Network Working Group
Request for Comments: 1644
Category: Experimental

R. Braden
ISI

July 1994

T/TCP -- TCP Extensions for Transactions Functional Specification

Status of this Memo

Abstract

This memo specifies T/TCP, an experimental TCP extension for efficient transaction-oriented (request/response) service. This backwards-compatible extension could fill the gap between the current connection-oriented TCP and the datagram-based UDP.

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

Table of Contents

1. INTRODUCTION 2
2. OVERVIEW 3
   2.1 Bypassing the Three-Way Handshake 4
   2.2 Transaction Sequences 6
   2.3 Protocol Correctness 8
   2.4 Truncating TIME-WAIT State 12
   2.5 Transition to Standard TCP Operation 14
3. FUNCTIONAL SPECIFICATION 17
   3.1 Data Structures 17
   3.2 New TCP Options 17
   3.3 Connection States 19
   3.4 T/TCP Processing Rules 25
   3.5 User Interface 28
4. IMPLEMENTATION ISSUES 30
   4.1 RFC-1323 Extensions 30
   4.2 Minimal Packet Sequence 31
   4.3 RTT Measurement 31
   4.4 Cache Implementation 32
   4.5 CPU Performance 32
   4.6 Pre-SYN Queue 33
6. ACKNOWLEDGMENTS 34
7. REFERENCES 34
APPENDIX A. ALGORITHM SUMMARY 35

Security Considerations 38
Author’s Address 38

1. INTRODUCTION

TCP was designed to around the virtual circuit model, to support streaming of data. Another common mode of communication is a client-server interaction, a request message followed by a response message. The request/response paradigm is used by application-layer protocols that implement transaction processing or remote procedure calls, as well as by a number of network control and management protocols (e.g., DNS and SNMP). Currently, many Internet user programs that need request/response communication use UDP, and when they require transport protocol functions such as reliable delivery they must effectively build their own private transport protocol at the application layer.

Request/response, or "transaction-oriented", communication has the following features:

(a) The fundamental interaction is a request followed by a response.

(b) An explicit open or close phase may impose excessive overhead.

(c) At-most-once semantics is required; that is, a transaction must not be "replayed" as the result of a duplicate request packet.

(d) The minimum transaction latency for a client should be RTT + SPT, where RTT is the round-trip time and SPT is the server processing time.

(e) In favorable circumstances, a reliable request/response handshake should be achievable with exactly one packet in each direction.

This memo concerns T/TCP, an backwards-compatible extension of TCP to provide efficient transaction-oriented service in addition to virtual-circuit service. T/TCP provides all the features listed above, except for (e); the minimum exchange for T/TCP is three segments.

In this memo, we use the term "transaction" for an elementary request/response packet sequence. This is not intended to imply any of the semantics often associated with application-layer transaction processing, like 3-phase commits. It is expected that T/TCP can be used as the transport layer underlying such an application-layer service, but the semantics of T/TCP is limited to transport-layer services such as reliable, ordered delivery and at-most-once operation.

An earlier memo [RFC-1379] presented the concepts involved in T/TCP. However, the real-world usefulness of these ideas depends upon practical issues like implementation complexity and performance. To help explore these issues, this memo presents a functional specification for a particular embodiment of the ideas presented in RFC-1379. However, the specific algorithms in this memo represent a later evolution than RFC-1379. In particular, Appendix A in RFC-1379 explained the difficulties in truncating TIME-WAIT state. However, experience with an implementation of the RFC-1379 algorithms in a workstation later showed that accumulation of TCB's in TIME-WAIT state is an intolerable problem; this necessity led to a simple solution for truncating TIME-WAIT state, described in this memo.

Section 2 introduces the T/TCP extensions, and section 3 contains the complete specification of T/TCP. Section 4 discusses some implementation issues, and Appendix A contains an algorithmic summary. This document assumes familiarity with the standard TCP specification [STD-007].

2. OVERVIEW

The TCP protocol is highly symmetric between the two ends of a connection. This symmetry is not lost in T/TCP; for example, T/TCP supports TCP's symmetric simultaneous open from both sides (Section 2.3 below). However, transaction sequences use T/TCP in a highly unsymmetrical manner. It is convenient to use the terms "client host" and "server host" for the host that initiates a connection and the host that responds, respectively.

The goal of T/TCP is to allow each transaction, i.e., each request/response sequence, to be efficiently performed as a single incarnation of a TCP connection. Standard TCP imposes two performance problems for transaction-oriented communication. First, a TCP connection is opened with a "3-way handshake", which must complete successfully before data can be transferred. The 3-way handshake adds an extra RTT (round trip time) to the latency of a transaction.

The second performance problem is that closing a TCP connection leaves one or both ends in TIME-WAIT state for a time 2*MSL, where MSL is the maximum segment lifetime (defined to be 120 seconds). TIME-WAIT state severely limits the rate of successive transactions between the same (host,port) pair, since a new incarnation of the connection cannot be opened until the TIME-WAIT delay expires. RFC-1379 explained why the alternative approach, using a different user port for each transaction between a pair of hosts, also limits the transaction rate: (1) the 16-bit port space limits the rate to 2**16/240 transactions per second, and (2) more practically, an excessive amount of kernel space would be occupied by TCP state blocks in TIME-WAIT state [RFC-1379].

T/TCP solves these two performance problems for transactions, by (1) bypassing the 3-way handshake (3WHS) and (2) shortening the delay in TIME-WAIT state.

2.1 Bypassing the Three-Way Handshake

T/TCP introduces a 32-bit incarnation number, called a "connection count" (CC), that is carried in a TCP option in each segment. A distinct CC value is assigned to each direction of an open connection. A T/TCP implementation assigns monotonically increasing CC values to successive connections that it opens actively or passively.

T/TCP uses the monotonic property of CC values in initial <SYN> segments to bypass the 3WHS, using a mechanism that we call TCP Accelerated Open (TAO). Under the TAO mechanism, a host caches a small amount of state per remote host. Specifically, a T/TCP host that is acting as a server keeps a cache containing the last valid CC value that it has received from each different client host. If an initial <SYN> segment (i.e., a segment containing a SYN bit but no ACK bit) from a particular client host carries a CC value larger than the corresponding cached value, the monotonic property of CC’s ensures that the <SYN> segment must be new and can therefore be accepted immediately. Otherwise, the server host does not know whether the <SYN> segment is an old duplicate or was simply delivered out of order; it therefore executes a normal 3WHS to validate the <SYN>. Thus, the TAO mechanism provides an optimization, with the normal TCP mechanism as a fallback.

The CC value carried in non-<SYN> segments is used to protect against old duplicate segments from earlier incarnations of the same connection (we call such segments ’antique duplicates’ for short). In the case of short connections (e.g., transactions), these CC values allow TIME-WAIT state delay to be safely discuss in Section 2.3.

T/TCP defines three new TCP options, each of which carries one 32-bit CC value. These options are named CC, CC.NEW, and CC.ECHO. The CC option is normally used; CC.NEW and CC.ECHO have special functions, as follows.

(a) CC.NEW

Correctness of the TAO mechanism requires that clients generate monotonically increasing CC values for successive connection initiations. These values can be generated using a simple global counter. There are certain circumstances (discussed below in Section 2.2) when the client knows that monotonicity may be violated; in this case, it sends a CC.NEW rather than a CC option in the initial <SYN> segment. Receiving a CC.NEW causes the server to invalidate its cache entry and do a 3WHS.

(b) CC.ECHO

When a server host sends a <SYN,ACK> segment, it echoes the connection count from the initial <SYN> in a CC.ECHO option, which is used by the client host to validate the <SYN,ACK> segment.

