraged to
       take measures to avoid a possible burst of data, in case the
       amount of data outstanding in the network is much less than the
       new congestion window allows. A simple mechanism is to limit the
       number of data packets that can be sent in response to a single
       acknowledgment. Exit the fast recovery procedure.

      Partial acknowledgments:
      If this ACK does *not* acknowledge all of the data up to and
      including recover, then this is a partial ACK. In this case,
      retransmit the first unacknowledged segment. Deflate the
      congestion window by the amount of new data acknowledged by the
      Cumulative Acknowledgment field. If the partial ACK acknowledges
      at least one SMSS of new data, then add back SMSS bytes to the
      congestion window. This artificially inflates the congestion
      window in order to reflect the additional segment that has left
      the network. Send a new segment if permitted by the new value of
      cwnd. This "partial window deflation" attempts to ensure that,
      when fast recovery eventually ends, approximately ssthresh amount
      of data will be outstanding in the network. Do not exit the fast
      recovery procedure (i.e., if any duplicate ACKs subsequently
      arrive, execute step 4 of Section 3.2 of [RFC5681]).

      For the first partial ACK that arrives during fast recovery, also
      reset the retransmit timer. Timer management is discussed in
      more detail in Section 4.

   4) Retransmit timeouts:
      After a retransmit timeout, record the highest sequence number
      transmitted in the variable recover, and exit the fast recovery
      procedure if applicable.

Step 2 above specifies a check that the Cumulative Acknowledgment
field covers more than recover. Because the acknowledgment field
contains the sequence number that the sender next expects to receive,
the acknowledgment "ack_number" covers more than recover when

   ack_number - 1 > recover;

i.e., at least one byte more of data is acknowledged beyond the
highest byte that was outstanding when fast retransmit was last
entered.

Note that in step 3 above, the congestion window is deflated after a
partial acknowledgment is received. The congestion window was likely
to have been inflated considerably when the partial acknowledgment
was received. In addition, depending on the original pattern of
packet losses, the partial acknowledgment might acknowledge nearly a
window of data. In this case, if the congestion window was not
deflated, the data sender might be able to send nearly a window of
data back-to-back.

This document does not specify the sender's response to duplicate
ACKs when the fast retransmit/fast recovery algorithm is not invoked.
This is addressed in other documents, such as those describing the
Limited Transmit procedure [RFC3042]. This document also does not
address issues of adjusting the duplicate acknowledgment threshold,
but assumes the threshold specified in the IETF standards; the
current standard is [RFC5681], which specifies a threshold of three
duplicate acknowledgments.

As a final note, we would observe that in the absence of the SACK
option, the data sender is working from limited information. When
the issue of recovery from multiple dropped packets from a single
window of data is of particular importance, the best alternative
would be to use the SACK option.

4. Handling Duplicate Acknowledgments after a Timeout

   After each retransmit timeout, the highest sequence number
   transmitted so far is recorded in the variable recover. If, after a
   retransmit timeout, the TCP data sender retransmits three consecutive
   packets that have already been received by the data receiver, then
   the TCP data sender will receive three duplicate acknowledgments that
   do not cover more than recover. In this case, the duplicate
   acknowledgments are not an indication of a new instance of
   congestion. They are simply an indication that the sender has
   unnecessarily retransmitted at least three packets.

   However, when a retransmitted packet is itself dropped, the sender
   can also receive three duplicate acknowledgments that do not cover
   more than recover. In this case, the sender would have been better
   off if it had initiated fast retransmit. For a TCP sender that
   implements the algorithm specified in Section 3.2 of this document,
   the sender does not infer a packet drop from duplicate
   acknowledgments in this scenario. As always, the retransmit timer is
   the backup mechanism for inferring packet loss in this case.

   There are several heuristics, based on timestamps or on the amount of
   advancement of the Cumulative Acknowledgment field, that allow the
   sender to distinguish, in some cases, between three duplicate
   acknowledgments following a retransmitted packet that was dropped,
   and three duplicate acknowledgments from the unnecessary
   retransmission of three packets [Gur03] [GF04]. The TCP sender may
   use such a heuristic to decide to invoke a fast retransmit in some
   cases, even when the three duplicate acknowledgments do not cover
   more than recover.

For example, when three duplicate acknowledgments are caused by the
unnecessary retransmission of three packets, this is likely to be
accompanied by the Cumulative Acknowledgment field advancing by at
least four segments. Similarly, a heuristic based on timestamps uses
the fact that when there is a hole in the sequence space, the
timestamp echoed in the duplicate acknowledgment is the timestamp of
the most recent data packet that advanced the Cumulative
Acknowledgment field [RFC1323]. If timestamps are used, and the
sender stores the timestamp of the last acknowledged segment, then
the timestamp echoed by duplicate acknowledgments can be used to
distinguish between a retransmitted packet that was dropped and three
duplicate acknowledgments from the unnecessary retransmission of
three packets.

