                 Extending TCP for Transactions -- Concepts

Status of This Memo

   This memo provides information for the Internet community. It does
   not specify an Internet standard. Distribution of this memo is
   unlimited.

Abstract

   This memo discusses extension of TCP to provide transaction-oriented
   service, without altering its virtual-circuit operation. This
   extension would fill the large gap between connection-oriented TCP
   and datagram-based UDP, allowing TCP to efficiently perform many
   applications for which UDP is currently used. A separate memo
   contains a detailed functional specification for this proposed
   extension.

   This work was supported in part by the National Science Foundation
   under Grant Number NCR-8922231.

TABLE OF CONTENTS

1. INTRODUCTION

   The TCP protocol [STD-007] implements a virtual-circuit transport
   service that provides reliable and ordered data delivery over a
   full-duplex connection. Under the virtual circuit model, the life of
   a connection is divided into three distinct phases: (1) opening the
   connection to create a full-duplex byte stream; (2) transferring data
   in one or both directions over this stream; and (3) closing the
   connection. Remote login and file transfer are examples of
   applications that are well suited to virtual-circuit service.

   Distributed applications, which are becoming increasingly numerous
   and sophisticated in the Internet, tend to use a transaction-oriented
   rather than a virtual circuit style of communication. Currently, a
   transaction-oriented Internet application must choose to suffer the
   overhead of opening and closing TCP connections or else build an
   application-specific transport mechanism on top of the connectionless
   transport protocol UDP. Greater convenience, uniformity, and
   efficiency would result from widely-available kernel implementations
   of a transport protocol supporting a transaction service model [RFC-
   955].

   The transaction service model has the following features:

   *   The fundamental interaction is a request followed by a response.

   *   An explicit open or close phase would impose excessive overhead.

   *   At-most-once semantics is required; that is, a transaction must
        not be "replayed" by a duplicate request packet.

   *   In favorable circumstances, a reliable request/response
        handshake can be performed with exactly one packet in each
        direction.

   *   The minimum transaction latency for a client is RTT + SPT, where
        RTT is the round-trip time and SPT is the server processing
        time.

   We use the term "transaction transport protocol" for a transport-
   layer protocol that follows this model [RFC-955].

   The Internet architecture allows an arbitrary collection of transport
   protocols to be defined on top of the minimal end-to-end datagram
   service provided by IP [Clark88]. In practice, however, production
   systems implement only TCP and UDP at the transport layer. It has
   proven difficult to leverage a new transport protocol into place, to
   be widely enough available to be useful for application builders.

This memo explores an alternative approach to providing a transaction
transport protocol: extending TCP to implement the transaction
service model, while continuing to support the virtual circuit model.
Each transaction will then be a single instance of a TCP connection.
The proposed transaction extension is effectively implementable
within current TCPs and operating systems, and it should also scale
to the much faster networks, interfaces, and CPUs of the future.

The present memo explains the theory behind the extension, in
somewhat exquisite detail. Despite the length and complexity of this
memo, the TCP extensions required for transactions are in fact quite
limited and simple. Another memo [TTCP-FS] provides a self-contained
functional specification of the extensions.

Section 2 of this memo describes the limitations of standard TCP for
transaction processing, to motivate the extensions. Sections 3, 4,
and 5 explore the fundamental extensions that are required for
transactions. Section 6 discusses the changes required in the TCP
connection state diagram. Finally, Section 7 presents conclusions
and acknowledgments. Familiarity with the standard TCP protocol
[STD-007] is assumed.

2. TRANSACTIONS USING STANDARD TCP

Reliable transfer of data depends upon sequence numbers. Before data
transfer can begin, both parties must "synchronize" the connection,
i.e, agree on common sequence numbers. The synchronization procedure
must preserve at-most-once semantics, i.e., be free from replay
hazards due to duplicate packets. The TCP developers adopted a
synchronization mechanism known as the 3-way handshake.

Consider a simple transaction in which client host A sends a single-
segment request to server host B, and B returns a single-segment
response. Many current TCP implementations use at least ten segments
(i.e., packets) for this sequence: three for the 3-way handshake
opening the connection, four to send and acknowledge the request and
response data, and three for TCP's full-duplex data-conserving close
sequence. These ten segments represent a high relative overhead for
two data-bearing segments. However, a more important consideration
is the transaction latency seen by the client: $2*RTT + SPT$, larger
than the minimum by one RTT. As CPU and network speeds increase, the
relative significance of this extra transaction latency also
increases.

Proposed transaction transport protocols have typically used a
"timer-based" approach to connection synchronization [Birrell84]. In
this approach, once end-to-end connection state is established in the
client and server hosts, a subset of this state is maintained for

some period of time. A new request before the expiration of this
timeout period can then reestablish the full state without an
explicit handshake. Watson pointed out that the timer-based approach
of his Delta-T protocol [Watson81] would encompass both virtual
circuits and transactions. However, the TCP group adopted the 3-way
handshake (because of uncertainty about the robustness of enforcing
the packet lifetime bounds required by Delta-T, within a general
Internet environment). More recently, Liskov, Shrira, and Wroclawski
[Liskov90] have proposed a different timer-based approach to
connection synchronization, requiring loosely-synchronized clocks in
the hosts.

The technique proposed in this memo, suggested by Clark [Clark89],
depends upon cacheing of connection state but not upon clocks or
timers; it is described in Section 3 below. Garlick, Rom, and Postel
also proposed a connection synchronization mechanism using cached
state [Garlick77]. Their scheme required each host to maintain
connection records containing the highest sequence number on each
connection. The technique suggested here retains only per-host
state, not per-connection state.

During TCP development, it was suggested that TCP could support
transactions with data segments containing both SYN and FIN bits.
(These "Kamikaze" segments were not supported as a service; they were
used mainly to crash other experimental TCPs!) To illustrate this
idea, Figure 1 shows a plausible application of the current TCP rules
to create a minimal transaction. (In fact, some minor adjustments in
the standard TCP spec would be required to make Figure 1 fully legal
[STD-007]).

Figure 1, like many of the examples shown in this memo, uses an
abbreviated form to illustrate segment sequences. For clarity and
brevity, it omits explicit sequence and acknowledgment numbers,
assuming that these will follow the well-known TCP rules. The
notation "ACK(x)" implies a cumulative acknowledgment for the control
bit or data "x" and everything preceding "x" in the sequence space.
The referent of "x" should be clear from the context. Also, host A
will always be the client and host B will be the server in these
diagrams.

The first three segments in Figure 1 implement the standard TCP
three-way handshake. If segment #1 had been an old duplicate, the
client side would have sent an RST (Reset) bit in segment #3,
terminating the sequence. The request data included on the initial
SYN segment cannot be delivered to user B until segment #3 completes
the 3-way handshake. Loading control bits onto the segments has
reduced the total number of segments to 5, but the client still
observes a transaction latency of 2*RTT + SPT. The 3-way handshake

thus precludes high-performance transaction processing.

```
     TCP A  (Client)                              TCP B (Server)
     _____                              _____

     CLOSED                                                LISTEN

(Client sends request)
 1. SYN-SENT              --> <SYN,data1,FIN> -->       SYN-RCVD
                                                      (data1 queued)

 2. ESTABLISHED  <-- <SYN,ACK(SYN)> <--                SYN-RCVD


 3. FIN-WAIT-1           --> <ACK(SYN),FIN> -->       CLOSE-WAIT
                                                    (data1 to server)

                                               (Server sends reply)
 4. TIME-WAIT    <-- <ACK(FIN),data2,FIN> <--         LAST-ACK
 (data2 to client)

 5. TIME-WAIT                --> <ACK(FIN)> -->          CLOSED

     (timeout)
     CLOSED
```
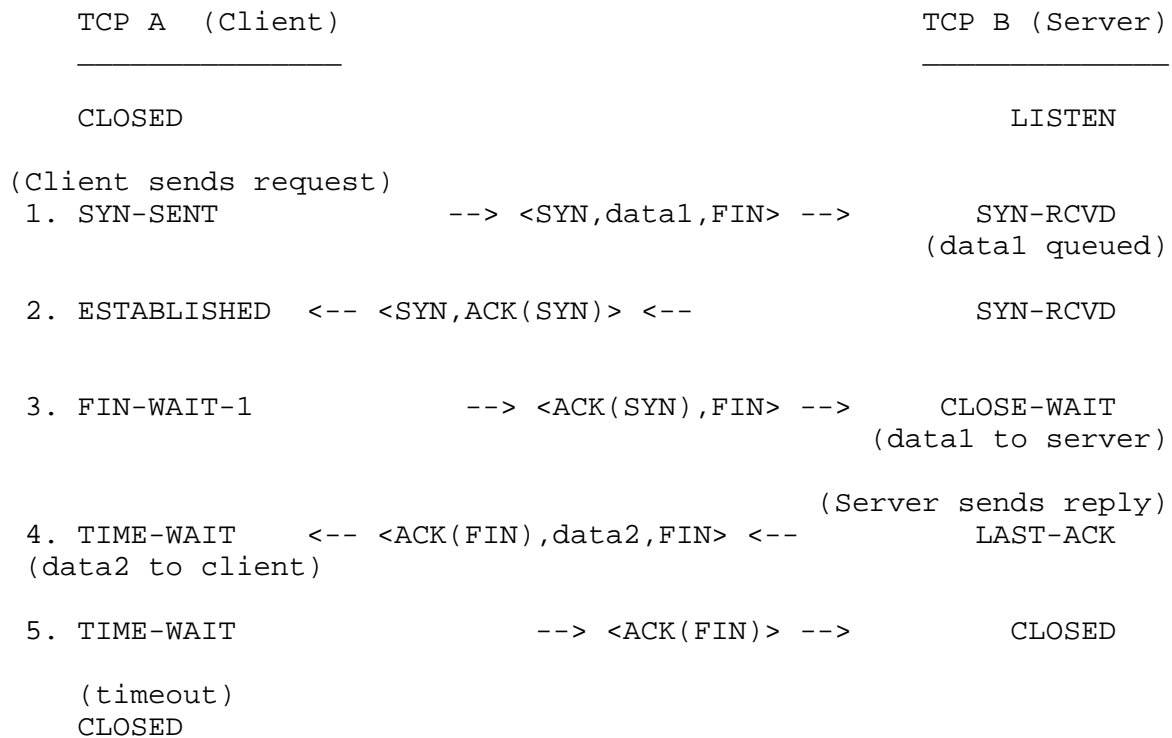
Figure 1: Transaction Sequence: RFC-793 TCP

The TCP close sequence also poses a performance problem for
transactions: one or both end(s) of a closed connection must remain
in "TIME-WAIT" state until a 4 minute timeout has expired [STD-007].
The same connection (defined by the host and port numbers at both
ends) cannot be reopened until this delay has expired. Because of
TIME-WAIT state, a client program should choose a new local port
number (i.e., a different connection) for each successive
transaction. However, the TCP port field of 16 bits (less the
"well-known" port space) provides only 64512 available user ports.
This limits the total rate of transactions between any pair of hosts
to a maximum of 64512/240 = 268 per second. This is much too low a
rate for low-delay paths, e.g., high-speed LANs. A high rate of
short connections (i.e., transactions) could also lead to excessive
consumption of kernel memory by connection control blocks in TIME-
WAIT state.

In summary, to perform efficient transaction processing in TCP, we
need to suppress the 3-way handshake and to shorten TIME-WAIT state.

Protocol mechanisms to accomplish these two goals are discussed in
Sections 3 and 4, respectively. Both require the choice of a
monotonic sequence-like space; Section 5 analyzes the choices and
makes a selection for this space. Finally, the TCP connection state
machine must be extended as described in Section 6.

Transaction processing in TCP raises some other protocol issues,
which are discussed in the functional specification memo [TTCP-FS].
These include:

(1) augmenting the user interface for transactions,

(2) delaying acknowledgment segments to allow maximum piggy-backing
     of control bits with data,

(3) measuring the retransmission timeout time (RTO) on very short
     connections, and

(4) providing an initial server window.