Figure 1 illustrates the TAO mechanism bypassing a 3WHS. The cached CC values, denoted by cache.CC[host], are shown on each side. The server host compares the new CC value x in segment #1 against x0, its cached value for client host A; this comparison is called the "TAO test". Since x > x0, the <SYN> must be new and can be accepted immediately; the data in the segment can therefore be delivered to the user process B, and the cached value is updated. If the TAO test failed (x <= x0), the server host would do a normal three-way handshake to validate the <SYN> segment, but the cache would not be updated.

```
          TCP A  (Client)                            TCP B (Server)
          _______________                            ______________

                                                        cache.CC[A]
                                                          V

                                                        [ x0 ]

        #1        -->  <SYN, data1, CC=x> -->  (TAO test OK (x > x0) =>
                                                     data1->user_B and
                                                     cache.CC[A]= x; )

                                                           [ x ]
        #2       <-- <SYN, ACK(data1), data2, CC=y, CC.ECHO=x> <--
            (data2->user_A;)
```

Figure 1. TAO: Three-Way Handshake is Bypassed

The CC value x is echoed in a CC.ECHO option in the <SYN,ACK> segment (#2); the client side uses this option to validate the segment. Since segment #2 is valid, its data2 is delivered to the client user process. Segment #2 also carries B’s CC value; this is used by A to validate non-SYN segments from B, as explained in Section 2.4.

Implementing the T/TCP extensions expands the connection control block (TCB) to include the two CC values for the connection; call these variables TCB.CCsend and TCB.CCrecv (or CCsend, CCrecv for short). For example, the sequence shown in Figure 1 sets TCB.CCsend = x and TCB.CCrecv = y at host A, and vice versa at host B. Any segment that is received with a CC option containing a value SEG.CC different from TCB.CCsend will be rejected as an antique duplicate.

2.2 Transaction Sequences

T/TCP applies the TAO mechanism described in the previous section to perform a transaction sequence. Figure 2 shows a minimal transaction, when the request and response data can each fit into a single segment. This requires three segments and completes in one round-trip time (RTT). If the TAO test had failed on segment #1, B would have queued data1 and the FIN for later processing, and then it would have returned a <SYN,ACK> segment to A, to perform a normal 3WHS.

```
       TCP A  (Client)                                    TCP B (Server)
       _______________                                    ______________

       CLOSED                                                     LISTEN

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

                                                           <-- LAST-ACK*
   #2  TIME-WAIT   <-- <SYN,ACK(FIN),data2,FIN,CC=y,CC.ECHO=x>
     (data2->user_A)


   #3  TIME-WAIT          --> <ACK(FIN),CC=x> -->                 CLOSED

       (timeout)
         CLOSED
```

Figure 2: Minimal T/TCP Transaction Sequence

T/TCP extensions require additional connection states, e.g., the SYN-SENT*, CLOSE-WAIT*, and LAST-ACK* states shown in Figure 2. Section 3.3 describes these new connection states.

To obtain the minimal 3-segment sequence shown in Figure 2, the server host must delay acknowledging segment #1 so the response may be piggy-backed on segment #2. If the application takes longer than this delay to compute the response, the normal TCP retransmission mechanism in TCP B will send an acknowledgment to forestall a retransmission from TCP A. Figure 3 shows an example of a slow server application. Although the sequence in Figure 3 does contain a 3-way handshake, the TAO mechanism has allowed the request data to be accepted immediately, so that the client still sees the minimum latency.

```
       TCP A  (Client)                                    TCP B (Server)
       _______________                                    ______________

       CLOSED                                                     LISTEN

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

                                                               (timeout)
   #2  FIN-WAIT-1  <-- <SYN,ACK(FIN),CC=y,CC.ECHO=x> <--     CLOSE-WAIT*


   #3  FIN-WAIT-1      --> <ACK(SYN),FIN,CC=x> -->            CLOSE-WAIT


   #4  TIME-WAIT   <-- <ACK(FIN),data2,FIN,CC=y> <--            LAST-ACK
       (data2->user_A)

   #5  TIME_WAIT       --> <ACK(FIN),CC=x> -->                    CLOSED

         (timeout)
        CLOSED
```

Figure 3: Acknowledgment Timeout in Server

2.3 Protocol Correctness

This section fills in more details of the TAO mechanism and provides an informal sketch of why the T/TCP protocol works.

CC values are 32-bit integers. The TAO test requires the same kind of modular arithmetic that is used to compare two TCP sequence numbers. We assume that the boundary between y < z and z < y for two CC values y and z occurs when they differ by 2**31, i.e., by half the total CC space.

The essential requirement for correctness of T/TCP is this:

CC values must advance at a rate slower than 2**31 [R1]
counts per 2*MSL

where MSL denotes the maximum segment lifetime in the Internet. The requirement [R1] is easily met with a 32-bit CC. For example, it will allow 10**6 transactions per second with the very liberal MSL of 1000 seconds [RFC-1379]. This is well in excess of the transaction rates achievable with current operating systems and network latency.

Assume for the present that successive connections from client A to server B contain only monotonically increasing CC values. That is, if x(i) and x(i+1) are CC values carried in two successive initial <SYN> segments from the same host, then x(i+1) > x(i). Assuming the requirement [R1], the CC space cannot wrap within the range of segments that can be outstanding at one time. Therefore, those successive <SYN> segments from a given host that have not exceeded their MSL must contain an ordered set of CC values:

```
x(1) < x(2) < x(3) ... < x(n),
```

where the modular comparisons have been replaced by simple arithmetic comparisons. Here x(n) is the most recent acceptable <SYN>, which is cached by the server. If the server host receives a <SYN> segment containing a CC option with value y where y > x(n), that <SYN> must be newer; an antique duplicate SYN with CC value greater than x(n) must have exceeded its MSL and vanished. Hence, monotonic CC values and the TAO test prevent erroneous replay of antique <SYN>s.

There are two possible reasons for a client to generate non-monotonic CC values: (a) the client may have crashed and restarted, causing the generated CC values to jump backwards; or (b) the generated CC values may have wrapped around the finite space. Wraparound may occur because CC generation is global to all connections. Suppose that host A sends a transaction to B, then sends more than 2**31 transactions to other hosts, and finally sends another transaction to B. From B’s viewpoint, CC will have jumped backward relative to its cached value.

In either of these two cases, the server may see the CC value jump backwards only after an interval of at least MSL since the last <SYN> segment from the same client host. In case (a), client host restart, this is because T/TCP retains TCP’s explicit "Quiet Time" of an MSL interval [STD-007]. In case (b). wrap around, [R1] ensures that a time of at least MSL must have passed before the CC space wraps around. Hence, there is no possibility that a TAO test will succeed erroneously due to either cause of non-monotonicity; i.e., there is no chance of replays due to TAO.

However, although CC values jumping backwards will not cause an error, it may cause a performance degradation due to unnecessary 3WHS’s. This results from the generated CC values jumping backwards through approximately half their range, so that all succeeding TAO tests fail until the generated CC values catch up to the cached value. To avoid this degradation, a client host sends a CC.NEW option instead of a CC option in the case of either system restart or CC wraparound. Receiving CC.NEW forces a 3WHS, but when this 3WHS completes successfully the server cache is updated to the new CC value. To detect CC wraparound, the client must cache the last CC value it sent to each server. It therefore maintains cache.CCsent[B] for each server B. If this cached value is undefined or if it is larger than the next CC value generated at the client, then the client sends a CC.NEW instead of a CC option in the next SYN segment.