4.1. ACK Heuristic

If the ACK-based heuristic is used, then following the advancement of
the Cumulative Acknowledgment field, the sender stores the value of
the previous cumulative acknowledgment as prev_highest_ack, and
stores the latest cumulative ACK as highest_ack. In addition, the
following check is performed if, in step 2 of Section 3.2, the
Cumulative Acknowledgment field does not cover more than recover.

   2*) If the Cumulative Acknowledgment field didn't cover more than
        recover, check to see if the congestion window is greater than
        SMSS bytes and the difference between highest_ack and
        prev_highest_ack is at most 4*SMSS bytes. If true, duplicate
        ACKs indicate a lost segment (enter fast retransmit).
        Otherwise, duplicate ACKs likely result from unnecessary
        retransmissions (do not enter fast retransmit).

The congestion window check serves to protect against fast retransmit
immediately after a retransmit timeout.

If several ACKs are lost, the sender can see a jump in the cumulative
ACK of more than three segments, and the heuristic can fail.
[RFC5681] recommends that a receiver should send duplicate ACKs for
every out-of-order data packet, such as a data packet received during
fast recovery. The ACK heuristic is more likely to fail if the
receiver does not follow this advice, because then a smaller number
of ACK losses are needed to produce a sufficient jump in the
cumulative ACK.

4.2. Timestamp Heuristic

   If this heuristic is used, the sender stores the timestamp of the
   last acknowledged segment. In addition, the last sentence of step 2
   in Section 3.2 of this document is replaced as follows:

   2**) If the Cumulative Acknowledgment field didn't cover more than
        recover, check to see if the echoed timestamp in the last
        non-duplicate acknowledgment equals the stored timestamp. If
        true, duplicate ACKs indicate a lost segment (enter fast
        retransmit). Otherwise, duplicate ACKs likely result from
        unnecessary retransmissions (do not enter fast retransmit).

   The timestamp heuristic works correctly, both when the receiver
   echoes timestamps, as specified by [RFC1323], and by its revision
   attempts. However, if the receiver arbitrarily echoes timestamps,
   the heuristic can fail. The heuristic can also fail if a timeout was
   spurious and returning ACKs are not from retransmitted segments.
   This can be prevented by detection algorithms such as the Eifel
   detection algorithm [RFC3522].

5. Implementation Issues for the Data Receiver

   [RFC5681] specifies that "Out-of-order data segments SHOULD be
   acknowledged immediately, in order to accelerate loss recovery".
   Neal Cardwell has noted that some data receivers do not send an
   immediate acknowledgment when they send a partial acknowledgment, but
   instead wait first for their delayed acknowledgment timer to expire
   [C98]. As [C98] notes, this severely limits the potential benefit of
   NewReno by delaying the receipt of the partial acknowledgment at the
   data sender. Echoing [RFC5681], our recommendation is that the data
   receiver send an immediate acknowledgment for an out-of-order
   segment, even when that out-of-order segment fills a hole in the
   buffer.

6. Implementation Issues for the Data Sender

   In Section 3.2, step 3 above, it is noted that implementations should
   take measures to avoid a possible burst of data when leaving fast
   recovery, in case the amount of new data that the sender is eligible
   to send due to the new value of the congestion window is large. This
   can arise during NewReno when ACKs are lost or treated as pure window
   updates, thereby causing the sender to underestimate the number of
   new segments that can be sent during the recovery procedure.
   Specifically, bursts can occur when the FlightSize is much less than
   the new congestion window when exiting from fast recovery. One
   simple mechanism to avoid a burst of data when leaving fast recovery

is to limit the number of data packets that can be sent in response
to a single acknowledgment. (This is known as "maxburst_" in ns-2
[NS].) Other possible mechanisms for avoiding bursts include rate-
based pacing, or setting the slow start threshold to the resultant
congestion window and then resetting the congestion window to
FlightSize. A recommendation on the general mechanism to avoid
excessively bursty sending patterns is outside the scope of this
document.

An implementation may want to use a separate flag to record whether
or not it is presently in the fast recovery procedure. The use of
the value of the duplicate acknowledgment counter for this purpose is
not reliable, because it can be reset upon window updates and out-of-
order acknowledgments.

When updating the Cumulative Acknowledgment field outside of fast
recovery, the state variable recover may also need to be updated in
order to continue to permit possible entry into fast recovery
(Section 3.2, step 2). This issue arises when an update of the
Cumulative Acknowledgment field results in a sequence wraparound that
affects the ordering between the Cumulative Acknowledgment field and
the state variable recover. Entry into fast recovery is only
possible when the Cumulative Acknowledgment field covers more than
the state variable recover.