A recently proposed set of enhancements [RFC-1323] defines a TCP
Timestamps option that carries two 32-bit timestamp values. The
Timestamps option is used to accurately measure round-trip time
(RTT). The same option is also used in a procedure known as "PAWS"
(Protect Againsts Wrapped Sequence) to prevent erroneous data
delivery due to a combination of old duplicate segments and sequence
number reuse at very high bandwidths. The particular approach to
transactions chosen in this memo does not require the RFC-1323
enhancements; however, they are important and should be implemented
in every TCP, with or without the transaction extensions described
here.

3. BYPASSING THE 3-WAY HANDSHAKE

To avoid 3-way handshakes for transactions, we introduce a new
mechanism for validating initial SYN segments, i.e., for enforcing
at-most-once semantics without a 3-way handshake. We refer to this
as the TCP Accelerated Open, or TAO, mechanism.

3.1 Concept of TAO

The basis of TAO is this: a TCP uses cached per-host information
to immediately validate new SYNs [Clark89]. If this validation
fails, e.g., because there is no current cached state or the
segment is an old duplicate, the procedure falls back to a normal
3-way handshake to validate the SYN. Thus, bypassing a 3-way
handshake is considered to be an optional optimization.

The proposed TAO mechanism uses a finite sequence-like space of
values that increase monotonically with successive transactions
(connections) between a given (client, server) host pair. Call
this monotonic space M, and let each initial SYN segment carry an
M value SEG.M. If M is not the existing sequence (SEG.SEQ) field,
SEG.M may be carried in a TCP option.

When host B receives from host A an initial SYN segment containing
a new value SEG.M, host B compares this against cache.M[A], the
latest M value that B has cached for host A. This comparison is
the "TAO test". Because the M values are monotonically
increasing, SEG.M > cache.M[A] implies that the SYN must be new
and can be accepted immediately. If not, a normal 3-way handshake
is performed to validate the initial SYN segment. Figure 2
illustrates the TAO mechanism; cached M values are shown enclosed
in square brackets. The M values generated by host A satisfy
x0 < x1, and the M values generated by host B satisfy y0 < y1.

An appropriate choice for the M value space is discussed in
Section 5. M values are drawn from a finite number space, so
inequalities must be defined in the usual way for sequence numbers
[STD-007]. The M space must not wrap so quickly that an old
duplicate SYN will be erroneously accepted. We assume that some
maximum segment lifetime (MSL) is enforced by the IP layer.

```
       TCP  A                                      TCP  B


     cache.M[B]                                  cache.M[A]
        V                                           V

     [ y0 ]                                       [ x0 ]

1.                -->  <SYN,data1,M=x1> -->     ( (x1 > x0) =>
                                                 data1 -> user_B;
                                                 cache.M[A]= x1)

     [ y0 ]                                       [ x1 ]
2.              <-- <SYN,ACK(data1),data2,M=y1> <--

   (data2 -> user_A,
    cache.M[B]= y1)

     [ y1 ]                                       [ x1 ]
                      ... (etc.) ...
```
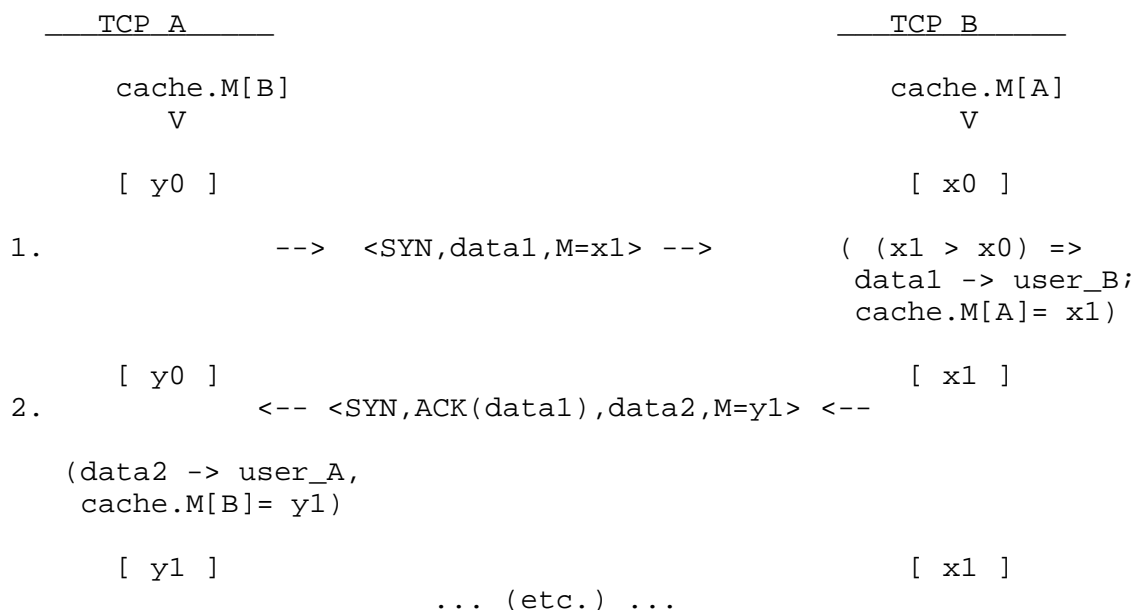
Figure 2. TAO: Three-Way Handshake is Bypassed

Figure 2 shows the simplest case: each side has cached the latest
M value of the other, and the SEG.M value in the client's SYN
segment is greater than the value in the cache at the server host.
As a result, B can accept the client A's request data1 immediately
and pass it to the server application. B's reply data2 is shown
piggybacked on the <SYN,ACK> segment. As a result of this 2-way
exchange, the cached M values are updated at both sites; the
client side becomes relevant only if the client/server roles
reverse. Validation of the <SYN,ACK> segment at host A is
discussed later.

Figure 3 shows the TAO test failing but the consequent 3-way
handshake succeeding. B updates its cache with the value x2 >= x1
when the initial SYN is known to be valid.


```
       TCP A                                        TCP B

        cache.M[B]                                 cache.M[A]
          V                                          V

        [ y0 ]                                      [ x0 ]
1.                  --> <SYN,data1,M=x1> -->   ( (x1 <= x0) =>
                                                 data1 queued;
                                                 3-way handshake)

        [ y0 ]                                      [ x0 ]
2.                  <-- <SYN,ACK(SYN),M=y1> <--
     (cache.M[B]= y1)

        [ y1 ]                                      [ x0 ]
3.                  --> <ACK(SYN),M=x2> -->   (Handshake OK =>
                                                 data1->user_B,
                                                 cache.M[A]= x2)

        [ y1 ]                                      [ x2 ]
                    ...  (etc.)  ...
```

        Figure 3. TAO Test Fails but 3-Way Handshake Succeeds.

There are several possible causes for a TAO test failure on a
legitimate new SYN segment (not an old duplicate).

(1) There may be no cached M value for this particular client
     host.

(2) The SYN may be the one of a set of nearly-simultaneous SYNs
     for different connections but from the same host, which

arrived out of order.

(3) The finite M space may have wrapped around between successive
     transactions from the same client.

(4) The M values may advance too slowly for closely-spaced
     transactions.

None of these TAO failures will cause a lockout, because the
resulting 3-way handshake will succeed. Note that the first
transaction between a given host pair will always require a 3-way
handshake; subsequent transactions can take advantage of TAO.

The per-host cache required by TAO is highly desirable for other
reasons, e.g., to retain the measured round trip time and MTU for
a given remote host. Furthermore, a host should already have a
per-host routing cache [HR-COMM] that should be easily extensible
for this purpose.

Figure 4 illustrates a complete TCP transaction sequence using the
TAO mechanism. Bypassing the 3-way handshake leads to new
connection states; Figure 4 shows three of them, "SYN-SENT*",
"CLOSE-WAIT*", and "LAST-ACK*". Explanation of these states is
deferred to Section 6.


```
      TCP A  (Client)                               TCP B (Server)
      _____                             _____

      CLOSED                                               LISTEN

1.    SYN-SENT*     --> <SYN,data1,FIN,M=x1> -->       CLOSE-WAIT*
                                                       (TAO test OK=>
                                                        data1->user_B)

              <-- <SYN,ACK(FIN),data2,FIN,M=y1> <--      LAST-ACK*
2.    TIME-WAIT
  (data2->user_A)


3.    TIME-WAIT             --> <ACK(FIN),M=x2> -->          CLOSED

      (timeout)
        CLOSED
```
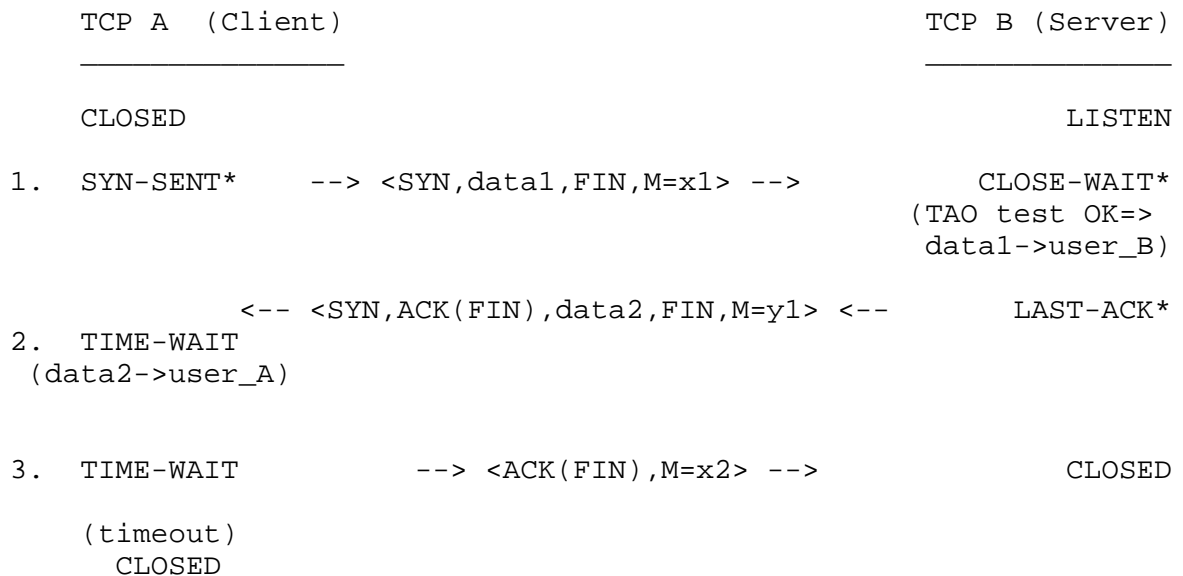

            Figure 4: Minimal Transaction Sequence Using TAO

3.2 Cache Initialization

   The first connection between hosts A and B will find no cached
   state at one or both ends, so both M caches must be initialized.
   This requires that the first transaction carry a specially marked
   SEG.M value, which we call SEG.M.NEW. Receiving a SEG.M.NEW value
   in an initial SYN segment, B will cache this value and send its
   own M back to initialize A's cache. When a host crashes and
   restarts, all its cached M values cache.M[*] must be invalidated
   in order to force a re-synchronization of the caches at both ends.

   This cache synchronization procedure is illustrated in Figure 5,
   where client host A has crashed and restarted with its cache
   entries undefined, as indicated by "??". Since cache.TS[B] is
   undefined, A sends a SEG.M.NEW value instead of SEG.M in the <SYN>
   segment of its first transaction request to B. Receiving this
   SEG.M.NEW, the server host B invalidates cache.TS[A] and performs
   a 3-way handshake. SEG.M in segment #2 updates A's cache, and
   when the handshake completes successfully, B updates its cached M
   value to x2 >= x1.

          TCP A   TCP B

           cache.M[B]   cache.M[A]
              V   V
           [??]   [x0]

   1.   --> <SYN,data1,M.NEW=x1> -->   (invalidate cache;
                                                       queue data1;
                   [??]   3-way handshake)

                                                      [??]
   2.   <-- <SYN,ACK(SYN),M=y1> <--
      (cache.M[B]= y1)

           [y1]   [??]