This is illustrated in Figure 4, which shows the scenario for the first transaction from A to B after the client host A has crashed and recovered. A similar sequence occurs if x is not greater than cache.CCsent[B], i.e., if there is a wraparound of the generated CC values. Because segment #1 contains a CC.NEW option, the server host invalidates the cache entry and does a 3WHS; however, it still sets B’s TCB.CCrecv for this connection to x. TCP B uses this CCrecv value to validate the <ACK> segment (#3) that completes the 3WHS. Receipt of this segment updates cache.CC[A], since the cache entry was previously undefined. (If a 3WHS always updated the cache, then out-of-order SYN segments could cause the cached value to jump backwards, possibly allowing replays). Finally, the CC.ECHO option in the <SYN,ACK> segment #2 defines A’s cache.CCsent entry.

This algorithm delays updating cache.CCsent[] until the <SYN> has been ACK’d. This allows the undefined cache.CCsent value to used as a a "first-time switch" to reliable resynchronization of the cached value at the server after a crash or wraparound.

When we use the term "cache", we imply that the value can be discarded at any time without introducing erroneous behavior although it may degrade performance.

(a) If a server host receives an initial <SYN> from client A but has no cached value cache.CC[A], the server simply forces a 3WHS to validate the <SYN> segment.

(b) If a client host has no cached value cache.CCsent[B] when it needs to send an initial <SYN> segment, the client simply sends a CC.NEW option in the segment. This forces a 3WHS at the server.

```
          TCP A  (Client)                               TCP B (Server)
          _______________                               ______________

          cache.CCsent[B]                                  cache.CC[A]
              V                                                 V

        (Crash and restart)
            [ ?? ]                                            [ x0 ]

        #1         --> <SYN, data1,CC.NEW=x> -->      (invalidate cache;
                                                            queue data1;
                                                        3-way handshake)

            [ ?? ]                                            [ ?? ]
        #2          <-- <SYN, ACK(data1),CC=y,CC.ECHO=x> <--
          (cache.CCsent[B]= x;)

            [ x ]                                             [ ?? ]

        #3                 --> <ACK(SYN),CC=x> -->       data1->user_B;
                                                         cache.CC[A]= x;

            [ x ]                                              [ x ]
```

Figure 4. Client Host Restarting

So far, we have considered only correctness of the TAO mechanism for bypassing the 3WHS. We must also protect a connection against antique duplicate non-SYN segments. In standard TCP, such protection is one of the functions of the TIME-WAIT state delay. (The other function is the TCP full-duplex close semantics, which we need to preserve; that is discussed below in Section 2.5). In order to achieve a high rate of transaction processing, it must be possible to truncate this TIME-WAIT state delay without exposure to antique duplicate segments [RFC-1379].

For short connections (e.g., transactions), the CC values assigned to each direction of the connection can be used to protect against antique duplicate non-SYN segments. Here we define "short" as a duration less than MSL. Suppose that there is a connection that uses the CC values TCB.CCsend = x and TCB.CCrecv = y. By the requirement [R1], neither x nor y can be reused for a new connection from the same remote host for a time at least 2*MSL. If the connection has been in existence for a time less than MSL, then its CC values will not be reused for a period that exceeds MSL, and therefore all antique duplicates with that CC value must vanish before it is reused. Thus, for "short" connections we can guard against antique non-SYN segments by simply checking the CC value in the segment againsts TCB.CCrecv. Note that this check does not use the monotonic property of the CC values, only that they not cycle in less than 2*MSL. Again, the quiet time at system restart protects against errors due to crash with loss of state.

If the connection duration exceeds MSL, safety from old duplicates still requires a TIME-WAIT delay of 2*MSL. Thus, truncation of TIME-WAIT state is only possible for short connections. (This problem has also been noticed by Shankar and Lee [ShankarLee93]). This difference in behavior for long and for short connections does create a slightly complex service model for applications using T/TCP. An application has two different strategies for multiple connections. For "short" connections, it should use a fixed port pair and use the T/TCP mechanism to get rapid and efficient transaction processing. For connections whose durations are of the order of MSL or longer, it should use a different user port for each successive connection, as is the current practice with unmodified TCP. The latter strategy will cause excessive overhead (due to TCB’s in TIME-WAIT state) if it is applied to high-frequency short connections. If an application makes the wrong choice, its attempt to open a new connection may fail with a "busy" error. If connection durations may range between long and short, an application may have to be able to switch strategies when one fails.

2.4 Truncating TIME-WAIT State

Truncation of TIME-WAIT state is necessary to achieve high transaction rates. As Figure 2 illustrates, a standard transaction leaves the client end of the connection in TIME-WAIT state. This section explains the protocol implications of truncating TIME-WAIT state, when it is allowed (i.e., when the connection has been in existence for less than MSL). In this case, the client host should be able to interrupt TIME-WAIT state to initiate a new incarnation of the same connection (i.e., using the same host and ports). This will send an initial <SYN> segment.

It is possible for the new <SYN> to arrive at the server before the retransmission state from the previous incarnation is gone, as shown in Figure 5. Here the final <ACK> (segment #3) from the previous incarnation is lost, leaving retransmission state at B. However, the client received segment #2 and thinks the transaction completed successfully, so it can initiate a new transaction by sending <SYN> segment #4. When this <SYN> arrives at the server host, it must implicitly acknowledge segment #2, signalling success to the server application, deleting the old TCB, and creating a new TCB, as shown in Figure 5. Still assuming that the new <SYN> is known to be valid, the server host marks the new connection half-synchronized and delivers data3 to the server application. (The details of how this is accomplished are presented in Section 3.3.)

The earlier discussion of the TAO mechanism assumed that the previous incarnation was closed before a new <SYN> arrived at the server. However, TAO cannot be used to validate the <SYN> if there is still state from the previous incarnation, as shown in Figure 5; in this case, it would be exceedingly awkward to perform a 3WHS if the TAO test should fail. Fortunately, a modified version of the TAO test can still be performed, using the state in the earlier TCB rather than the cached state.

(A) If the <SYN> segment contains a CC or CC.NEW option, the value SEG.CC from this option is compared with TCB.CCrecv, the CC value in the still-existing state block of the previous incarnation. If SEG.CC > TCB.CCrecv, the new <SYN> segment must be valid.

(B) Otherwise, the <SYN> is an old duplicate and is simply discarded.

Truncating TIME-WAIT state may be looked upon as composing an extended state machine that joins the state machines of the two incarnations, old and new. It may be described by introducing new intermediate states (which we call I-states), with transitions that join the two diagrams and share some state from each. I-states are detailed in Section 3.3.

Notice also segment #2’ in Figure 5. TCP’s mechanism to recover from half-open connections (see Figure 10 of [STD-007]) cause TCP A to send a RST when 2’ arrives, which would incorrectly make B think that the previous transaction did not complete successfully. The half-open recovery mechanism must be defeated in this case, by A ignoring segment #2’.

```
      TCP A  (Client)                                     TCP B (Server)
      _______________                                     ______________

      CLOSED                                                      LISTEN

  #1                --> <...,FIN,CC=x> -->                     LAST-ACK*

  #2         <-- <...ACK(FIN),data2,FIN,CC=y,CC.ECHO=x>  <---  LAST-ACK*
      TIME-WAIT
    (data2->user_A)


  #3  TIME-WAIT          --> <ACK(FIN),CC=x> --> X (DROP)

      (New Active Open)                           (New Passive Open)

  #4  SYN-SENT*    -->  <SYN, data3,CC=z> ...

                                                               LISTEN-LA
  #2' (discard) <-- <...ACK(FIN),data2,FIN,CC=y> <--- (retransmit)

  #4  SYN-SENT*        ... <SYN,data3,CC=z> -->           ESTABLISHED*
                                                    SYN OK (see text) =>
                                                            {Ack seg #2;
                                                         Delete old TCB;
                                                         Create new TCB;
                                                        data3 -> user_B;
                                                        cache.CC[A]= z;}
```

Figure 5: Truncating TIME-WAIT State: SYN as Implicit ACK

2.5 Transition to Standard TCP Operation

T/TCP includes all normal TCP semantics, and it will continue to operate exactly like TCP when the particular assumptions for transactions do not hold. There is no limit on the size of an individual transaction, and behavior of T/TCP should merge seamlessly from pure transaction operation as shown in Figure 2, to pure streaming mode for sending large files. All the sequences shown in [STD-007] are still valid, and the inherent symmetry of TCP is preserved.