It is important for the sender to respond correctly to duplicate ACKs
received when the sender is no longer in fast recovery (e.g., because
of a retransmit timeout). The Limited Transmit procedure [RFC3042]
describes possible responses to the first and second duplicate
acknowledgments. When three or more duplicate acknowledgments are
received, the Cumulative Acknowledgment field doesn't cover more than
recover, and a new fast recovery is not invoked, the sender should
follow the guidance in Section 4. Otherwise, the sender could end up
in a chain of spurious timeouts. We mention this only because
several NewReno implementations had this bug, including the
implementation in ns-2 [NS].

It has been observed that some TCP implementations enter a slow start
or congestion avoidance window updating algorithm immediately after
the cwnd is set by the equation found in Section 3.2, step 3, even
without a new external event generating the cwnd change. Note that
after cwnd is set based on the procedure for exiting fast recovery
(Section 3.2, step 3), cwnd should not be updated until a further
event occurs (e.g., arrival of an ack, or timeout) after this
adjustment.

7. Security Considerations

   [RFC5681] discusses general security considerations concerning TCP
   congestion control. This document describes a specific algorithm
   that conforms with the congestion control requirements of [RFC5681],
   and so those considerations apply to this algorithm, too. There are
   no known additional security concerns for this specific algorithm.

8. Conclusions

   This document specifies the NewReno fast retransmit and fast recovery
   algorithms for TCP. This NewReno modification to TCP can even be
   important for TCP implementations that support the SACK option,
   because the SACK option can only be used for TCP connections when
   both TCP end-nodes support the SACK option. NewReno performs better
   than Reno in a number of scenarios discussed in previous versions of
   this RFC ([RFC2582] [RFC3782]).

   A number of options for the basic algorithms presented in Section 3
   are also referenced in Appendix A of this document. These include
   the handling of the retransmission timer, the response to partial
   acknowledgments, and whether or not the sender must maintain a state
   variable called recover. Our belief is that the differences between
   these variants of NewReno are small compared to the differences
   between Reno and NewReno. That is, the important thing is to
   implement NewReno instead of Reno for a TCP connection without SACK;
   it is less important exactly which variant of NewReno is implemented.

9. Acknowledgments

   Many thanks to Anil Agarwal, Mark Allman, Armando Caro, Jeffrey Hsu,
   Vern Paxson, Kacheong Poon, Keyur Shah, and Bernie Volz for detailed
   feedback on the precursor RFCs 2582 and 3782. Jeffrey Hsu provided
   clarifications on the handling of the variable recover; these
   clarifications were applied to RFC 3782 via an erratum and are
   incorporated into the text of Section 6 of this document. Yoshifumi
   Nishida contributed a modification to the fast recovery algorithm to
   account for the case in which FlightSize is 0 when the TCP sender
   leaves fast recovery and the TCP receiver uses delayed
   acknowledgments. Alexander Zimmermann provided several suggestions
   to improve the clarity of the document.

10. References

10.1. Normative References

   [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
             Requirement Levels", BCP 14, RFC 2119, March 1997.

   [RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion
             Control", RFC 5681, September 2009.

10.2. Informative References

   [C98]   Cardwell, N., "delayed ACKs for retransmitted packets:
             ouch!". November 1998, Email to the tcpimpl mailing list,
             archived at
             <http://groups.yahoo.com/group/tcp-impl/message/1428>.

   [F94]   Floyd, S., "TCP and Successive Fast Retransmits", Technical
             report, May 1995.
             <ftp://ftp.ee.lbl.gov/papers/fastretrans.ps>.

   [FF96]   Fall, K. and S. Floyd, "Simulation-based Comparisons of
             Tahoe, Reno and SACK TCP", Computer Communication Review,
             July 1996. <ftp://ftp.ee.lbl.gov/papers/sacks.ps.Z>.

   [GF04]   Gurtov, A. and S. Floyd, "Resolving Acknowledgment
             Ambiguity in non-SACK TCP", NExt Generation Teletraffic and
             Wired/Wireless Advanced Networking (NEW2AN'04),
             February 2004. <http://www.cs.helsinki.fi/u/gurtov/
             papers/heuristics.html>.

   [Gur03]   Gurtov, A., "[Tsvwg] resolving the problem of unnecessary
             fast retransmits in go-back-N", email to the tsvwg mailing
             list, July 28, 2003. <http://www.ietf.org/mail-archive/
             web/tsvwg/current/msg04334.html>.

   [Hen98]   Henderson, T., "Re: NewReno and the 2001 Revision",
             September 1998. Email to the tcpimpl mailing list,
             archived at
             <http://groups.yahoo.com/group/tcp-impl/message/1321>.