   3.   --> <ACK(SYN),M=x2> --> data1->user_B,
                                                cache.M[A]= x2)

           [y1]   [x2]
                        ... (etc.) ...
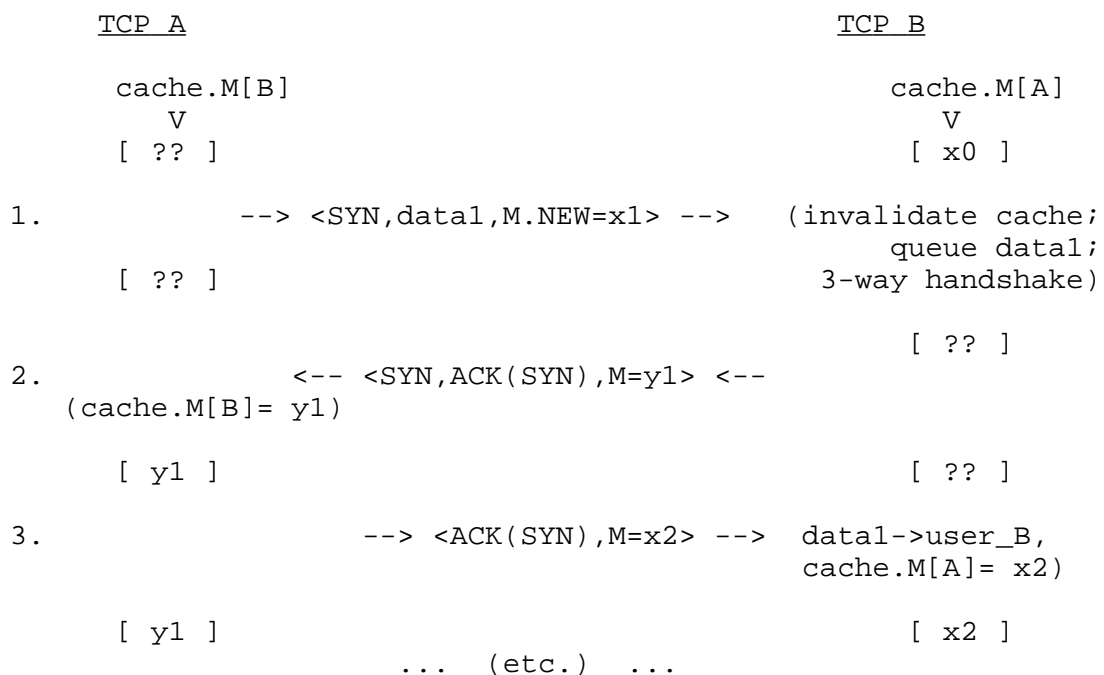
             Figure 5. Client Host Crashed

   Suppose that the 3-way handshake failed, presumably because

segment #1 was an old duplicate. Then segment #3 from host A
would be an RST segment, with the result that both side's caches
would be left undefined.

Figure 6 shows the procedure when the server crashes and restarts.
Upon receiving a <SYN> segment from a host for which it has no
cached M value, B initiates a 3-way handshake to validate the
request and sends its own M value to A. Again the result is to
update cached M values on both sides.

```
          TCP A                                        TCP B

        cache.M[B]                                    cache.M[A]
            V                                             V
         [ y0 ]                                        [ ?? ]

     1.                  --> <SYN,data1,M=x1> -->      (data1 queued;
                                                        3-way handshake)

         [ y0 ]                                        [ ?? ]
     2.                  <-- <SYN,ACK(SYN),M=y1> <--
        (cache.M[B]= y1)

         [ y1 ]                                        [ ?? ]
     3.                  --> <ACK(SYN),M=x2> -->       (data1->user_B,
                                                        cache.M[A]= x2)

         [ y1 ]                                        [ x2 ]
                         ...  (etc.)  ...
```

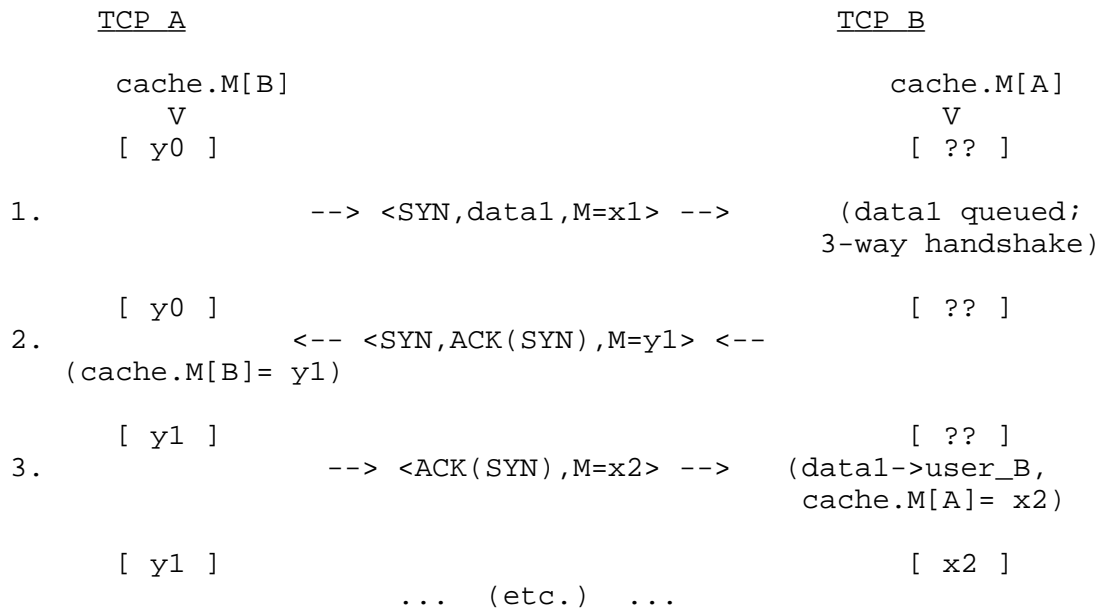   Figure 6. Server Host Crashed

3.3 Accepting <SYN,ACK> Segments

   Transactions introduce a new hazard of erroneously accepting an
   old duplicate <SYN,ACK> segment. To be acceptable, a <SYN,ACK>
   segment must arrive in SYN-SENT state, and its ACK field must
   acknowledge something that was sent. In current TCPs the
   effective send window in SYN-SENT state is exactly one octet, and
   an acceptable <SYN,ACK> must exactly ACK this one octet. The
   clock-driven selection of Initial Sequence Number (ISN) makes an
   erroneous acceptance exceedingly unlikely. An old duplicate SYN
   could be accepted erroneously only if successive connection
   attempts occurred more often than once every 4 microseconds, or if
   the segment lifetime exceeded the 4 hour wraparound time for ISN

selection.

However, when TCP is used for transactions, data sent with the
initial SYN increases the range of sequence numbers that have been
sent. This increases the danger of accepting an old duplicate
<SYN,ACK> segment, and the consequences are more serious. In the
example in Figure 7, segments 1-3 form a normal transaction
sequence, and segment 4 begins a new transaction (incarnation) for
the same connection. Segment #5 is a duplicate of segment #2 from
the preceding transaction. Although the new transaction has a
larger ISN, the previous ACK value 402 falls into the new range
[200,700) of sequence numbers that have been sent, so segment #5
could be erroneously accepted and passed to the client as the
response to the new request.

```
        TCP A                                          TCP B

     CLOSED                                                    LISTEN

1.              --> <seq=100,SYN,data=300,FIN,M=x1> --> (TAO test OK)


2.          <-- <seq=800,ack=402,SYN,data=350,FIN,M=y1> <--


3. TIME-WAIT                     --> <ACK(FIN)> -->        CLOSED
   (short timeout)
   CLOSED

   (New Request)
4.              --> <seq=200,SYN,data=500,FIN,M=x2> --> ...

                                    (Duplicate of segment #2)
5.          <-- <seq=800,ack=402,SYN,data=300,FIN,M=y1> <--...
   (Acceptable!!)
```


            Figure 7: Old Duplicate <SYN,ACK> Causing Error

Unfortunately, we cannot simply use TAO on the client side to
detect and reject old duplicate <SYN,ACK> segments. A TAO test at
the client might fail for a valid <SYN,ACK> segment, due to out-
of-order delivery, and this could result in permanent non-delivery
of a valid transaction reply.

Instead, we include a second M value, an echo of the client's M
value from the initial <SYN> segment, in the <SYN,ACK> segment. A

specially-marked M value, SEG.M.ECHO, is used for this purpose.
The client knows the value it sent in the initial <SYN> and can
therefore positively validate the <SYN,ACK> using the echoed
value. This is illustrated in Figure 12, which is the same as
Figure 4 with the addition of the echoed value on the <SYN,ACK>
segment #2.

It should be noted that TCP allows a simultaneous open sequence in
which both sides send and receive an initial <SYN> (see Figure 8
of [STD-007]. In this case, the TAO test must be performed on
both sides to preserve the symmetry. See [TTCP-FS] for an
example.

4. SHORTENING TIME-WAIT STATE

   Once a transaction has been initiated for a particular connection
   (pair of ports) between a given host pair, a new transaction for the
   same connection cannot take place for a time that is at least:

       RTT + SPT + TIME-WAIT_delay

   Since the client host can cycle among the 64512 available port
   numbers, an upper bound on the transaction rate between a particular
   host pair is:

   [1]   TRmax = 64512 /(RTT + TIME-WAIT_Delay)

   in transactions per second (Tps), where we assumed SPT is negligible.
   We must reduce TIME-WAIT_Delay to support high-rate TCP transaction
   processing.

   TIME-WAIT state performs two functions: (1) supporting the full-
   duplex reliable close of TCP, and (2) allowing old duplicate segments
   from an earlier connection incarnation to expire before they can
   cause an error (see Appendix to [RFC-1185]). The first function
   impacts the application model of a TCP connection, which we would not
   want to change. The second is part of the fundamental machinery of
   TCP reliable delivery; to safely truncate TIME-WAIT state, we must
   provide another means to exclude duplicate packets from earlier
   incarnations of the connection.

   To minimize the delay in TIME-WAIT state while performing both
   functions, we propose to set the TIME-WAIT delay to:

   [2]   TIME-WAIT_Delay = max(K*RTO, U)

   where U and K are constants and RTO is the dynamically-determined
   retransmission timeout, the measured RTT plus an allowance for the

RTT variance [Jacobson88]. We choose K large enough so that there is
high probability of the close completing successfully if at all
possible; K = 8 seems reasonable. This takes care of the first
function of TIME-WAIT state.

In a real implementation, there may be a minimum RTO value Tr,
corresponding to the precision of RTO calculation. For example, in
the popular BSD implementation of TCP, the minimum RTO is Tr = 0.5
second. Assuming K = 8 and U = 0, Eqns [1] and [2] impose an upper
limit of TRmax = 16K Tps on the transaction rate of these
implementations.

It is possible to have many short connections only if RTO is very
small, in which case the TIME-WAIT delay [2] reduces to U. To
accelerate the close sequence, we need to reduce U below the MSL
enforced by the IP layer, without introducing a hazard from old
duplicate segments. For this purpose, we introduce another monotonic
number sequence; call it X. X values are required to be monotonic
between successive connection incarnations; depending upon the choice
of the X space (see Section 5), X values may also increase during a
connection. A value from the X space is to be carried in every
segment, and a segment is rejected if it is received with an X value
smaller than the largest X value received. This mechanism does not
use a cache; the largest X value is maintained in the TCP connection
control block (TCB) for each connection.

The value of U depends upon the choice for the X space, discussed in
the next section. If X is time-like, U can be set to twice the time
granularity (i.e, twice the minimum "tick" time) of X. The TIME-WAIT
delay will then ensure that current X values do not overlap the X
values of earlier incarnations of the same connection. Another
consequence of time-like X values is the possibility that an open but
idle connection might allow the X value to wrap its sign bit,
resulting in a lockup of the connection. To prevent this, a 24-day
idle timer on each open connection could bypass the X check on the
first segment following the idle period, for example. In practice,
many implementations have keep-alive mechanisms that prevent such
long idle periods [RFC-1323].