Figure 6 shows a possible sequence when the request and response messages each require two segments. Segment #2 is a non-SYN segment that contains a TCP option. To avoid compatibility problems with existing TCP implementations, the client side should send segment #2 only if cache.CCsent[B] is defined, i.e., only if host A knows that host B plays the new game.

```
          TCP A  (Client)                                 TCP B (Server)
          _______________                                 ______________

          CLOSED                                                  LISTEN

       #1  SYN-SENT*       --> <SYN,data1,CC=x>  -->        ESTABLISHED*
                                                       (TAO test OK =>
                                                        data1-> user)

       #2  SYN-SENT*       --> <data2,FIN,CC=x>  -->         CLOSE-WAIT*
                                                       (data2-> user)

                                                             CLOSE-WAIT*
       #3  FIN-WAIT-2  <-- <SYN,ACK(FIN),data3,CC=y,CC.ECHO=x> <--
            (data3->user)

       #4  TIME_WAIT   <-- <ACK(FIN),data4,FIN,CC=y> <--       LAST-ACK*
            (data4->user)

       #5  TIME-WAIT       --> <ACK(FIN),CC=x> -->                CLOSED
```

Figure 6. Multi-Packet Request/Response Sequence

Figure 7 shows a more complex example, one possible sequence with TAO combined with simultaneous open and close. This may be compared with Figure 8 of [STD-007].

```
          TCP A                                                    TCP B
          _______________                                 ______________

          CLOSED                                                  CLOSED

      #1  SYN-SENT*         --> <SYN,data1,FIN,CC=x> ...

      #2  CLOSING*     <-- <SYN,data2,FIN,CC=y> <--            SYN-SENT*
          (TAO test OK =>
           data2->user_A

      #3  CLOSING*      --> <FIN,ACK(FIN),CC=x,CC.ECHO=y> ...

      #1’                       ... <SYN,data1,FIN,CC=x> -->    CLOSING*
                                                       (TAO test OK =>
                                                        data1->user_B)

      #4  TIME-WAIT   <-- <FIN,ACK(FIN),CC=y,CC.ECHO=x> <--     CLOSING*

      #5  TIME-WAIT    --> <ACK(FIN),CC=x> ...

      #3’              ... <FIN,ACK(FIN),CC=x,CC.ECHO=y> -->   TIME-WAIT

      #6  TIME-WAIT            <-- <ACK(FIN),CC=y> <---        TIME-WAIT

      #5’ TIME-WAIT               ... <ACK(FIN),CC=x> -->      TIME-WAIT

          (timeout)                                            (timeout)
            CLOSED                                                CLOSED
```

Figure 7: Simultaneous Open and Close

3. FUNCTIONAL SPECIFICATION

3.1 Data Structures

A connection count is an unsigned 32-bit integer, with the value zero excluded. Zero is used to denote an undefined value.

A host maintains a global connection count variable CCgen, and each connection control block (TCB) contains two new connection count variables, TCB.CCsend and TCB.CCrecv. Whenever a TCB is created for the active or passive end of a new connection, CCgen is incremented by 1 and placed in TCB.CCsend of the TCB; however, if the previous CCgen value was 0xffffffff (-1), then the next value should be 1. TCB.CCrecv is initialized to zero (undefined).

T/TCP adds a per-host cache to TCP. An entry in this cache for foreign host fh includes two CC values, cache.CC[fh] and cache.CCsent[fh]. It may include other values, as discussed in Sections 4.3 and 4.4. According to [STD-007], a TCP is not permitted to send a segment larger than the default size 536, unless it has received a larger value in an MSS (Maximum Segment Size) option. This could constrain the client to use the default MSS of 536 bytes for every request. To avoid this constraint, a T/TCP may cache the MSS option values received from remote hosts, and we allow a TCP to use a cached MSS option value for the initial SYN segment.

When the client sends an initial <SYN> segment containing data, it does not have a send window for the server host. This is not a great difficulty; we simply define a default initial window; our current suggestion is 4K. Such a non-zero default should be be conditioned upon the existence of a cached connection count for the foreign host, so that data may be included on an initial SYN segment only if cache.CC[foreign host] is non-zero.

In TCP, the window is dynamically adjusted to provide congestion control/avoidance [Jacobson88]. It is possible that a particular path might not be able to absorb an initial burst of 4096 bytes without congestive losses. If this turns out to be a problem, it should be possible to cache the congestion threshold for the path and use this value to determine the maximum size of the initial packet burst created by a request.

3.2 New TCP Options

Three new TCP options are defined: CC, CC.NEW, and CC.ECHO. Each carries a connection count SEG.CC. The complete rules for sending and processing these options are given in Section 3.4 below.

CC Option

Kind: 11

Length: 6

```
+--------+--------+--------+--------+--------+--------+
|00001011|00000110|    Connection Count:  SEG.CC      |
+--------+--------+--------+--------+--------+--------+
 Kind=11  Length=6
```

This option may be sent in an initial SYN segment, and it may be sent in other segments if a CC or CC.NEW option has been received for this incarnation of the connection. Its SEG.CC value is the TCB.CCsend value from the sender’s TCB.

CC.NEW Option

Kind: 12

Length: 6

```
+--------+--------+--------+--------+--------+--------+
|00001100|00000110|    Connection Count:  SEG.CC      |
+--------+--------+--------+--------+--------+--------+
 Kind=12  Length=6
```

This option may be sent instead of a CC option in an initial <SYN> segment (i.e., SYN but not ACK bit), to indicate that the SEG.CC value may not be larger than the previous value. Its SEG.CC value is the TCB.CCsend value from the sender’s TCB.

CC.ECHO Option

Kind: 13

Length: 6

```
+--------+--------+--------+--------+--------+--------+
|00001101|00000110|    Connection Count:  SEG.CC      |
+--------+--------+--------+--------+--------+--------+
 Kind=13  Length=6
```

This option must be sent (in addition to a CC option) in a segment containing both a SYN and an ACK bit, if the initial SYN segment contained a CC or CC.NEW option. Its SEG.CC value is the SEG.CC value from the initial SYN.

A CC.ECHO option should be sent only in a <SYN,ACK> segment and should be ignored if it is received in any other segment.

3.3 Connection States

T/TCP requires new connection states and state transitions. Figure 8 shows the resulting finite state machine; see [RFC-1379] for a detailed development. If all state names ending in stars are removed from Figure 8, the state diagram reduces to the standard TCP state machine (see Figure 6 of [STD-007]), with two exceptions:

- STD-007 shows a direct transition from SYN-RECEIVED to FIN-WAIT-1 state when the user issues a CLOSE call. This transition is suspect; a more accurate description of the state machine would seem to require the intermediate SYN-RECEIVED* state shown in Figure 8.