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

   [Hoe96]   Hoe, J., "Improving the Start-up Behavior of a Congestion
             Control Scheme for TCP", ACM SIGCOMM, August 1996.
             <http://ccr.sigcomm.org/archive/1996/conf/hoe.pdf>.

    [LM97]   Lin, D. and R. Morris, "Dynamics of Random Early
               Detection", SIGCOMM 97, October 1997.

    [NS]   "The Network Simulator version 2 (ns-2)",
               <http://www.isi.edu/nsnam/ns/>.

    [RFC1323] Jacobson, V., Braden, R., and D. Borman, "TCP Extensions
               for High Performance", RFC 1323, May 1992.

    [RFC2582] Floyd, S. and T. Henderson, "The NewReno Modification to
               TCP's Fast Recovery Algorithm", RFC 2582, April 1999.

    [RFC2883] Floyd, S., Mahdavi, J., Mathis, M., and M. Podolsky, "An
               Extension to the Selective Acknowledgement (SACK) Option
               for TCP", RFC 2883, July 2000.

    [RFC3042] Allman, M., Balakrishnan, H., and S. Floyd, "Enhancing
               TCP's Loss Recovery Using Limited Transmit", RFC 3042,
               January 2001.

    [RFC3522] Ludwig, R. and M. Meyer, "The Eifel Detection Algorithm for
               TCP", RFC 3522, April 2003.

    [RFC3782] Floyd, S., Henderson, T., and A. Gurtov, "The NewReno
               Modification to TCP's Fast Recovery Algorithm", RFC 3782,
               April 2004.

Appendix A. Additional Information

   Previous versions of this RFC ([RFC2582] [RFC3782]) contained
   additional informative material on the following subjects, and may be
   consulted by readers who may want more information about possible
   variants to the algorithms and who may want references to specific
   [NS] simulations that provide NewReno test cases.

   Section 4 of [RFC3782] discusses some alternative behaviors for
   resetting the retransmit timer after a partial acknowledgment.

   Section 5 of [RFC3782] discusses some alternative behaviors for
   performing retransmission after a partial acknowledgment.

   Section 6 of [RFC3782] describes more information about the
   motivation for the sender's state variable recover.

   Section 9 of [RFC3782] introduces some NS simulation test suites for
   NewReno. In addition, references to simulation results can be found
   throughout [RFC3782].

   Section 10 of [RFC3782] provides a comparison of Reno and
   NewReno TCP.

   Section 11 of [RFC3782] lists changes relative to [RFC2582].

Appendix B. Changes Relative to RFC 3782

   In [RFC3782], the cwnd after Full ACK reception will be set to
   (1) min (ssthresh, FlightSize + SMSS) or (2) ssthresh. However, the
   first option carries a risk of performance degradation: With the
   first option, if FlightSize is zero, the result will be 1 SMSS. This
   means TCP can transmit only 1 segment at that moment, which can cause
   a delay in ACK transmission at the receiver due to a delayed ACK
   algorithm.

   The FlightSize on Full ACK reception can be zero in some situations.
   A typical example is where the sending window size during fast
   recovery is small. In this case, the retransmitted packet and new
   data packets can be transmitted within a short interval. If all
   these packets successfully arrive, the receiver may generate a Full
   ACK that acknowledges all outstanding data. Even if the window size
   is not small, loss of ACK packets or a receive buffer shortage during
   fast recovery can also increase the possibility of falling into this
   situation.

   The proposed fix in this document, which sets cwnd to at least 2*SMSS
   if the implementation uses option 1 in the Full ACK case
   (Section 3.2, step 3, option 1), ensures that the sender TCP
   transmits at least two segments on Full ACK reception.

   In addition, an erratum was reported for RFC 3782 (an editorial
   clarification to Section 8); this erratum has been addressed in
   Section 6 of this document.

   The specification text (Section 3.2 herein) was rewritten to more
   closely track Section 3.2 of [RFC5681].

   Sections 4, 5, and 9-11 of [RFC3782] were removed, and instead
   Appendix A of this document was added to back-reference this
   informative material. A few references that have no citation in the
   main body of the document have been removed.

Authors' Addresses

   Tom Henderson
   The Boeing Company

   EMail: thomas.r.henderson@boeing.com

   Sally Floyd
   International Computer Science Institute

   Phone: +1 (510) 666-2989
   EMail: floyd@acm.org
   URL: http://www.icir.org/floyd/

   Andrei Gurtov
   University of Oulu
   Centre for Wireless Communications CWC
   P.O. Box 4500
   FI-90014 University of Oulu
   Finland

   EMail: gurtov@ee.oulu.fi

   Yoshifumi Nishida
   WIDE Project
   Endo 5322
   Fujisawa, Kanagawa 252-8520
   Japan

   EMail: nishida@wide.ad.jp