Referring back to Figure 4, our proposed transaction extension
results in a minimum exchange of 3 packets. Segment #3, the final
ACK segment, does not increase transaction latency, but in
combination with the TIME-WAIT delay of K*RTO it ensures that the
server side of the connection will be closed before a new transaction
is issued for this same pair of ports. It also provides an RTT
measurement for the server.

We may ask whether it would be possible to further reduce the TIME-

WAIT delay. We might set K to zero; alternatively, we might allow
the client TCP to start a new transaction request while the
connection was still in TIME-WAIT state, with the new initial SYN
acting as an implied acknowledgment of the previous FIN. Appendix A
summarizes the issues raised by these alternatives, which we call
"truncating" TIME-WAIT state, and suggests some possible solutions.
Further study would be required, but these solutions appear to bend
the theory and/or implementations of the TCP protocol farther than we
wish to bend them.

We therefore propose using formula [2] with K=8 and retaining the
final ACK(FIN) transmission. To raise the transaction rate,
therefore, we require small values of RTO and U.

5. CHOOSING A MONOTONIC SEQUENCE

   For simplicity, we want the monotonic sequence X used for shortening
   TIME-WAIT state to be identical to the monotonic sequence M for
   bypassing the 3-way handshake. Calling the common space M, we will
   send an M value SEG.M in each TCP segment. Upon receipt of an
   initial SYN segment, SEG.M will be compared with a per-host cached
   value to authenticate the SYN without a 3-way handshake; this is the
   TAO mechanism. Upon receipt of a non-SYN segment, SEG.M will be
   compared with the current value in the connection control block and
   used to discard old duplicates.

   Note that the situation with TIME-WAIT state differs from that of
   bypassing 3-way handshakes in two ways: (a) TIME-WAIT requires
   duplicate detection on every segment vs. only on SYN segments, and
   (b) TIME-WAIT applies to a single connection vs. being global across
   all connections. This section discusses possible choices for the
   common monotonic sequence.

   The SEG.M values must satisfy the following requirements.

   *   The values must be monotonic; this requirement is defined more
        precisely below.

   *   Their granularity must be fine-grained enough to support a high
        rate of transaction processing; the M clock must "tick" at least
        once between successive transactions.

   *   Their range (wrap-around time) must be great enough to allow a
        realistic MSL to be enforced by the network.

   The TCP spec calls for an MSL of 120 secs. Since much of the
   Internet does not carefully enforce this limit, it would be safer to
   have an MSL at least an order of magnitude larger. We set as an

objective an MSL of at least 2000 seconds. If there were no TIME-
WAIT delay, the ultimate limit on transaction rate would be set by
speed-of-light delays in the network and by the latency of host
operating systems. As the bottleneck problems with interfacing CPUs
to gigabit LANs are solved, we can imagine transaction durations as
short as 1 microsecond. Therefore, we set an ultimate performance
goal of TRmax at least $10^{**}6$ Tps.

A particular connection between hosts A and B is identified by the
local and remote TCP "sockets", i.e., by the quadruplet: {A, B,
Port.A, Port.B}. Imagine that each host keeps a count CC of the
number of TCP connections it has initiated. We can use this CC
number to distinguish different incarnations of the same connection.
Then a particular SEG.M value may be labeled implicitly by 6
quantities: {A, B, Port.A, Port.B, CC, n}, where n is the byte offset
of that segment within the connection incarnation.

To bypass the 3-way handshake, we require thgt SEG.M values on
successive SYN segments from a host A to a host B be monotone
increasing. If CC' > CC, then we require that:

    SEG.M(A,B,Port.A,Port.B,CC',0) > SEG.M(A,B,Port.A,Port.B,CC,0)

for any legal values of Port.A and Port.B.

To delete old duplicates (allowing TIME-WAIT state to be shortened),
we require that SEG.M values be disjoint across different
incarnations of the same connection.   If CC' > CC then

    SEG.M(A,B,Port.A,Port.B,CC',n') > SEG.M(A,B,Port.A,Port.B,CC,n),

for any non-negative integers n and n'.

We now consider four different choices for the common monotonic
space: RFC-1323 timestamps, TCP sequence numbers, the connection
count, and 64-bit TCP sequence numbers. The results are summarized
in Table I.

5.1 Cached Timestamps

    The PAWS mechanism [RFC-1323] uses TCP "timestamps" as
    monotonically increasing integers in order to throw out old
    duplicate segments within the same incarnation. Jacobson
    suggested the cacheing of these timestamps for bypassing 3-way
    handshakes [Jacobson90], i.e., that TCP timestamps be used for our
    common monotonic space M. This idea is attractive since it would
    allow the same timestamp options to be used for RTTM, PAWS, and
    transactions.

To obtain at-most-once service, the criterion for immediate
acceptance of a SYN must be that SEG.M is strictly greater than
the cached M value. That is, to be useful for bypassing 3-way
handshakes, the timestamp clock must tick at least once between
any two successive transactions between the same pair of hosts
(even if different ports are used). Hence, the timestamp clock
rate would determine TRmax, the maximum possible transaction rate.

Unfortunately, the timestamp clock frequency called for by RFC-
1323, in the range 1 sec to 1 ms, is much too slow for
transactions. The TCP timestamp period was chosen to be
comparable to the fundamental interval for computing and
scheduling retransmission timeouts; this is generally in the range
of 1 sec. to 1 ms., and in many operating systems, much closer to
1 second. Although it would be possible to increase the timestamp
clock frequency by several orders of magnitude, to do so would
make implementation more difficult, and on some systems
excessively expensive.

The wraparound time for TCP timestamps, at least 24 days, causes
no problem for transactions.

The PAWS mechanism uses TCP timestamps to protect against old
duplicate non-SYN segments from the same incarnation [RFC-1323].
It can also be used to protect against old duplicate data segments
from earlier incarnations (and therefore allow shortening of
TIME-WAIT state) if we can ensure that the timestamp clock ticks
at least once between the end of one incarnation and the beginning
of the next. This can be achieved by setting U = 2 seconds, i.e.,
to twice the maximum timestamp clock period. This value in
formula [2] leads to an upper bound TRmax = 32K Tps between a host
pair. However, as pointed out above, old duplicate SYN detection
using timestamps leads to a smaller transaction rate bound, 1 Tps,
which is unacceptable. In addition, the timestamp approach is
imperfect; it allows old ACK segments to enter the new connection
where they can cause a disconnect. This happens because old
duplicate ACKs that arrive during TIME-WAIT state generate new
ACKs with the current timestamp [RFC-1337].

We therefore conclude that timestamps are not adequate as the
monotonic space M; see Table I. However, they may still be useful
to effectively extend some other monotonic number space, just as
they are used in PAWS to extend the TCP sequence number space.
This is discussed below.

   5.2 Current TCP Sequence Numbers

      It is useful to understand why the existing 32-bit TCP sequence
      numbers do not form an appropriate monotonic space for
      transactions.

      The sequence number sent in an initial SYN is called the Initial
      Sequence Number or ISN. According to the TCP specification, an
      ISN is to be selected using:

      [3]   ISN = (R*T) mod 2**32

      where T is the real time in seconds (from an arbitrary origin,
      fixed when the system is started) and R is a constant, currently
      250 KBps. These ISN values form a monotonic time sequence that
      wraps in 4.55 hours = 16380 seconds and has a granularity of 4
      usecs. For transaction rates up to roughly 250K Tps, the ISN
      value calculated by formula [3] will be monotonic and could be
      used for bypassing the 3-way handshake.

      However, TCP sequence numbers (alone) could not be used to shorten
      TIME-WAIT state, because there are several ways that overlap of
      the sequence space of successive incarnations can occur (as
      described in Appendix to [RFC-1185]). One way is a "fast
      connection", with a transfer rate greater than R; another is a
      "long" connection, with a duration of approximately 4.55 hours.
      TIME-WAIT delay is necessary to protect against these cases. With
      the official delay of 240 seconds, formula [1] implies a upper
      bound (as RTT -> 0) of TRmax = 268 Tps; with our target MSL of
      2000 sec, TRmax = 32 Tps. These values are unacceptably low.

      To improve this transaction rate, we could use TCP timestamps to
      effectively extend the range of the TCP sequence numbers.
      Timestamps would guard against sequence number wrap-around and
      thereby allow us to increase R in [3] to exceed the maximum
      possible transfer rate. Then sequence numbers for successive
      incarnations could not overlap. Timestamps would also provide
      safety with an MSL as large as 24 days. We could then set U = 0
      in the TIME-WAIT delay calculation [2]. For example, R = 10**9
      Bps leads to TRmax <= 10**9 Tps. See 2(b) in Table I. These
      values would more than satisfy our objectives.

      We should make clear how this proposal, sequence numbers plus
      timestamps, differs from the timestamps alone discussed (and
      rejected) in the previous section. The difference lies in what is
      cached and tested for TAO; the proposal here is to cache and test
      BOTH the latest TCP sequence number and the latest TCP timestamp.
      In effect, we are proposing to use timestamps to logically extend

the sequence space to 64 bits. Another alternative, presented in
the next section, is to directly expand the TCP sequence space to
64 bits.

Unfortunately, the proposed solution (TCP sequence numbers plus
timestamps) based on equation [3] would be difficult or impossible
to implement on many systems, which base their TCP implementation
upon a very low granularity software clock, typically O(1 sec).
To adapt the procedure to a system with a low granularity software
clock, suppose that we calculate the ISN as:

[4]   ISN = (R*Ts*floor(T/Ts) + q*CC) mod 2**32

where Ts is the time per tick of the software clock, CC is the
connection count, and q is a constant. That is, the ISN is
incremented by the constant R*Ts once every clock tick and by the
constant q for every new connection. We need to choose q to
obtain the required monotonicity.

For monotonicity of the ISN's themselves, q=1 suffices. However,
monotonicity during the entire connection requires q = R*Ts. This
value of q can be deduced as follows. Let S(T, CC, n) be the
sequence number for byte offset n in a connection with number CC
at time T:

    S(T, CC, n) = (R*Ts*floor(T/Ts) + q*CC + n) mod 2**32.

For any T1 > T2, we require that: S(T2, CC+1, 0) - S(T1, CC, n) >
0 for all n. Since R is assumed to be an upper bound on the
transfer rate, we can write down:

    R > n/(T2 - T1),   or   T2/Ts - T1/Ts > n/(R*Ts)

Using the relationship: floor(x)-floor(y) > x-y-1 and a little
algebra leads to the conclusion that using q = R*Ts creates the
required monotonic number sequence. Therefore, we consider:

[5]   ISN = R*Ts*(floor(T/Ts) + CC) mod 2**32

(which is the algorithm used for ISN selection by BSD TCP).

For error-free operation, the sequence numbers generated by [5]
must not wrap the sign bit in less than MSL seconds. Since CC
cannot increase faster than TRmax, the safe condition is:

    R* (1 + Ts*TRmax) * MSL < 2**31.

We are interested in the case: Ts*TRmax >> 1, so this relationship

reduces to:

[6]   R * Ts * TRmax * MSL < 2**31.

This shows a direct trade-off among the maximum effective
bandwidth R, the maximum transaction rate TRmax, and the maximum
segment lifetime MSL. For reasonable limiting values of R, Ts,
and MSL, formula [6] leads to a very low value of TRmax. For
example, with MSL= 2000 secs, R=10**9 Bps, and Ts = 0.5 sec, TRmax
< 2*10**-3 Tps.

To ease the situation, we could supplement sequence numbers with
timestamps. This would allow an effective MSL of 2 seconds in
[6], since longer times would be protected by differing
timestamps. Then TRmax < 2**30/(R*Ts). The actual enforced MSL
would be increased to 24 days. Unfortunately, TRmax would still
be too small, since we want to support transfer rates up to R ~
10**9 Bps. Ts = 0.5 sec would imply TRmax ~ 2 Tps. On many
systems, it appears infeasible to decrease Ts enough to obtain an
acceptable TRmax using this approach.