- In STD-007, a user CLOSE call in SYN-SENT state causes a direct transition to CLOSED state. The extended diagram of Figure 8 forces the connection to open before it closes, since calling CLOSE to terminate the request in SYN-SENT state is normal behavior for a transaction client. In the case that no data has been sent in SYN-SENT state, it is reasonable for a user CLOSE call to immediately enter CLOSED state and delete the TCB.

Each of the new states in Figure 8 bears a starred name, created by suffixing a star onto a standard TCP state. Each "starred" state bears a simple relationship to the corresponding "unstarred" state.

- SYN-SENT* and SYN-RECEIVED* differ from the SYN-SENT and SYN-RECEIVED state, respectively, in recording the fact that a FIN needs to be sent.

- The other starred states indicate that the connection is half-synchronized (hence, a SYN bit needs to be sent).

```
      ________      g        ________
     |        |<------------|        |
     | CLOSED |------------>| LISTEN |
     |________|  h    ------|________|
          |          /        |     |
          |         /        i|    j|
          |        /          |     |
         a|     a'/           |    _V______               ________
          |      /     j      |   |ESTAB-  |       e'    | CLOSE- |
          |     /  -----------|-->| LISHED*|------------>|   WAIT*|
          |    /  /           |   |________|             |________|
          |   /  /            |    |     |                |     |
          |  /  /             |    |    c|              d'|    c|
      ____V_V_ /       _______V    |   __V_____           |   __V_____
     | SYN-   |   b'  |  SYN-  |c  |  |ESTAB-  |  e       |  | CLOSE- |
     |   SENT |------>|RECEIVED|---|->|  LISHED|----------|->|   WAIT |
     |________|       |________|   |  |________|          |  |________|
        |               |          |     |                |        |
        |               |          |     |              __V_____   |
        |               |          |     |             | LAST-  |  |
      d'|             d'|        d'|    d|             |  ACK*  |  |
        |               |          |     |             |________|  |
        |               |          |     |                    |    |
        |               |    ______V_    |        ________    |c'  |d
        |          k    |   |  FIN-  |   |  e''' |        |   |    |
        |        -------|-->| WAIT-1*|---|------>|CLOSING*|   |    |
        |       /       |   |________|   |       |________|   |    |
        |      /        |          |     |            |       |    |
        |     /         |        c'|     |          c'|       |    |
     ___V___ /      ____V___       V_____V_       ____V___    V____V__
    | SYN-   | b'' |  SYN-  |  c  |  FIN-  | e'' |        |  | LAST-  |
    |  SENT* |---->|RECEIVD*|---->| WAIT-1 |---->|CLOSING |  |   ACK  |
    |________|     |________|     |________|     |________|  |________|
                                        |               |           |
                                       f|              f|         f'|
                                     ___V____      ____V___     ___V____
                                    |  FIN-  | e   |TIME-   | T |        |
                                    | WAIT-2 |---->|   WAIT |-->| CLOSED |
                                    |________|     |________|   |________|
```

Figure 8A: Basic T/TCP State Diagram

Label	Event / Action
a	Active OPEN / create TCB, snd SYN
a’	Active OPEN / snd SYN
b	rcv SYN [no TAO]/ snd ACK(SYN)
b’	rcv SYN [no TAO]/ snd SYN,ACK(SYN)
b’’	rcv SYN [no TAO]/ snd SYN,FIN,ACK(SYN)
c	rcv ACK(SYN) /
c’	rcv ACK(SYN) / snd FIN
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) /
f’	rcv ACK(FIN) / delete TCB
g	CLOSE / 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)
T	timeout=2MSL / delete TCB

Figure 8B. Definition of State Transitions

This simple correspondence leads to an alternative state model, which makes it easy to incorporate the new states in an existing implementation. Each state in the extended FSM is defined by the triplet:

```
    (old_state, SENDSYN, SENDFIN)
```

where ’old_state’ is a standard TCP state and SENDFIN and SENDSYN are Boolean flags see Figure 9. The SENDFIN flag is turned on (on the client side) by a SEND(... EOF=YES) call, to indicate that a FIN should be sent in a state which would not otherwise send a FIN. The SENDSYN flag is turned on when the TAO test succeeds to indicate that the connection is only half synchronized; as a result, a SYN will be sent in a state which would not otherwise send a SYN.

New state:		Old_state:	SENDSYN:	SENDFIN:
SYN-SENT*	=>	SYN-SENT	FALSE	TRUE
SYN-RECEIVED*	=>	SYN-RECEIVED	FALSE	TRUE
ESTABLISHED*	=>	ESTABLISHED	TRUE	FALSE
CLOSE-WAIT*	=>	CLOSE-WAIT	TRUE	FALSE
LAST-ACK*	=>	LAST-ACK	TRUE	FALSE
FIN-WAIT-1*	=>	FIN-WAIT-1	TRUE	FALSE
CLOSING*	=>	CLOSING	TRUE	FALSE

Figure 9: Alternative State Definitions

Here is a more complete description of these boolean variables.

* SENDFIN

  SENDFIN is turned on by the SEND(...EOF=YES) call, and turned off when FIN-WAIT-1 state is entered. It may only be on in SYN-SENT* and SYN-RECEIVED* states.

  SENDFIN has two effects. First, it causes a FIN to be sent on the last segment of data from the user. Second, it causes the SYN-SENT[*] and SYN-RECEIVED[*] states to transition directly to FIN-WAIT-1, skipping ESTABLISHED state.

* SENDSYN

  The SENDSYN flag is turned on when an initial SYN segment is received and passes the TAO test. SENDSYN is turned off when the SYN is acknowledged (specifically, when there is no RST or SYN bit and SEG.UNA < SND.ACK).

  SENDSYN has three effects. First, it causes the SYN bit to be set in segments sent with the initial sequence number (ISN). Second, it causes a transition directly from LISTEN state to ESTABLISHED*, if there is no FIN bit, or otherwise to CLOSE-WAIT*. Finally, it allows data to be received and processed (passed to the application) even if the segment does not contain an ACK bit.

According to the state model of the basic TCP specification [STD-007], the server side must explicitly issued a passive OPEN call, creating a TCB in LISTEN state, before an initial SYN may be accepted. To accommodate truncation of TIME-WAIT state within this model, it is necessary to add the five "I-states" shown in Figure 10. The I-states are: LISTEN-LA, LISTEN-LA*, LISTEN-CL, LISTEN-CL*, and LISTEN-TW. These are ’bridge states’ between two successive the state diagrams of two successive incarnations. Here D is the duration of the previous connection, i.e., the elapsed time since the connection opened. The transitions labeled with lower-case letters are taken from Figure 8.

Fortunately, many TCP implementations have a different user interface model, in which the use can issue a generic passive open ("listen") call; thereafter, when a matching initial SYN arrives, a new TCB in LISTEN state is automatically generated. With this user model, the I-states of Figure 10 are unnecessary.

For example, suppose an initial SYN segment arrives for a connection that is in LAST-ACK state. If this segment carries a CC option and if SEG.CC is greater than TCB.CCrecv in the existing TCB, the "q" transition shown in Figure 10 can be made directly from the LAST-ACK state. That is, the previous TCB is processed as if an ACK(FIN) had arrived, causing the user to be notified of a successful CLOSE and the TCB to be deleted. Then processing of the new SYN segment is repeated, using a new TCB that is generated automatically. The same principle can be used to avoid implementing any of the I-states.

```
 ______________________________
| P: Passive OPEN /            |
|                              |
| Q: Rcv SYN, special TAO test |                     d’|     d|
|     (see text) / Delete TCB, |    ________        ___V____  |
|     create TCB, snd SYN      |   |LISTEN- |  P   | LAST-  | |
|                              |   |   LA*  |<-----|  ACK*  | |
| Q’: (same as Q) if D < MSL   |   |________|      |________| |
|                              |    |     |            |      |
| R: Rcv ACK(FIN) / Delete TCB,|   Q|   c’|          c’|      |
|     create TCB               |    |     |            |      |
|                              |    |  ___V____        V______V
| S’: Active OPEN if D < MSL / |    | |LISTEN- |  P   | LAST-  |
|     Delete TCB, create TCB,  |    | |  LA    |<-----|   ACK  |
|     snd SYN.                 |    | |________|      |________|
|______________________________|    |  |     |            |
                                    | Q|    R|           f|
         ________        ________   |  |     |            |
   e’’’ |        |  P   |LISTEN- |  |  |     V            V
   ---->|CLOSING*|----->|   CL*  |  |  |    LISTEN       CLOSED
        |________|      |________|  |  |
             |            |   Q|    |  |
           c’|          c’|    V    V  V
             |            |   ESTABLISHED*
         ____V___         V_______
    e’’ |        |  P    |LISTEN- |
   ---->|CLOSING |------>|   CL   |
        |________|       |________|
             |           R|     Q|
            f|            V      V
             |         LISTEN   ESTABLISHED*
         ____V___                _________
     e  |TIME-   |  P           | LISTEN- |
   ---->|   WAIT |------------->|    TW   |
        |________|              |_________|
        /     |                  |    |  |
     S’/     T|                 T|  Q’|  |S’
      |  _____V_      h     _____V__  |  V
      | |        |-------->|        | |  SYN-SENT
      | | CLOSED |<--------| LISTEN | |
      | |________|   ------|________| |
      |   |        /        |   j|    |
      |  a|     a’/        i|    V    V
      |   |      /          |   ESTABLISHED*
      V   V     V           V
        SYN-SENT          ...
```

Figure 10: I-States for TIME-WAIT Truncation

3.4 T/TCP Processing Rules

This section summarizes the rules for sending and processing the T/TCP options.

INITIALIZATION

I1: All cache entries cache.CC[*] and cache.CCsent[*] are undefined (zero) when a host system initializes, and CCgen is set to a non-zero value.

I2: A new TCB is initialized with TCB.CCrecv = 0 and TCB.CCsend = current CCgen value; CCgen is then incremented. If the result is zero, CCgen is incremented again.

SENDING SEGMENTS

S1: Sending initial <SYN> Segment

An initial <SYN> segment is sent with either a CC option or a CC.NEW option. If cache.CCsent[fh] is undefined or if TCB.CCsend < cache.CCsent[fh], then the option CC.NEW(TCB.CCsend) is sent and cache.CCsent[fh] is set to zero. Otherwise, the option CC(TCB.CCsend) is sent and cache.CCsent[fh] is set to CCsend.

S2: Sending <SYN,ACK> Segment

If the sender’s TCB.CCrecv is non-zero, then a <SYN,ACK> segment is sent with both a CC(TCB.CCsend) option and a CC.ECHO (TCB.CCrecv) option.

S3: Sending Non-SYN Segment

A non-SYN segment is sent with a CC(TCB.CCsend) option if the TCB.CCrecv value is non-zero, or if the state is SYN-SENT or SYN-SENT* and cache.CCsent[fh] is non-zero (this last is required to send CC options in the segments following the first of a multi-segment request message; see segment #2 in Figure 6).

RECEIVING INITIAL <SYN> SEGMENT

Suppose that a server host receives a segment containing a SYN bit but no ACK bit in LISTEN, SYN-SENT, or SYN-SENT* state.

R1.1:If the <SYN> segment contains a CC or CC.NEW option, SEG.CC is stored into TCB.CCrecv of the new TCB.

R1.2:If the segment contains a CC option and if the local cache entry cache.CC[fh] is defined and if SEG.CC > cache.CC[fh], then the TAO test is passed and the connection is half-synchronized in the incoming direction. The server host replaces the cache.CC[fh] value by SEG.CC, passes any data in the segment to the user, and processes a FIN bit if present.

Acknowledgment of the SYN is delayed to allow piggybacking on a response segment.

R1.3:If SEG.CC <= cache.CC[fh] (the TAO test has failed), or if cache.CC[fh] is undefined, or if there is no CC option (but possibly a CC.NEW option), the server host proceeds with normal TCP processing. If the connection was in LISTEN state, then the host executes a 3-way handshake using the standard TCP rules. In the SYN-SENT or SYN-SENT* state (i.e., the simultaneous open case), the TCP sends ACK(SYN) and enters SYN-RECEIVED state.

R1.4:If there is no CC option (but possibly a CC.NEW option), then the server host sets cache.CC[fh] undefined (zero). Receiving an ACK for a SYN (following application of rule R1.3) will update cache.CC[fh], by rule R3.

Suppose that an initial <SYN> segment containing a CC or CC.NEW option arrives in an I-state (i.e., a state with a name of the form 'LISTEN-xx', where xx is one of TW, LA, L8, CL, or CL*):

R1.5:If the state is LISTEN-TW, then the duration of the current connection is compared with MSL. If duration > MSL then send a RST:

```
  <SEQ=0><ACK=SEG.SEQ+SEG.LEN><CTL=RST,ACK>
```

drop the packet, and return.

R1.6:Perform a special TAO test: compare SEG.CC with TCB.CCrecv.

If SEG.CC is greater, then processing is performed as if an ACK(FIN) had arrived: signal the application that the previous close completed successfully and delete the previous TCB. Then create a new TCB in LISTEN state and reprocess the SYN segment against the new TCB.

Otherwise, silently discard the segment.

RECEIVING <SYN,ACK> SEGMENT

Suppose that a client host receives a <SYN,ACK> segment for a connection in SYN-SENT or SYN-SENT* state.

R2.1:If SEG.ACK is not acceptable (see [STD-007]) and cache.CCsent[fh] is non-zero, then simply drop the segment without sending a RST. (The new SYN that the client is (re-)transmitting will eventually acknowledge any outstanding data and FIN at the server.)

R2.2:If the segment contains a CC.ECHO option whose SEG.CC is different from TCB.CCsend, then the segment is unacceptable and is dropped.

R2.3:If cache.CCsent[fh] is zero, then it is set to TCB.CCsend.

R2.4:If the segment contains a CC option, its SEG.CC is stored into TCB.CCrecv of the TCB.

RECEIVING <ACK> SEGMENT IN SYN-RECEIVED STATE

R3.1:If a segment contains a CC option whose SEG.CC differs from TCB.CCrecv, then the segment is unacceptable and is dropped.

R3.2:Otherwise, a 3-way handshake has completed successfully at the server side. If the segment contains a CC option and if cache.CC[fh] is zero, then cache.CC[fh] is replaced by TCB.CCrecv.

RECEIVING OTHER SEGMENT

R4: Any other segment received with a CC option is unacceptable if SEG.CC differs from TCB.CCrecv. However, a RST segment is exempted from this test.

OPEN REQUEST

To allow truncation of TIME-WAIT state, the following changes are made in the state diagram for OPEN requests (see Figure 10):

O1.1:A new passive open request is allowed in any of the states: LAST-ACK, LAST-ACK*, CLOSING, CLOSING*, or TIME-WAIT. This causes a transition to the corresponding I-state (see Figure 10), which retains the previous state, including the retransmission queue and timer.

O1.2 A new active open request is allowed in TIME-WAIT or LISTEN-TW state, if the elapsed time since the current connection opened is less than MSL. The result is to delete the old TCB and create a new one, send a new SYN segment, and enter SYN-SENT or SYN-SENT* state (depending upon whether or not the SYN segment contains a FIN bit).