5.3 64-bit TCP Sequence Numbers

Another possibility would be to simply increase the TCP sequence
space to 64 bits as suggested in [RFC-1263]. We would also
increase the R value for clock-driven ISN selection, beyond the
fastest transfer rate of which the host is capable. A reasonable
upper limit might be R = 10**9 Bps. As noted above, in a
practical implementation we would use:

        ISN = R*Ts*(floor(T/Ts) + CC) mod 2**64

leading to:

        R*(1 + Ts * TRmax) * MSL < 2**63

For example, suppose that R = 10**9 Bps, Ts = 0.5, and MSL = 16K
secs (4.4 hrs); then this result implies that TRmax < 10**6 Tps.
We see that adding 32 bits to the sequence space has provided
feasible values for transaction processing.

5.4 Connection Counts

The Connection Count CC is well suited to be the monotonic
sequence M, since it "ticks" exactly once for each new connection
incarnation and is constant within a single incarnation. Thus, it
perfectly separates segments from different incarnations of the
same connection and would allow U = 0 in the TIME-WAIT state delay

formula [2]. (Strictly, U cannot be reduced below 1/R = 4 usec,
as noted in Section 4. However, this is of little practical
consequence until the ultimate limits on TRmax are approached).

Assume that CC is a 32-bit number. To prevent wrap-around in the
sign bit of CC in less than MSL seconds requires that:

        TRmax * MSL < 2**31

For example, if MSL = 2000 seconds then TRmax < 10**6 Tp. These
are acceptable limits for transaction processing. However, if
they are not, we could augment CC with TCP timestamps to obtain
very far-out limits, as discussed below.

It would be an implementation choice at the client whether CC is
global for all destinations or private to each destination host
(and maintained in the per-host cache). In the latter case, the
last CC value assigned for each remote host could also be
maintained in the per-host cache. Since there is not typically a
large amount of parallelism in the network connection of a host,
there should be little difference in the performance of these two
different approaches, and the single global CC value is certainly
simpler.

To augment CC with TCP timestamps, we would bypass a 3-way
handshake if both SEG.CC > cache.CC[A] and SEG.TSval >=
cache.TS[A]. The timestamp check would detect a SYN older than 2
seconds, so that the effective wrap-around requirement would be:

        TRmax * 2 < 2**31

i.e., TRmax < 10**9 Tps. The required MSL would be raised to 24
days. Using timestamps in this way, we could reduce the size of
CC. For example, suppose CC were 16 bits. Then the wrap-around
condition TRmax * 2 < 2**15 implies that TRmax is 16K.

Finally, note that using CC to delete old duplicates from earlier
incarnations would not obviate the need for the time-stamp-based
PAWS mechanism to prevent errors within a single incarnation due
to wrapping the 32-bit TCP sequence space at very high transfer
rates.

5.5 Conclusions

The alternatives for monotonic sequence are summarized in Table I.
We see that there are two feasible choices for the monotonic
space: the connection count and 64-bit sequence numbers. Of these
two, we believe that the simpler is the connection count.

Implementation of 64-bit sequence numbers would require
negotiation of a new header format and expansion of all variables
and calculations on the sequence space. CC can be carried in an
option and need be examined only once per packet.

We propose to use a simple 32-bit connection count CC, without
augmentation with timestamps, for the transaction extension. This
choice has the advantages of simplicity and directness. Its
drawback is that it adds a third sequence-like space (in addition
to the TCP sequence number and the TCP timestamp) to each TCP
header and to the main line of packet processing. However, the
additional code is in fact very modest.

We now have a general outline of the proposed TCP extensions for
transactions.

o   A host maintains a 32-bit global connection counter variable CC.

o   The sender's current CC value is carried in an option in every
     TCP segment.

o   CC values are cached per host, and the TAO mechanism is used to
     bypass the 3-way handshake when possible.

o   In non-SYN segments, the CC value is used to reject duplicates
     from earlier incarnations. This allows TIME-WAIT state delay to
     be reduced to K*RTO (i.e., U=0 in Eq. [2]).

TABLE I: Summary of Monotonic Sequences

APPROACH	TRmax (Tps)	Required MSL	COMMENTS
1. Timestamp & PAWS	1	24 days	TRmax is too small
2. Current TCP Sequence Numbers			
(a) clock-driven ISN: eq. [3]	268	240 secs	TRmax & MSL too small
(b) Timestamps& clock-driven ISN [3] & R=10**9	10**9	24 days	Hard to implement
(c) Timestamps & c-dr ISN: eq. [4]	2**30/(R*Ts)	24 days	TRmax too small.
3. 64-bit TCP Sequence Numbers	2**63/(MSL*R*Ts) e.g., R=10**9 Bps, MSL = 4.4 hrs, Ts = 0.5 sec=> TRmax = 10**6	MSL	Significant TCP change
4. Connection Counts			
(a) no timestamps	2**31/MSL e.g., MSL=2000 sec TRmax = 10**6	MSL	3rd sequence space
(b) with timestamps and PAWS	2**30	24 days	(ditto)

6. CONNECTION STATES

   TCP has always allowed a connection to be half-closed. TAO makes a
   significant addition to TCP semantics by allowing a connection to be
   half-synchronized, i.e., to be open for data transfer in one
   direction before the other direction has been opened. Thus, the
   passive end of a connection (which receives an initial SYN) can
   accept data and even a FIN bit before its own SYN has been
   acknowledged. This SYN, data, and FIN may arrive on a single segment
   (as in Figure 4), or on multiple segments; packetization makes no
   difference to the logic of the finite-state machine (FSM) defining
   transitions among connection states.

   Half-synchronized connections have several consequences.

   (a)   The passive end must provide an implied initial data window in
         order to accept data. The minimum size of this implied window
         is a parameter in the specification; we suggest 4K bytes.

   (b)   New connection states and transitions are introduced into the
         TCP FSM at both ends of the connection. At the active end, new
         states are required to piggy-back the FIN on the initial SYN
         segment. At the passive end, new states are required for a
         half-synchronized connection.

   This section develops the resulting FSM description of a TCP
   connection as a conventional state/transition diagram. To develop a
   complete FSM, we take a constructive approach, as follows: (1) write
   down all possible events; (2) write down the precedence rules that
   govern the order in which events may occur; (3) construct the
   resulting FSM; and (4) augment it to support TAO. In principle, we
   do this separately for the active and passive ends; however, the
   symmetry of TCP results in the two FSMs being almost entirely
   coincident.

   Figure 8 lists all possible state transitions for a TCP connection in
   the absence of TAO, as elementary events and corresponding actions.
   Each transition is labeled with a letter. Transitions a-g are used
   by the active side, and c-i are used by the passive side. Without
   TAO, transition "c" (event "rcv ACK(SYN)") synchronizes the
   connection, allowing data to be accepted for the user.

   By definition, the first transition for an active (or passive) side
   must be "a" (or "i", respectively). During a single instance of a
   connection, the active side will progress through some permutation of
   the complete sequence of transitions {a b c d e f } or the sequence
   {a b c d e f g}. The set of possible permutations is determined by
   precedence rules governing the order in which transitions can occur.

```
         Label              Event / Action
         _____              _____
          a                 OPEN / snd SYN

          b                 rcv SYN [No TAO]/ snd ACK(SYN)

          c                 rcv ACK(SYN) /

          d                 CLOSE / snd FIN

          e                 rcv FIN / snd ACK(FIN)

          f                 rcv ACK(FIN) /

          g                 timeout=2MSL / delete TCB
    _____
          h                 passive OPEN / create TCB

          i                 rcv SYN [No TAO]/ snd SYN, ACK(SYN)
    _____
```

        Figure 8. Basic TCP Connection Transitions

   Using the notation "<." to mean "must precede", the precedence rules
   are:

   (1) Logical ordering: must open connection before closing it:

        b <. e

   (2) Causality -- cannot receive ACK(x) before x has been sent:

        a <. c and i <. c and d <. f

   (3) Acknowledgments are cumulative

        c <. f

   (4) First packet in each direction must contain a SYN.

        b <. c and b <. f

   (5) TIME-WAIT state

        Whenever d precedes e in the sequence, g must be the last
        transition.

Applying these rules, we can enumerate all possible permutations of
the events and summarize them in a state transition diagram. Figure
9 shows the result, with boxes representing the states and directed
arcs representing the transitions.

```
         _____           h          _____
        |        |--------->|        |
        | CLOSED |          | LISTEN |
        |_____|          |_____|
            |                   |
            | a                 | i
          __V____            ___V___                   _____
        |        |    b     |        |     e          |        |
        |        |--------->|        |-------------->|        |
        |_____|          |_____|                |_____|
            |  /               |  /  |                   |  /  |
            | /               / |   | c               d / |   | c
            |/               /  __V____                /  ___V___
         d  |             d /  |        |    e        |  |        |
            |             /    |        |----------->|  |        |
            |             |    |_____|             |  |_____|
            |             |       |                   |     |
            |             |       |                  __V____ |
            |             |       |                 |        ||
            |             |       |                 |        ||
            |             |       |                 |_____||
          __V___        __V_      |                   |      |
        |        |  b  |        | e |                 |      |
        |        |---->|        |------->|        |    |      |
        |_____|     |_____|        |        |    |      |
                          |   |          |_____|    |      |
                        c |   | / d      c |         c |    d |
                          |   |/           |           |      |
                        _V___V__          _V_          V_____V_
                       |        | e     |        |   |        |
                       |        |---->  |        |   |        |
                       |_____|       |_____|   |_____|
                          |                |            |
                          | f              | f          | f
                        __V____          __V___       __V____
                       |        | e    | TIME- | g  |        |
                       |        |---->|  WAIT |-->| CLOSED |
                       |_____|     |_____|   |_____|
```
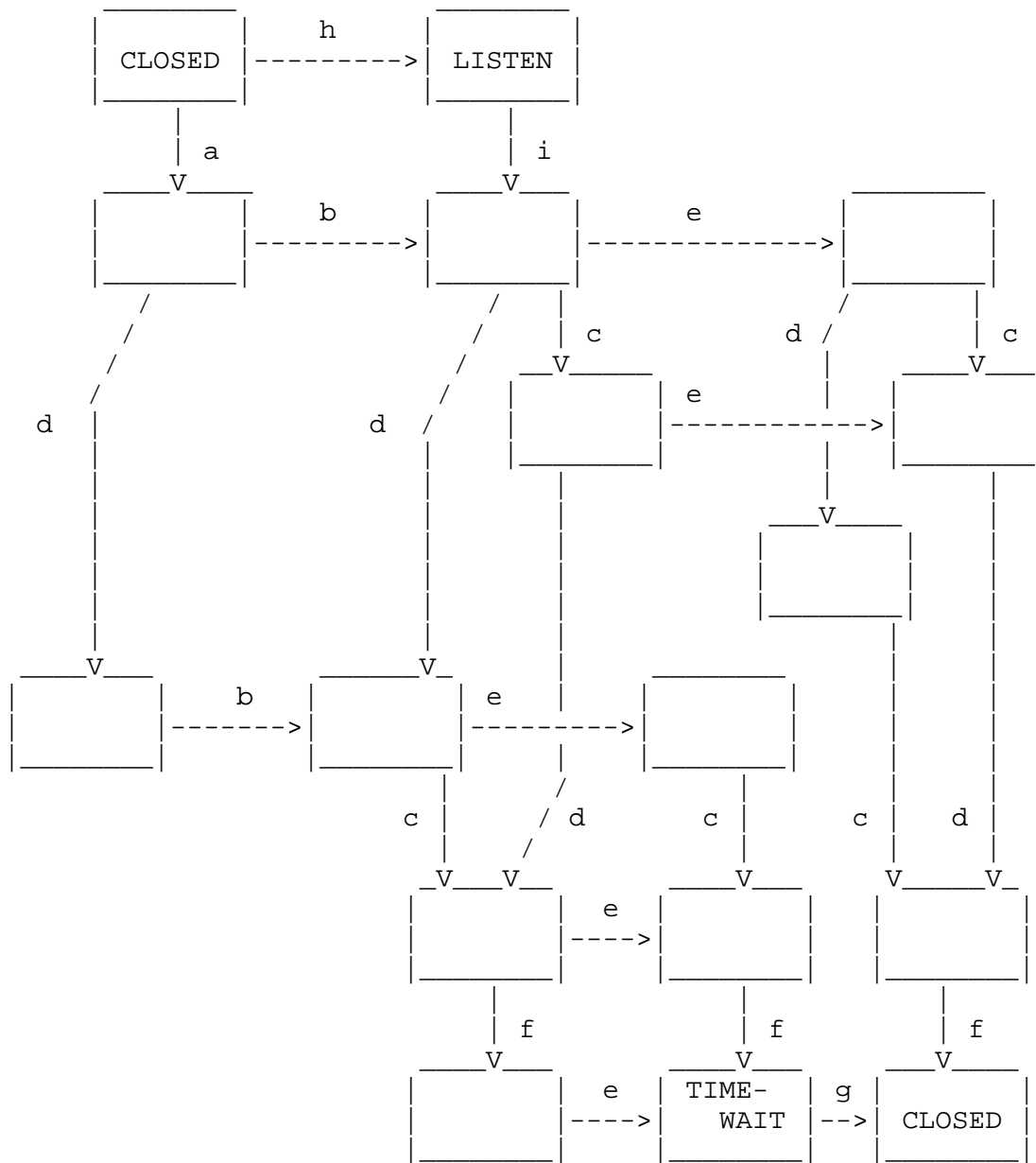
Figure 9: Basic State Diagram

Although Figure 9 gives a correct representation of the possible
event sequences, it is not quite correct for the actions, which do
not compose as shown.   In particular, once a control bit X has been
sent, it must continue to be sent until ACK(X) is received.   This
requires new transitions with modified actions, shown in the
following list. We use the labeling convention that transitions with
the same event part all have the same letter, with different numbers
of primes to indicate different actions.

```
        Label                 Event / Action
        _____                 _____

         b' (=i)              rcv SYN [No TAO] / snd SYN,ACK(SYN)
         b''                  rcv SYN [No TAO] / snd SYN,FIN,ACK(SYN)
         d'                   CLOSE / snd SYN,FIN
         e'                   rcv FIN / snd FIN,ACK(FIN)
         e''                  rcv FIN / snd SYN,FIN,ACK(FIN)
```

Figure 10 shows the state diagram of Figure 9, with the modified
transitions and with the states used by standard TCP [STD-007]
identified. Those states that do not occur in standard TCP are
numbered 1-5.

Standard TCP has another implied restriction: a FIN bit cannot be
recognized before the connection has been synchronized, i.e., c <. e.
This eliminates from standard TCP the states 1, 2, and 5 shown in
Figure 10. States 3 and 4 are needed if a FIN is to be piggy-backed
on a SYN segment (note that the states shown in Figure 1 are actually
wrong; the states shown as SYN-SENT and ESTABLISHED are really states
3 and 4). In the absence of piggybacking the FIN bit, Figure 10
reduces to the standard TCP state diagram [STD-007].

The FSM described in Figure 10 is intended to be applied
cumulatively; that is, parsing a single packet header may lead to
more than one transition. For example, the standard TCP state
diagram includes a direct transition from SYN-SENT to ESTABLISHED:

    rcv SYN,ACK(SYN) / snd ACK(SYN).

This is transition b followed immediately by c.

```
 _____                      _____
|        |          h         |        |
| CLOSED |--------->          | LISTEN |
|_____|                    |_____|
    |                             |
    | a                           | i
    V                             V
 _____         b'           _____         e'          _____
|  SYN-  |-------->           |  SYN-  |--------------->    |        |
|  SENT  |                    |RECEIVED|                    |   1    |
|_____|                    |_____|                    |_____|
    /                            /     |                        |    |
 d'/                          d'/      | c                  d' |  c |
  /                            /        V                      |    V
 /                            /      _____      e           |  _____
 |                            /     | ESTAB- |------------ |--> | CLOSE- |
 |                           /      | LISHED |             |    |  WAIT  |
 |                           |      |_____|                  |_____|
 |                           |         |                     _____V__
 |                           |         |                    |        |
 |                           |         |                    |   2    |
 |                           |         |                    |_____|
 |                           |         |                        |     |
 V___                      __V_      __|___                     |     |
|      |    b''          |      | e'''   |          _____   |     |
|  3   |-------->        |  4   |------------>      |   5   |   |     |
|_____|                 |_____|                   |_____|  |     |
                            |        /                   |      |     |
                         c  |       / d              c   |   c  |   d |
                            |      /                     |      |     |
                         _V___V__                     ___V___   V_____V_
                        |  FIN-  | e''   _____     |        |        |
                        | WAIT-1 |---->  |CLOSING|    | LAST-  |
                        |_____|       |_____|    |  ACK   |
                            |               |            |
                            | f             | f          | f
                         ___V___         ___V___       ___V____
                        |  FIN-  |   e   | TIME- |  g  |        |
                        | WAIT-2 |---->  | WAIT  |-->  | CLOSED |
                        |_____|       |_____|     |_____|
```
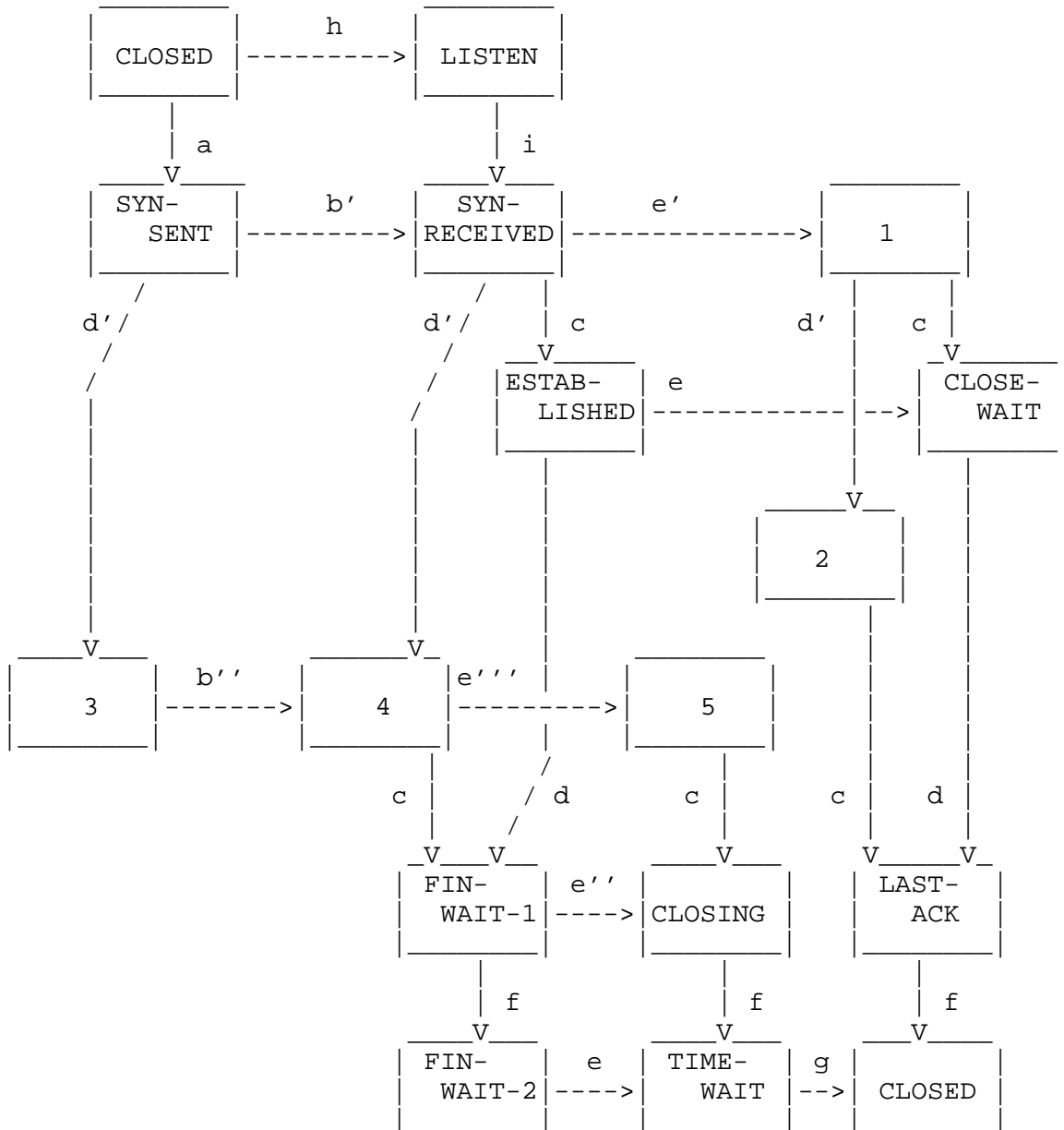
                Figure 10: Basic State Diagram -- Correct Actions

      Next we introduce TAO. If the TAO test succeeds, the connection
      becomes half-synchronized. This requires a new set of states,
      mirroring the states of Figure 10, beginning with acceptance of a SYN
      (transition "b" or "i"), and ending when ACK(SYN) arrives (transition

"c"). Figure 11 shows the result of augmenting Figure 10 with the
additional states for TAO. The transitions are defined in the
following table:

         Key for Figure 11: Complete State Diagram with TAO

            Label   Event / Action
            _____   _____

             a   OPEN / create TCB, snd SYN
             b'   rcv SYN [no TAO]/ snd SYN,ACK(SYN)
             b''   rcv SYN [no TAO]/ snd SYN,FIN,ACK(SYN)
             c   rcv ACK(SYN) /
             d   CLOSE / snd FIN
             d'   CLOSE / snd SYN,FIN
             e   rcv FIN / snd ACK(FIN)
             e'   rcv FIN / snd SYN,ACK(FIN)
             e''   rcv FIN / snd FIN,ACK(FIN)
             e'''   rcv FIN / snd SYN,FIN,ACK(FIN)
             f   rcv ACK(FIN) /
             g   timeout=2MSL / delete TCB
             h   passive OPEN / create TCB
             i (= b')   rcv SYN [no TAO]/ snd SYN,ACK(SYN)
             j   rcv SYN [TAO OK] / snd SYN,ACK(SYN)
             k   rcv SYN [TAO OK] / snd SYN,FIN,ACK(SYN)

Each new state in Figure 11 bears a very simple relationship to a
standard TCP state. We indicate this by naming the new state with
the standard state name followed by a star. States SYN-SENT* and
SYN-RECEIVED* differ from the corresponding unstarred states in
recording the fact that a FIN has been sent. The other new states
with starred names differ from the corresponding unstarred states in
being half-synchronized (hence, a SYN bit needs to be transmitted).