Finally, T/TCP has a provision to improve performance for the case of a client that "sprays" transactions rapidly using many different server hosts and/or ports. If TCB.CCrecv in the TCB is non-zero (and still assuming that the connection duration is less than MSL), then the TIME-WAIT delay may be set to min(K*RTO, 2*MSL). Here RTO is the measured retransmission timeout time and the constant K is currently specified to be 8.

3.5 User Interface

STD-007 defines a prototype user interface ("transport service") that implements the virtual circuit service model [STD-007, Section 3.8]. One addition to this interface in required for transaction processing: a new Boolean flag "end-of-file" (EOF), added to the SEND call. A generic SEND call becomes:

```
Send

  Format:  SEND (local connection name, buffer address,
       byte count, PUSH flag, URGENT flag, EOF flag [,timeout])
```

The following text would be added to the description of SEND in [STD-007]:

> If the EOF (End-Of-File) flag is set, any remaining queued data is pushed and the connection is closed. Just as with the CLOSE call, all data being sent is delivered reliably before the close takes effect, and data may continue to be received on the connection after completion of the SEND call.

Figure 8A shows a skeleton sequence of user calls by which a client could initiate a transaction. The SEND call initiates a transaction request to the foreign socket (host and port) specified in the passive OPEN call. The predicate "recv_EOF" tests whether or not a FIN has been received on the connection; this might be implemented using the STATUS command of [STD-007], or it might be implemented by some operating-system-dependent mechanism. When recv_EOF returns TRUE, the connection has been completely closed and the client end of the connection is in TIME-WAIT state.

```
OPEN(local_port, foreign_socket, PASSIVE) -> conn_name;

SEND(conn_name, request_buffer, length,
                                   PUSH=YES, URG=NO, EOF=YES);

while (not recv_EOF(conn_name)) {

   RECEIVE(conn_name, reply_buffer, length) -> count;

   <Process reply_buffer.>
}
```

Figure 8A: Client Side User Interface

If a client is going to send a rapid series of such requests to the same foreign_socket, it should use the same local_port for all. This will allow truncation of TIME-WAIT state. Otherwise, it could leave local_port wild, allowing TCP to choose successive local ports for each call, realizing that each transaction may leave behind a significant control block overhead in the kernel.

Figure 8B shows a basic sequence of server calls. The server application waits for a request to arrive and then reads and processes it until a FIN arrives (recv_EOF returns TRUE). At this time, the connection is half-closed. The SEND call used to return the reply completes the close in the other direction. It should be noted that the use of SEND(... EOF=YES) in Figure 4B instead of a SEND, CLOSE sequence is only an optimization; it allows piggybacking the FIN in order to minimize the number of segments. It should have little effect on transaction latency.

```
OPEN(local_port, ANY_SOCKET, PASSIVE) -> conn_name;

<Wait for connection to open.>

STATUS(conn_name) -> foreign_socket

while (not recv_EOF(conn_name)) {

   RECEIVE(conn_name, request_buffer, length) -> count;

    <Process request_buffer.>
}

<Compute reply and store into reply_buffer.>

SEND(conn_name, reply_buffer, length,
                                 PUSH=YES, URG=NO, EOF=YES);
```

Figure 8B: Server Side User Interface

4. IMPLEMENTATION ISSUES

4.1 RFC-1323 Extensions

A recently-proposed set of TCP enhancements [RFC-1323] defines a Timestamps option, which carries two 32-bit timestamp values. This option is used to accurately measure round-trip time (RTT). The same option is also used in a procedure known as "PAWS" (Protect Against Wrapped Sequence) to prevent erroneous data delivery due to a combination of old duplicate segments and sequence number reuse at very high bandwidths. The approach to transactions specified in this memo is independent of the RFC-1323 enhancements, but implementation of RFC-1323 is desirable for all TCP's.

The RFC-1323 extensions share several common implementation issues with the T/TCP extensions. Both require that TCP headers carry options. Accommodating options in TCP headers requires changes in the way that the maximum segment size is determined, to prevent inadvertent IP fragmentation. Both require some additional state variable in the TCB, which may or may not cause implementation difficulties.

4.2 Minimal Packet Sequence

Most TCP implementations will require some small modifications to allow the minimal packet sequence for a transaction shown in Figure 2.

Many TCP implementations contain a mechanism to delay acknowledgments of some subset of the data segments, to cut down on the number of acknowledgment segments and to allow piggybacking on the reverse data flow (typically character echoes). To obtain minimal packet exchanges for transactions, it is necessary to delay the acknowledgment of some control bits, in an analogous manner. In particular, the <SYN,ACK> segment that is to be sent in ESTABLISHED* or CLOSE-WAIT* state should be delayed. Note that the amount of delay is determined by the minimum RTO at the transmitter; it is a parameter of the communication protocol, independent of the application. We propose to use the same delay parameter (and if possible, the same mechanism) that is used for delaying data acknowledgments.

To get the FIN piggy-backed on the reply data (segment #3 in Figure 2), thos implementations that have an implied PUSH=YES on all SEND calls will need to augment the user interface so that PUSH=NO can be set for transactions.

4.3 RTT Measurement

Transactions introduce new issues into the problem of measuring round trip times [Jacobson88].

(a) With the minimal 3-segment exchange, there can be exactly one RTT measurement in each direction for each transaction. Since dynamic estimation of RTT cannot take place within a single transaction, it must take place across successive transactions. Therefore, cacheing the measured RTT and RTT variance values is essential for transaction processing; in normal virtual circuit communication, such cacheing is only desirable.

(b) At the completion of a transaction, the values for RTT and RTT variance that are retained in the cache must be some average of previous values with the values measured during the transaction that is completing. This raises the question of the time constant for this average; quite different dynamic considerations hold for transactions than for file transfers, for example.

(c) An RTT measurement by the client will yield the value:

$$T = RTT + \min(SPT, ATO),$$

where SPT (server processing time) was defined in the introduction, and ATO is the timeout period for sending a delayed ACK. Thus, the measured RTT includes SPT, which may be arbitrarily variable; however, the resulting variability of the measured T cannot exceed ATO. (In a popular TCP implementation, for example, ATO = 200ms, so that the variance of SPT makes a relatively small contribution to the variance of RTT.)

(d) Transactions sample the RTT at random times, which are determined by the client and the server applications rather than by the network dynamics. When there are long pauses between transactions, cached path properties will be poor predictors of current values in the network.

Thus, the dynamics of RTT measurement for transactions differ from those for virtual circuits. RTT measurements should work correctly for very short connections but reduce to the current TCP algorithms for long-lasting connections. Further study is this issue is needed.

4.4 Cache Implementation

This extension requires a per-host cache of connection counts. This cache may also contain values of the smoothed RTT, RTT variance, congestion avoidance threshold, and MSS values. Depending upon the implementation details, it may be simplest to build a new cache for these values; another possibility is to use the routing cache that should already be included in the host [RFC-1122].

Implementation of the cache may be simplified because it is consulted only when a connection is established; thereafter, the CC values relevant to the connection are kept in the TCB. This means that a cache entry may be safely reused during the lifetime of a connection, avoiding the need for locking.

4.5 CPU Performance

TCP implementations are customarily optimized for streaming of data at high speeds, not for opening or closing connections. Jacobson's Header Prediction algorithm [Jacobson90] handles the simple common cases of in-sequence data and ACK segments when streaming data. To provide good performance for transactions, an implementation might be able to do an analogous "header prediction" specifically for the minimal request and the response segments.