The state diagram of Figure 11 is more general than required for
transaction processing. In particular, it handles simultaneous
connection synchronization from both sides, allowing one or both
sides to bypass the 3-way handshake. It includes other transitions
that are unlikely in normal transaction processing, for example, the
server sending a FIN before it receives a FIN from the client
(ESTABLISHED* -> FIN-WAIT-1* in Figure 11).

```
 _____                             _____
|        |            h              |        |
| CLOSED |-------------------------->| LISTEN |
|_____|                           |_____|
     |                                /      |
    a|                              / i    | j
     |                            /        |
     |                          /          |
     |                 j      _V_____    |
     |               /------|ESTAB-    |    |          e'      _____
     |             /--------|----->| LISHED*  |------------->|CLOSE-  |
     |           /          |      |_____|              |WAIT*   |
     |         /            |          |   |               |_____|
     |       /              |        d'|   |c           d' |     |c
  ___V___  /          ___V__ |      _V_____  e         _V_____
 | SYN-  |/    b'     | SYN- |    |ESTAB-    |----------|->|CLOSE-  |
 | SENT  |------->|RECEIVED|-----|-->| LISHED |              | WAIT  |
 |_____|       |_____|      |  |_____|              |_____|
     |               |           |      |                 ___V_____  |
     |               |           |      |                | LAST-   | |
    d'|             d'|        d'|    d|               | ACK*    | |
     |               |           |      |                |_____| |
     |               |           |      |                    |       |
     |               |         ___V_    |    e'''           |c    |d
     |        k      |       |FIN-  |   |          _____  |     |
     |      /------|-->| WAIT-1*|---|------>|CLOSING*|  |     |
     |    /           |       |_____|   |          |_____|  |     |
     |  /             |         |  |      |            |       |     |
     |/               |        |c |      |           |c      |     |
  ___V___  /     ___V___    _V_____V_   |        ___V___   _V____V__
 | SYN-  |/ b''  | SYN-  |  |FIN-   |  e''       |       | | LAST-   |
 | SENT* |----->|RECEIVD*|---->| WAIT-1|---->|CLOSING|  | ACK   |
 |_____|      |_____|  |_____|         |_____| |_____|
                                |                 |           |
                               | f               | f         | f
                           ___V____          ___V___     ___V____
                          |FIN-   | e       |TIME-  | g |        |
                          | WAIT-2|---->| WAIT  |-->| CLOSED |
                          |_____|         |_____|   |_____|
```
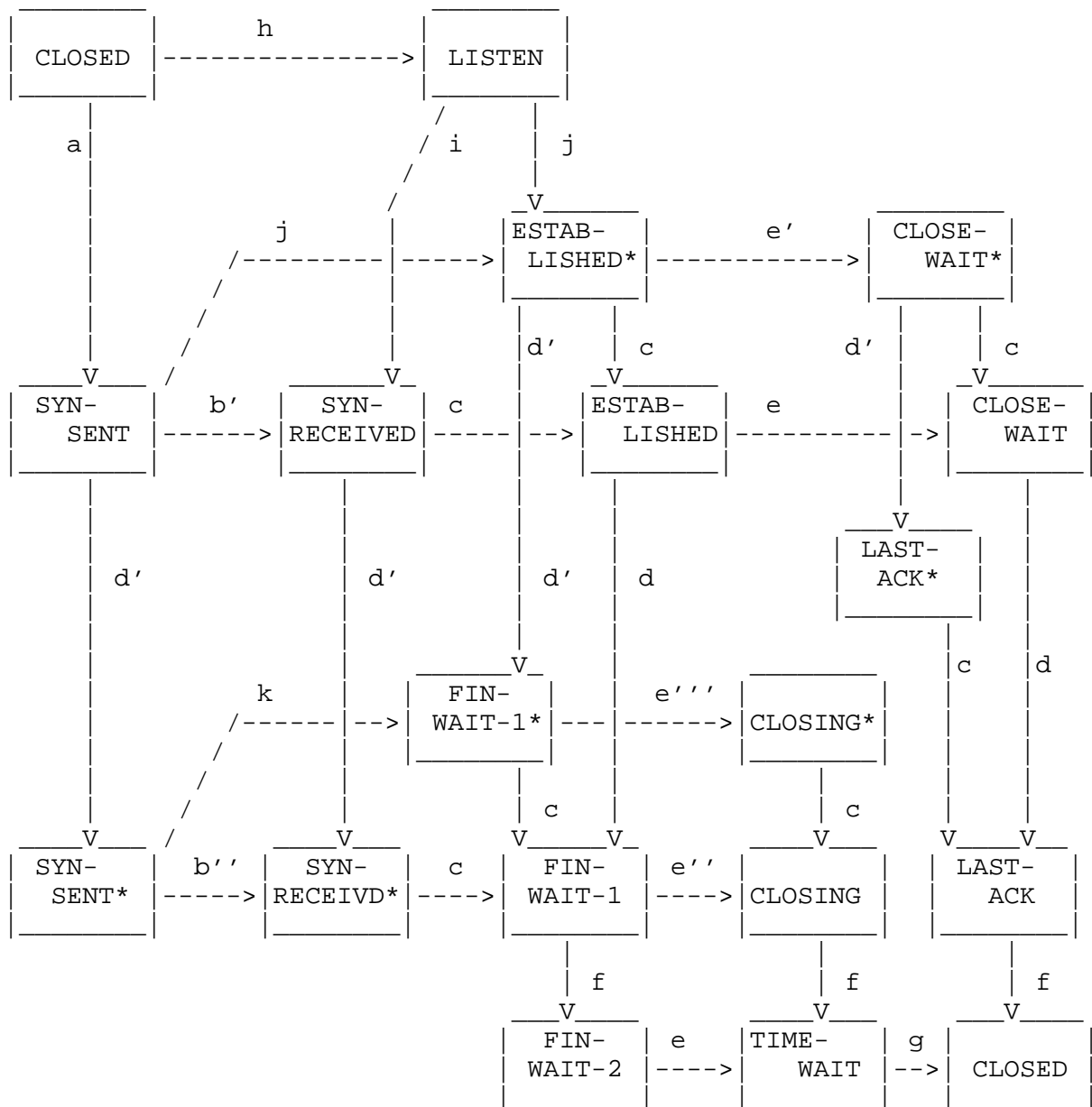
Figure 11: Complete State Diagram with TAO

The relationship between starred and unstarred states is very
regular. As a result, the state extensions can be implemented very
simply using the standard TCP FSM with the addition of two "hidden"
boolean flags, as described in the functional specification memo

[TTCP-FS].

As an example of the application of Figure 11, consider the minimal
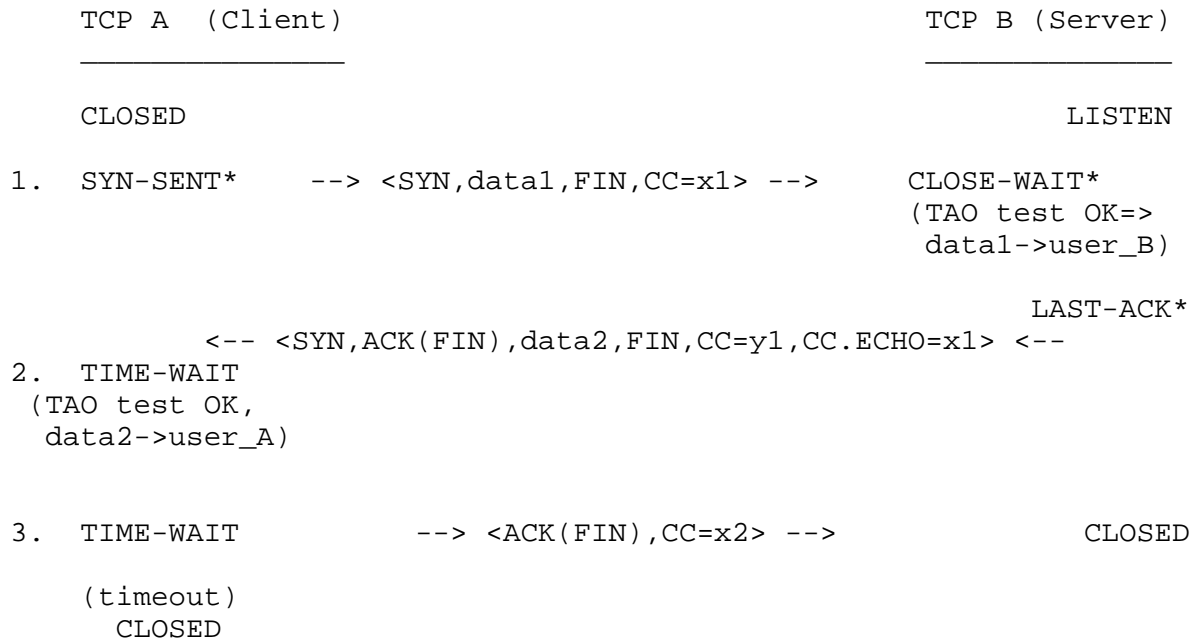transaction shown in Figure 12.

        TCP A (Client)   TCP B (Server)
        _____   _____

        CLOSED   LISTEN

1. SYN-SENT*   --> <SYN,data1,FIN,CC=x1> -->   CLOSE-WAIT*
                                                        (TAO test OK=>
                                                         data1->user_B)

                                                              LAST-ACK*
            <-- <SYN,ACK(FIN),data2,FIN,CC=y1,CC.ECHO=x1> <--
2. TIME-WAIT
   (TAO test OK,
    data2->user_A)

3. TIME-WAIT   --> <ACK(FIN),CC=x2> -->   CLOSED

     (timeout)
       CLOSED

             Figure 12: Minimal Transaction Sequence

Sending segment #1 leaves the client end in SYN-SENT* state, which
differs from SYN-SENT state in recording the fact that a FIN has been
sent. At the server end, passing the TAO test enters ESTABLISHED*
state, which passes the data to the user as in ESTABLISHED state and
also records the fact that the connection is half synchronized. Then
the server processes the FIN bit of segment #1, moving to CLOSE-WAIT*
state.

Moving to CLOSE-WAIT* state should cause the server to send a segment
containing SYN and ACK(FIN). However, transmission of this segment
is deferred so the server can piggyback the response data and FIN on
the same segment, unless a timeout occurs first. When the server
does send segment #2 containing the response data2 and a FIN, the
connection advances from CLOSE-WAIT* to LAST-ACK* state; the
connection is still half-synchronized from B's viewpoint.

Processing segment #2 at the client again results in multiple
transitions:

        SYN-SENT* -> FIN-WAIT-1* -> CLOSING* -> CLOSING -> TIME-WAIT

   These correspond respectively to receiving a SYN, a FIN, an ACK for
   A's SYN, and an ACK for A's FIN.

   Figure 13 shows a slightly more complex example, a transaction
   sequence in which request and response data each require two
   segments. This figure assumes that both client and server TCP are
   well-behaved, so that e.g., the client sends the single segment #5 to
   acknowledge both data segments #3 and #4. SEG.CC values are omitted
   for clarity.

        TCP A   TCP B

    1. SYN-SENT*   --> <SYN,data1>   -->   ESTABLISHED*
                                                      (TAO OK,
                                                       data1-> user)

    2. SYN-SENT*   --> <data2,FIN>   -->   CLOSE-WAIT*
                                                      (data2-> user)

    3. FIN-WAIT-2   <-- <SYN,ACK(FIN),data3> <-- CLOSE-WAIT*
        (data3->user)

    4. TIME_WAIT   <-- <ACK(FIN),data4,FIN> <--   LAST-ACK*
        (data4->user)

    5. TIME-WAIT   --> <ACK(FIN)> -->   CLOSED

            Figure 13. Multi-Packet Request/Response Transaction

7. CONCLUSIONS AND ACKNOWLEDGMENTS

   TCP was designed to be a highly symmetric protocol. This symmetry is
   evident in the piggy-backing of acknowledgments on data and in the
   common header format for data segments and acknowledgments. On the
   other hand, the examples and discussion in this memo are in general
   highly unsymmetrical; the actions of a "client" are clearly
   distinguished from those of a "server". To explain this apparent
   discrepancy, we note the following. Even when TCP is used for
   virtual circuit service, the data transfer phase is symmetrical but
   the open and close phases are not. A minimal transaction, consisting
   of one segment in each direction, compresses the open, data transfer,
   and close phases together, and making the asymmetry of the open and

close phases dominant. As request and response messages increase in
size, the virtual circuit model becomes increasingly relevant, and
symmetry again dominates.

TCP's 3-way handshake precludes any performance gain from including
data on a SYN segment, while TCP's full-duplex data-conserving close
sequence ties up communication resources to the detriment of high-
speed transactions. Merely loading more control bits onto TCP data
segments does not provide efficient transaction service. To use TCP
as an effective transaction transport protocol requires bypassing the
3-way handshake and shortening the TIME-WAIT delay. This memo has
proposed a backwards-compatible TCP extension to accomplish both
goals. It is our hope that by building upon the current version of
TCP, we can give a boost to community acceptance of the new
facilities. Furthermore, the resulting protocol implementations will
retain the algorithms that have been developed for flow and
congestion control in TCP [Jacobson88].

O'Malley and Peterson have recently recommended against backwards-
compatible extensions to TCP, and suggested instead a mechanism to
allow easy installation of alternative versions of a protocol [RFC-
1263]. While this is an interesting long-term approach, in the
shorter term we suggest that incremental extension of the current TCP
may be a more effective route.

Besides the backward-compatible extension proposed here, there are
two other possible approaches to making efficient transaction
processing widely available in the Internet: (1) a new version of TCP
or (2) a new protocol specifically adapted to transactions. Since
current TCP "almost" supports transactions, we favor (1) over (2). A
new version of TCP that retained the semantics of STD-007 but used 64
bit sequence numbers with the procedures and states described in
Sections 3, 4, and 6 of this memo would support transactions as well
as virtual circuits in a clean, coherent manner.

A potential application of transaction-mode TCP might be SMTP. If
commands and responses are batched, in favorable cases complete SMTP
delivery operations on short messages could be performed with a
single minimal transaction; on the other hand, the body of a message
may be arbitrarily large. Using a TCP extended as in this memo could
significantly reduce the load on large mail hosts.

This work began as an elaboration of the concept of TAO, due to Dave
Clark. I am grateful to him and to Van Jacobson, John Wroclawski,
Dave Borman, and other members of the End-to-End Research group for
helpful ideas and critiques during the long development of this work.
I also thank Liming Wei, who tested the initial implementation in Sun
OS.

APPENDIX A -- TIME-WAIT STATE AND THE 2-PACKET EXCHANGE

   This appendix considers the implications of reducing TIME-WAIT state
   delay below that given in formula [2].

   An immediate consequence of this would be the requirement for the
   server host to accept an initial SYN for a connection in LAST-ACK
   state. Without the transaction extensions, the arrival of a new
   <SYN> in LAST-ACK state looks to TCP like a half-open connection, and
   TCP's rules are designed to restore correspondence by destroying the
   state (through sending a RST segment) at one end or the other. We
   would need to thwart this action in the case of transactions.

   There are two different possible ways to further reduce TIME-WAIT
   delay.

   (1) Explicit Truncation of TIME-WAIT state

        TIME-WAIT state could be explicitly truncated by accepting a new
        sendto() request for a connection in TIME-WAIT state.

        This would allow the ACK(FIN) segment to be delayed and sent
        only if a timeout occurs before a new request arrives. This
        allows an ideal 2-segment exchange for closely-spaced
        transactions, which would restore some symmetry to the
        transaction exchange. However, explicit truncation would
        represent a significant change in many implementations.

        It might be supposed that even greater symmetry would result if
        the new request segment were a <SYN,ACK> that explicitly
        acknowledges the previous reply, rather than a <SYN> that is
        only an implicit acknowledgment. However, the new request
        segment might arrive at B to find the server side in either
        LAST-ACK or CLOSED state, depending upon whether the ACK(FIN)
        had arrived. In CLOSED state, a <SYN,ACK> would not be
        acceptable. Hence, if the client sent an initial <SYN,ACK>
        instead of a <SYN> segment, there would be a race condition at
        the server.

   (2) No TIME-WAIT delay

        TIME-WAIT delay could be removed entirely. This would imply
        that the ACK(FIN) would always be sent (which does not of course
        guarantee that it will be received). As a result, the arrival
        of a new SYN in LAST-ACK state would be rare.

        This choice is much simpler to implement. Its drawback is that
        the server will get a false failure report if the ACK(FIN) is

lost. This may not matter in practice, but it does represent a
significant change of TCP semantics. It should be noted that
reliable delivery of the reply is not an issue. The client
enter TIME-WAIT state only after the entire reply, including the
FIN bit, has been received successfully.

The server host B must be certain that a new request received in
LAST-ACK state is indeed a new SYN and not an old duplicate;
otherwise, B could falsely acknowledge a previous response that has
not in fact been delivered to A. If the TAO comparison succeeds, the
SYN must be new; however, the server has a dilemma if the TAO test
fails.

In Figure A.1, for example, the reply segment from the first
transaction has been lost; since it has not been acknowledged, it is
still in B's retransmission queue. An old duplicate request, segment
#3, arrives at B and its TAO test fails. B is in the position of
having old state it cannot discard (the retransmission queue) and
needing to build new state to pursue a 3-way handshake to validate
the new SYN. If the 3-way handshake failed, it would need to restore
the earlier LAST-ACK* state. (Compare with Figure 15 "Old Duplicate
SYN Initiates a Reset on Two Passive Sockets" in STD-007). This
would be complex and difficult to accomplish in many implementations.


```
        TCP A  (Client)                        TCP B (Server)
        _____                        _____

          CLOSED                                   LISTEN


   1.    SYN-SENT*        --> <SYN,data1,FIN> -->   CLOSE-WAIT*
                                                    (TAO test OK;
                                                     data1->server)

   2.        (lost) X<-- <SYN,ACK(FIN),data2,FIN> <-- LAST-ACK*

                 (old duplicate)
   3.                  ... <SYN,data3,FIN> -->    LAST-ACK*
                                                  (TAO test fail;
                                                   3-way handshake?)
```
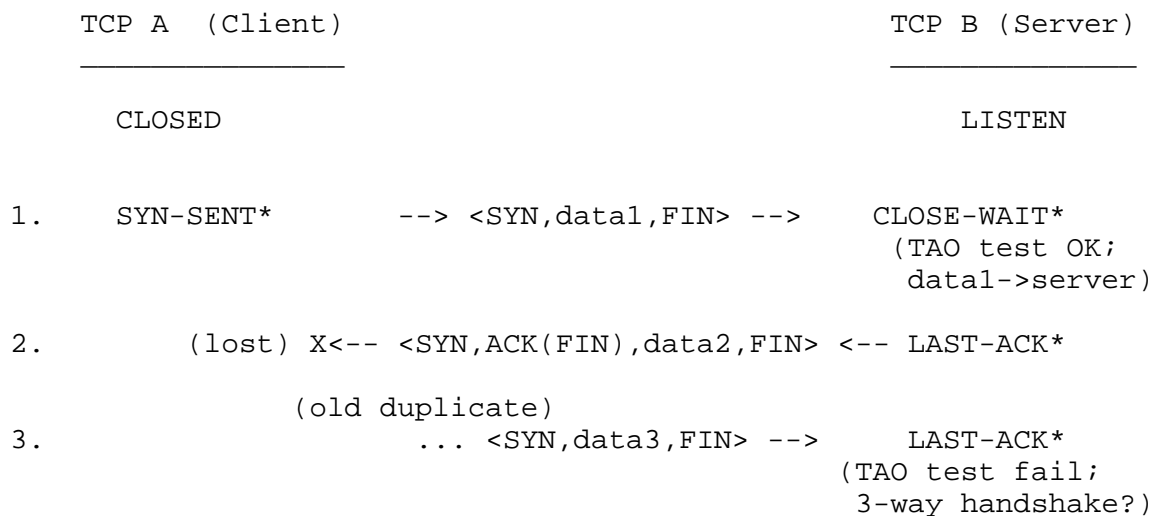
            Figure A.1: The Server's Dilemma

The only practical action A can taken when the TAO test fails on a
new SYN received in LAST-ACK state is to ignore the SYN, assuming it
is really an old duplicate. We must pursue the possible consequences

of this action.

Section 3.1 listed four possible reasons for failure of the TAO test
on a legitimate SYN segment: (1) no cached state, (2) out-of-order
delivery of SYNs, (3) wraparound of CCgen relative to the cached
value, or (4) the M values advance too slowly.   We are assuming that
there is a cached CC value at B (otherwise, the SYN cannot be
acceptable in LAST-ACK state). Wrapping the CC space is very
unlikely and probably impossible; it is difficult to imagine
circumstances which would allow the new SYN to be delivered but not
the ACK(FIN), especially given the long wraparound time of CCgen.

This leaves the problem of out-of-order delivery of two nearly-
concurrent SYNs for different ports. The second to be delivered may
have a lower CC option and thus be locked out. This can be solved by
using a new CCgen value for every retransmission of an initial SYN.

Truncation of TIME-WAIT state and acceptance of a SYN in LAST-ACK
state should take place only if there is a cached CC value for the
remote host. Otherwise, a SYN arriving in LAST-ACK state is to be
processed by normal TCP rules, which will result in a RST segment
from either A or B.

This discussion leads to a paradigm for rejecting old duplicate
segments that is different from TAO. This alternative scheme is
based upon the following:

(a) Each retransmission of an initial SYN will have a new value of
     CC, as described above.

     This provision takes care of reordered SYNs.

(b) A host maintains a distinct CCgen value for each remote host.
     This value could easily be maintained in the same cache used for
     the received CC values, e.g., as cache.CCgen[].

     Once the caches are primed, it should always be true that
     cache.CCgen[B] on host A is equal to cache.CC[A] on host B, and
     the next transaction from A will carry a CC value exactly 1
     greater. Thus, there is no problem of wraparound of the CC
     value.

(c) A new SYN is acceptable if its SEG.CC > cache.CC[client],
     otherwise the SYN is ignored as an old duplicate.

This alternative paradigm was not adopted because it would be a
somewhat greater perturbation of TCP rules, because it may not have
the robustness of TAO, and because all of its consequences may not be

understood.

REFERENCES

   [Birrell84] Birrell, A. and B. Nelson, "Implementing Remote
      Procedure Calls", ACM TOCS, Vo. 2, No. 1, February 1984.

   [Clark88] Clark, D., "The Design Philosophy of the Internet
      Protocols", ACM SIGCOMM '88, Stanford, CA, August 1988.

   [Clark89] Clark, D., Private communication, 1989.

   [Garlick77] Garlick, L., R. Rom, and J. Postel, "Issues in Reliable
      Host-to-Host Protocols", Proc. Second Berkeley Workshop on
      Distributed Data Management and Computer Networks, May 1977.

   [HR-COMM] Braden, R., Ed., "Requirements for Internet Hosts --
      Communication Layers", STD-003, RFC-1122, October 1989.

   [Jacobson88] Jacobson, V., "Congestion Avoidance and Control",
      SIGCOMM '88, Stanford, CA., August 1988.

   [Jacobson90] Jacobson, V., private communication, 1990.

   [Liskov90] Liskov, B., Shrira, L., and J. Wroclawski, "Efficient
      At-Most-Once Messages Based on Synchronized Clocks", ACM SIGCOMM
      '90, Philadelphia, PA, September 1990.

   [RFC-955] Braden, R., "Towards a Transport Service Transaction
      Protocol", RFC-955, September 1985.

   [RFC-1185] Jacobson, V., Braden, R., and Zhang, L., "TCP Extension
      for High-Speed Paths", RFC-1185, October 1990.

   [RFC-1263] O'Malley, S. and L. Peterson, "TCP Extensions Considered
      Harmful", RFC-1263, University of Arizona, October 1991.

   [RFC-1323] Jacobson, V., Braden, R., and Borman, D., "TCP
      Extensions for High Performance, RFC-1323, February 1991.

   [RFC-1337] Braden, R., "TIME-WAIT Assassination Hazards in TCP",
      RFC-1337, May 1992.

   [STD-007] Postel, J., "Transmission Control Protocol - DARPA
      Internet Program Protocol Specification", STD-007, RFC-793,
      September 1981.

   [TTCP-FS] Braden, R., "Transaction TCP -- Functional
      Specification", Work in Progress, September 1992.

   [Watson81] Watson, R., "Timer-based Mechanisms in Reliable
      Transport Protocol Connection Management", Computer Networks, Vol.
      5, 1981.

Security Considerations

   Security issues are not discussed in this memo.

Author's Address

   Bob Braden
   University of Southern California
   Information Sciences Institute
   4676 Admiralty Way
   Marina del Rey, CA 90292

   Phone: (310) 822-1511
   EMail: Braden@ISI.EDU