The overhead of UDP provides a lower bound on the overhead of TCP-based transaction processing. It will probably not be possible to reach this bound for TCP transactions, since opening a TCP connection involves creating a significant amount of state that is not required by UDP.

McKenney and Dove [McKenney92] have pointed out that transaction processing applications of TCP can stress the performance of the demultiplexing algorithm, i.e., the algorithm used to look up the TCB when a segment arrives. They advocate the use of hash-table techniques rather than a linear search. The effect of demultiplexing on performance may become especially acute for a transaction client using the extended TCP described here, due to TCB’s left in TIME-WAIT state. A high rate of transactions from a given client will leave a large number of TCB’s in TIME-WAIT state, until their timeout expires. If the TCP implementation uses a linear search for demultiplexing, all of these control blocks must be traversed in order to discover that the new association does not exist. In this circumstance, performance of a hash table lookup should not degrade severely due to transactions.

4.6 Pre-SYN Queue

Suppose that segment #1 in Figure 4 is lost in the network; when segment #2 arrives in LISTEN state, it will be ignored by the TCP rules (see [STD-007] p.66, "fourth other text and control"), and must be retransmitted. It would be possible for the server side to queue any ACK-less data segments received in LISTEN state and to "replay" the segments in this queue when a SYN segment does arrive. A data segment received with an ACK bit, which is the normal case for existing TCP’s, would still a generate RST segment.

Note that queueing segments in LISTEN state is different from queueing out-of-order segments after the connection is synchronized. In LISTEN state, the sequence number corresponding to the left window edge is not yet known, so that the segment cannot be trimmed to fit within the window before it is queued. In fact, no processing should be done on a queued segment while the connection is still in LISTEN state. Therefore, a new "pre-SYN queue" would be needed. A timeout would be required, to flush the Pre-SYN Queue in case a SYN segment was not received.

Although implementation of a pre-SYN queue is not difficult in BSD TCP, its limited contribution to throughput probably does not justify the effort.

6. ACKNOWLEDGMENTS

I am very grateful to Dave Clark for pointing out bugs in RFC-1379 and for helping me to clarify the model. I also wish to thank Greg Minshall, whose probing questions led to further elucidation of the issues in T/TCP.

7. REFERENCES

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

[Jacobson90] Jacobson, V., "4BSD Header Prediction", Comp Comm Review, v. 20, no. 2, April 1990.

[McKenney92] McKenney, P., and K. Dove, "Efficient Demultiplexing of Incoming TCP Packets", ACM SIGCOMM ’92, Baltimore, MD, October 1992.

[RFC-1122] Braden, R., Ed., "Requirements for Internet Hosts -- Communications Layers", STD-3, RFC-1122, USC/Information Sciences Institute, October 1989.

[RFC-1323] Jacobson, V., Braden, R., and D. Borman, "TCP Extensions for High Performance, RFC-1323, LBL, USC/Information Sciences Institute, Cray Research, February 1991.

[RFC-1379] Braden, R., "Transaction TCP -- Concepts", RFC-1379, USC/Information Sciences Institute, September 1992.

[ShankarLee93] Shankar, A. and D. Lee, "Modulo-N Incarnation Numbers for Cache-Based Transport Protocols", Report CS-TR-3046/ UIMACS-TR-93-24, University of Maryland, March 1993.

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

APPENDIX A. ALGORITHM SUMMARY

This appendix summarizes the additional processing rules introduced by T/TCP. We define the following symbols:

Options

CC(SEG.CC):	TCP Connection Count (CC) Option
CC.NEW(SEG.CC):	TCP CC.NEW option
CC.ECHO(SEG.CC):	TCP CC.ECHO option

Here SEG.CC is option value in segment.

Per-Connection State Variables in TCB

CCsend:	CC value to be sent in segments
CCrecv:	CC value to be received in segments
Elapsed:	Duration of connection

Global Variables:

CCgen:	CC generator variable
cache.CC[fh]:	Cache entry: Last CC value received.
cache.CCsent[fh]:	Cache entry: Last CC value sent.

PSEUDO-CODE SUMMARY:

```
Passive OPEN => {
    Create new TCB;
}

Active OPEN => {
    <Create new TCB>
    CCrecv = 0;
    CCsend = CCgen;
    If (CCgen == 0xffffffff) then Set CCgen = 1;
                             else Set CCgen = CCgen + 1.
    <Send initial {SYN} segment (see below)>
}


Send initial {SYN} segment => {

    If (cache.CCsent[fh] == 0 OR CCsend < cache.CCsent[fh] ) then {

          Include CC.NEW(CCsend) option in segment;
          Set cache.CCsent[fh] = 0;
```

```
       }
       else {

             Include CC(CCsend) option in segment;
             Set cache.CCsent[fh] = CCsend;
       }
    }


   Send {SYN,ACK} segment => {

       If (CCrecv != 0) then
             Include CC(CCsend), CC.ECHO(CCrecv) options in segment.
   }


   Receive {SYN} segment in LISTEN, SYN-SENT, or SYN-SENT* state => {

       If state == LISTEN then {
             CCrecv = 0;
             CCsend = CCgen;
             If (CCgen == 0xffffffff) then Set CCgen = 1;
                                      else Set CCgen = CCgen + 1.
       }

       If (Segment contains CC option  OR
             Segment contains CC.NEW option) then
                   Set CCrecv = SEG.CC.

       if (Segment contains CC option  AND
             cache.CC[fh] != 0  AND
                   SEG.CC > cache.CC[fh] ) then {  /* TAO Test OK */

             Set cache.CC[fh] = CCrecv;
             <Mark connection half-synchronized>
             <Process data and/or FIN and return>
       }


       If (Segment does not contain CC option)  then
             Set cache.CC[fh] = 0;

       <Do normal TCP processing and return>.
   }

   Receive {SYN} segment in LISTEN-TW, LISTEN-LA, LISTEN-LA*, LISTEN-CL,
       or LISTEN-CL* state => {
```

```
       If ( (Segment contains CC option AND CCrecv != 0 )  then  {

             If (state = LISTEN-TW AND Elapsed > MSL ) then
                   <Send RST, drop segment, and return>.

             if (SEG.CC > CCrecv )  then {
                   <Implicitly ACK FIN and data in retransmission queue>;
                   <Close and delete TCB>;
                   <Reprocess segment>.
                           /* Expect to match new TCB
                            * in LISTEN state.
                            */
              }
       }
       else
             <Drop segment>.
   }


   Receive {SYN,ACK} segment => {

       if (Segment contains CC.ECHO option  AND
                   SEG.CC != CCsend) then
             <Send a reset and discard segment>.

       if (Segment contains CC option) then {
             Set CCrecv = SEG.CC.

             if (cache.CC[fh] is undefined) then
                   Set cache.CC[fh] = CCrecv.
       }
   }


   Send non-SYN segment => {

       if (CCrecv != 0  OR
             (cache.CCsent[fh] != 0  AND
              state is SYN-SENT or SYN-SENT*)) then
                  Include CC(CCsend) option in segment.
   }


   Receive non-SYN segment in SYN-RECEIVED state => {

       if (Segment contains CC option  AND  RST bit is off) {
               if (SEG.CC != CCrecv)  then
                     <Segment is unacceptable; drop it and send an
```

```
                       ACK segment, as in normal TCP processing>.

               if (cache.CC[fh] is undefined)  then
                     Set cache.CC[fh] = CCrecv.
       }
   }


   Receive non-SYN segment in (state >= ESTABLISHED) => {

       if (Segment contains CC option  AND  RST bit is off) {
               if (SEG.CC != CCrecv)  then
                     <Segment is unacceptable; drop it and send an
                       ACK segment, as in normal TCP processing>.
       }
   }
```

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